

ios 6

by tutorials

By the raywenderlich.com Tutorial Team

Adam Burkepile, Charlie Fulton, Matt Galloway, Jacob Gundersen, Kauserali Hafizji,
Matthijs Hollemans, Felipe Laso Marsetti, Marin Todorov, Brandon Trebitowski, Ray Wenderlich

iOS 6 By Tutorials

By the raywenderlich.com Tutorial Team

Adam [Burkepile](#), Charlie [Fulton](#), Matt [Galloway](#), Jacob [Gundersen](#), Kauserali [Hafizji](#), Matthijs [Hollemans](#), Felipe [Laso Marsetti](#), Marin [Todorov](#), Brandon [Trebitowski](#), Ray [Wenderlich](#)

Copyright © 2012 Razeware LLC.

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this book are the property of their respective owners.

Table of Contents

| | |
|--|------|
| Chapter 1: Introduction..... | 7 |
| Chapter 2: Programming in Modern Objective-C | 15 |
| Chapter 3: Beginning Auto Layout..... | 74 |
| Chapter 4: Intermediate Auto Layout..... | 189 |
| Chapter 5: Beginning UICollectionView | 297 |
| Chapter 6: Intermediate UICollectionView..... | 341 |
| Chapter 7: Beginning Passbook..... | 403 |
| Chapter 8: Intermediate Passbook | 485 |
| Chapter 9: Beginning In-App Purchases..... | 589 |
| Chapter 10: Intermediate In-App Purchases..... | 657 |
| Chapter 11: Beginning Social Framework..... | 745 |
| Chapter 12: Intermediate Social Framework | 781 |
| Chapter 13: Beginning Challenges with GameKit | 865 |
| Chapter 14: Intermediate Challenges with GameKit..... | 915 |
| Chapter 15: What's New with Attributed Strings..... | 959 |
| Chapter 16: State Preservation and Restoration..... | 1013 |
| Chapter 17: What's New with Core Image | 1073 |
| Chapter 18: What's New with MapKit | 1133 |
| Chapter 19: What's New with EventKit | 1183 |
| Chapter 20: What's New with Cocoa Touch..... | 1223 |
| Chapter 21: What's New with Storyboards | 1279 |
| Chapter 22: What's New with User Interface Customization | 1343 |
| Chapter 23: Beginning Automated Testing with XCode | 1385 |

| | |
|--|------|
| Chapter 24: Intermediate Automated Testing with Xcode..... | 1429 |
| Chapter 25: Accessibility | 1457 |
| Chapter 26: Secrets of Info.plist..... | 1473 |
| Chapter 27: Conclusion | 1523 |

Dedications

My parents, James and Donna, for putting up with me and stuff.
- Adam Burkepile

To my amazing wife Kristen and wonderful boys, Jack and Nick.
- Charlie Fulton

To everyone who has helped me along the way from moral support when making tough decisions to discussing challenging problems over a beer.

- Matt Galloway

To my wife, Traci. Thanks for helping me pursue a dream.
- Jacob Gundersen

To my parents and Batul - thanks for listening and believing in me.
- Kauserali Hafizji

To the crazy ones, the misfits, the rebels and the troublemakers.
- Matthijs Hollemans

To my nephews Leonardo and Tiziano, love you little guys to death. Also to mom, for her love and support in everything I do.
- Felipe Laso Marsetti

To my parents - ever so supportive and loving.
- Marin Todorov

To my incredible wife Ashely, and my two brilliant boys, Cayden and Jackson.

- Brandon Trebitowski

To my Mom and Dad, for believing in me and supporting me the entire way.

- Ray Wenderlich

Chapter 1: Introduction

iOS 6 introduces an abundance of great new APIs and technologies that all iOS developers should learn – from Auto Layout, to Collection Views, to Passbook, to the new literal syntax that makes code so much more compact – and that's just the start! In fact, there's so much new stuff that learning it all via the official API docs can be time-consuming and difficult – especially when you have other responsibilities beckoning.

This is where *iOS 6 by Tutorials* comes to the rescue! In this book, you will be able to learn these new topics in a much more efficient way – by building real apps through fun and easy-to-read tutorials!

And when it comes to tutorials, you're in the right place. The tutorials in this book guide you from being an iOS 6 API beginner to a master. You'll go far beyond the mere surface of a subject and will dive in deep – you'll find material here you simply won't find anywhere else. Through hands-on practice you'll gain the knowledge you need to apply these new APIs in your apps – and you'll have a ton of fun along the way! ☺

The raywenderlich.com Tutorial Team and I have been excited about iOS 6 from the very first day it was announced at WWDC. We hit the ground running, taking a "divide and conquer" approach where we each selected just a few topics to pursue with intense focus. This allowed each of us to spend our energy concentrating on a small set of topics in detail, rather than spreading our research too thin and tiring ourselves out in the process.

In the end, this is a huge benefit for YOU – you'll be able to learn these new APIs and technologies from someone who's really spent a lot of time digging into it, in an easy and fun way. You'll learn the material in a fraction of the time it took us, and avoid much of the learning curve and common mistakes.

So get ready for your personal tour through the amazing new features of iOS 6! By the time you are done, your iOS knowledge will be completely up-to-date, and you'll be ready to add this exciting, spanking-new functionality to your own apps.

Sit back, relax, and prepare for some fun and informative tutorials!

Who this book is for

This book is for intermediate or advanced iOS developers, who already know the basics of iOS development but want to upgrade their skills for iOS 6.

If you're a complete beginner, you can follow along with this book as well, because the tutorials always walk you through in a step-by-step manner. But there may be some missing gaps in your knowledge. To fill in these gaps, you might want to go through the *iOS Apprentice* series available through the [raywenderlich.com store](http://raywenderlich.com/store).

iOS 5 by Tutorials

Last year, the Tutorial Team came out with its first book, *iOS 5 by Tutorials*. The book was a great success, hence the writing of this book – effectively its sequel.

Note that *iOS 5 by Tutorials* and *iOS 6 by Tutorials* are completely different books, with completely different content.

iOS 5 by Tutorials covers the new APIs introduced in iOS 5, and similarly *iOS 6 by Tutorials* covers the new APIs introduced in iOS 6. There are so many new APIs introduced with each version of iOS, that each release warrants an entirely new book with unique chapters.

iOS 5 by Tutorials is not required reading for this book, although if you haven't mastered the new APIs introduced in iOS 5 (like Storyboards, ARC, or Core Image), reading it would definitely help. In many respects, this book picks up in knowledge where the other book left off.

How to use this book

This is the biggest book we've ever written – currently at over 1,200 pages and counting! It's the equivalent of two or more average books in terms of page count – making this book a great value, packed with content.

This book is large because we wanted to provide you with the most detailed information possible on these topics, so that you really understand how to use these APIs in your own apps. However, we realize everyone has busy schedules, and it is unlikely most readers will have time to read this entire massive tome cover-to-cover.

As such, we suggest a pragmatic approach – pick and choose the subjects that interest you the most, or the ones you need for your projects, and jump directly to

those chapters. Most chapters are self-contained, so there will not be a problem if you go through them in non-sequential order.

Looking for some recommendations of important chapters to start with? Here's our suggested Core Reading List:

- Chapter 2, "Programming in Modern Objective-C"
- Chapter 3, "Beginning Auto Layout"
- Chapter 5, "Beginning UICollectionView"
- Chapter 7, "Beginning Passbook"

That covers the "Big 4" topics of iOS 6, and from there you can dig into other topics that are particularly interesting to you.

Book overview

iOS 6 has a host of killer new APIs that you'll want to start using in your apps right away. Here's what you'll be learning about in this book:

Programming in Modern Objective-C

If you've been developing on iOS for a while, a lot has changed with Objective-C since you first began. You no longer need to declare instance variables for properties, you can add private class methods via class extensions, and gone are the days of pre-declaring and ordering methods.

And now with Xcode 4.5's new Objective-C features, such as auto-synthesizing variables and literal syntax, you can write the simplest, easiest-to-read code of your iOS life. Read this chapter to modernize your coding style and take advantage of all that the compiler has to offer!

Beginning and Intermediate Auto Layout

Remember positioning your views and controls with the autosizing attributes in Interface Builder? Well, there's a new kid in town – Auto Layout – and with it comes a better way.

Auto Layout is one of the biggest – and also most confusing – aspects of iOS 6. So we have dedicated two huge chapters to guide you through learning Auto Layout with practice and experimentation. By the time you're done, you'll feel comfortable using Auto Layout to make your apps easier to localize and your views more adaptable to different sizes!

Beginning and Intermediate UICollectionView

If you're like most iOS developers and love `UITableView`, then you're in for a treat in iOS 6 with the new `UICollectionView`! This control is similar to `UITableView` in that it

is an easy way to present collections of data, but `UICollectionView` comes with much more flexibility in how it lays out its items. With it, you can create a grid view, a “cover flow style” view, a “stacked photo view,” and more. Check out these chapters to get some hands-on experience with one of the most useful new APIs in iOS 6!

Beginning and Intermediate Passbook

Passbook is a new app in iOS 6 that can keep track of “things you keep in your pocket” – think store loyalty cards, gift cards, event tickets, boarding passes, and other flat, rectangular things you use to “pass” through life. The best thing about Passbook is the possibility of creating your own passes, and even dynamically updating them via your own servers and push notifications.

In these chapters, we take a deeper dive into this subject than you’ll find anywhere else – and by the end, you’ll have your own server-updated store credit pass system!

Beginning and Intermediate In-App Purchases

These days, over 70% of the top-grossing apps in the App Store use In-App Purchases, so this technology is increasingly important to know and understand. iOS 6 introduces some amazing new features with In-App Purchases: the ability to host downloads on Apple’s servers, and the ability to sell items directly from the iTunes Store within your app.

In these chapters, we walk you through the process of selling content in your app via In-App Purchases, from the very beginning – and in the end, you’ll have created a completely dynamic server-side system!

Beginning and Intermediate Social Framework

In iOS 5, Apple introduced the “tweet sheet controller,” allowing you to easily send tweets from within your app. Now in iOS 6, this has evolved a step further through the introduction of a social framework that you can use for all kinds of social network integration.

In these chapters, you’ll learn how to use this new framework to send Tweets, post to Facebook, and more – and we’ll show you how to access the underlying APIs to do custom tasks, such as allowing the user to “like” a Facebook page by tapping a button!

Beginning and Intermediate Challenges with GameKit

iOS 6 introduces a number of improvements to GameKit, but by far the biggest is the new Challenges feature. With Challenges, you can play a game, rack up a score, and then “challenge” your friends to do better than you. This can be a good way to introduce more social aspects into your games and increase their chances of

going viral – and it's easy to integrate! In these chapters, you'll learn how to add Challenges to your games, and you'll build a cool server-based replay system.

Secondary Topics

The above list comprises the big topics introduced in iOS 6, but there are some smaller improvements to existing APIs that are also important to understand. First you'll learn about the new Attributed Strings and State Preservation and Restoration technology in iOS 6. Then you'll learn about the changes that have been made to Core Image, Cocoa Touch, Storyboards, User Interface Customization, Attributed Strings, MapKit, and EventKit. If you use any of these technologies, definitely check them out to get yourself up to speed!

Bonus Chapters

Just because we love you for supporting the work we do at raywenderlich.com, we have included three bonus chapters as well. ☺ These are on topics that are not necessarily new to iOS 6, but that we have not covered previously on the site – so we thought you might enjoy them!

There's a chapter on adding automated unit testing and continuous integration into your apps, a chapter on adding Accessibility to your apps, and a chapter on everything you ever wanted to know about Info.plist.

Book source code and forums

This book comes with the source code for each of the chapters – it's shipped with the PDF. Some of the chapters have starter projects or required resources, so you'll definitely want to have them on hand as you go through the book.

We've also set up an official forum for the book at raywenderlich.com/forums. This is a great place to ask any questions you have about the book or about iOS 6 in general, or to submit any errata you may find.

Acknowledgements

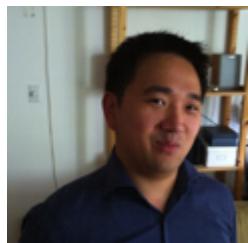
We would like to thank many people for their assistance in making this book possible:

- **Our families:** For bearing with us in this crazy time as we worked all hours of the night to get this book ready for publication!
- **Everyone at Apple:** For developing an amazing set of APIs, constantly inspiring us to improve our apps and skills, and making it possible for many developers to have their dream jobs!
- **Andrea Coulter:** For developing the layout of this book.

- **Adam Burkepile, Richard Casey, Adam Eberbach, Marcio Valenzuela, and Nick Waynik:** For being excellent forum moderators and for generously donating their time and expertise to the iOS community.
- And most importantly, **the readers of raywenderlich.com and you!** Thank you so much for reading our site and purchasing this book. Your continued readership and support is what makes this all possible!

About the authors

Harken all genders! You may or may not have noticed that all of this book's authors are men. This is unfortunate, and not by design. If you are a woman developing for iOS and are interested in joining the Tutorial Team, we'd love to hear from you! ☺



Adam Burkepile is a software developer with experience on many platforms and languages. He regularly writes tutorials for raywenderlich.com and moderates the forums. He also maintains a few projects at [github](https://github.com). When he's not on the computer, he enjoys drinking tea or doing Krav Maga.



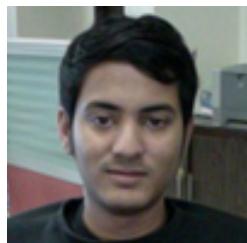
Charlie Fulton is a full time iOS developer. He has worked with many languages and technologies in the past 16 years, and is currently specializing in iOS and Cocos2D development. In his spare time, Charlie enjoys hunting, fishing, and hanging out with his family. You can follow him on Twitter as [@charlie_fulton](https://twitter.com/charlie_fulton).



Matt Galloway is a software developer specialising in mobile app development. In particular he enjoys creating awesome iOS apps and has worked on many including some top 10 apps. He writes on his blog at <http://www.galloway.me.uk/> and is founder of an app development company, Swipe Stack.



Jake Gunderson is a gamer, maker, and programmer. He is Co-Founder of the educational game company, Third Rail Games. He has a particular interest in gaming, image processing, and computer graphics. You can find his musings and codings at <http://indieambitions.com>.



Kauserali Hafizji (a.k.a. Ali) is a full time iOS developer and a tech lead working out of India. He is an avid programmer and loves writing code, even over the weekend. A good read, cool dip in the pool and a hot cheesy meal would be the perfect end to his weekend. You can find Ali on Twitter as [@Ali_hafizji](#).



Matthijs Hollemans is an independent designer and developer who loves to create awesome software for the iPad and iPhone. He also enjoys teaching others to do the same, which is why he wrote The iOS Apprentice series of ebooks. In his spare time, Matthijs is learning to play jazz piano (it's hard!) and likes to go barefoot running when the sun is out. Check out his blog at <http://www.hollance.com>.



Felipe Laso Marsetti is an iOS programmer working at Lextech Global Services. He loves everything related to Apple, video games, cooking and playing the violin, piano or guitar. In his spare time, Felipe loves to read and learn new programming languages or technologies. You can find him on Twitter as [@Airjordan12345](#).



Marin Todorov is an independent iOS developer and publisher, with background in various platforms and languages. He has published several books, written about iOS development on his blog, and authored an online game programming course. He loves to read, listen and produce music, and travel. Visit his web site: <http://www.touch-code-magazine.com>



Brandon Trebitowski is a software developer and author from Albuquerque, New Mexico. Brandon holds a BS in Computer Science from The University of New Mexico and has been developing software for the last 10 years. In 2010, he coauthored the book *iPhone & iPad In Action*. He is currently the Director of Mobile Engineering for ELC Technologies and a regularly blogs at <http://brandontreb.com>.

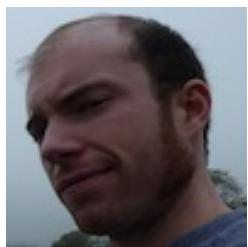


Ray Wenderlich is an iPhone developer and gamer, and the founder of [Razeware LLC](#). Ray is passionate about both making apps and teaching others the techniques to make them. He and the Tutorial Team have written a bunch of tutorials about iOS development available at <http://www.raywenderlich.com>.

About the Editors



Fahim Farook is a developer with over two decades of experience in developing in over a dozen different languages. Fahim's current focus is on the mobile app space with over 60 apps developed for iOS (iPhone and iPad). He's the CTO of [RookSoft Pte Ltd](#) of Singapore. Fahim has lived in Sri Lanka, USA, Saudi Arabia, New Zealand, and Singapore and enjoys science fiction and fantasy novels, TV shows, and movies. You can follow Fahim on [Twitter](#).



B.C. Phillips is an independent researcher and editor who splits his time between New York City and the Northern Catskills. He has many interests, but particularly loves cooking, eating, being active, thinking about deep questions, and working on his cabin and land in the mountains (even though his iPhone is pretty useless up there).

About the Artist



Vicki Wenderlich is a ceramic sculptor who was convinced two years ago to make art for her husband's iPhone apps. She discovered a love of digital art, and has been making app art and digital illustrations ever since. She is passionate about helping people pursue their dreams, and makes free app art for developers available on her website, <http://www.vickiwenderlich.com>.

Chapter 2: Programming in Modern Objective-C

By Matthijs Hollemans

Old habits die hard, but if you’re still writing Objective-C in the style that was practiced when iPhone OS 2.0 first came out (mid-2008), then you’re missing out on some really useful advancements that were made to the language recently.

Languages are living things, and computer languages like Objective-C are no exception. As far as programming languages go, Objective-C is already quite the elder – about 30 years old! For a while it was left to wither and die... until the success of the iPhone breathed new life into it. Amazingly, Objective-C is now one of the most popular languages in the world.

Ever since Apple took charge of the compiler tools used to create iOS apps (GCC in the past, now LLVM and Clang), the company has invested heavily in making the Objective-C faster, simpler and better. With every recent new version of Xcode, the language has seen refinements that make it easier for us to write our apps.

Because you may have missed some of these changes – a lot of books and code samples still cling to the old style of doing things due to habit or lack of updates – this chapter will show you how to make the most of these new techniques.

You will review the improvements that were made to Objective-C over the years – including the hot new stuff in Xcode 4.5 – and learn to take advantage of these features to make your code shorter (less typing!), simpler, and easier to debug.

After reading this chapter, you will have an elegant and modern Objective-C coding style you can be proud of. Read along – don’t let your programming style get rusty!

The Numberpedia app

In this chapter you are going to take an app that is written in old-fashioned Objective-C style from the time of iPhone OS 2.0 (also known as the “Stone Age”) and modernize it to take advantage of the exciting new features in Xcode 4.5 and the Clang 4.0 compiler.

The app itself is very simple. It's called "Numberpedia," and it consists of two screens. The first screen contains a table view that lists interesting numbers (that is, if you like numbers). Tap on a row and a "detail" screen opens that lets you edit the number, just in case you know better than Pythagoras.



You can find the source code for this app with this chapter's resources – open it up and take a quick look around before continuing.

You'll notice there are essentially two classes: `MasterViewController` for the main list and `DetailViewController` for the screen that lets you edit the numbers. The view controllers are designed in the Storyboard Editor and segue from one to the other.

The `MasterViewController` contains a `UITableView`, and most of what it does happens in the table view data source methods. If you have ever worked with a table view before, then this code will look very familiar.

Challenge: As you're looking through the sample project, see if you can spot some areas of code that aren't up-to-speed with the latest Objective-C features. Jot down the classes of problems that you find, for example "This project does not use ARC." Yeah, we'll give you one for free. ☺

At the end of the chapter, we'll check how many missing modern Objective-C techniques you caught and you can see if you're an Objective-C Padawan or a Jedi Master!

In this chapter, you are not going to change *what* the app does, only *how* it does it. After every change, you should build and run the app to see if it still works. At the end of the chapter, you will have an app that looks and works the same as before, but the code under the hood will be notably more elegant and functional, taking advantage of the latest features Objective-C has to offer.

This chapter won't teach you how to write better apps, but it will teach you how to write better source code!

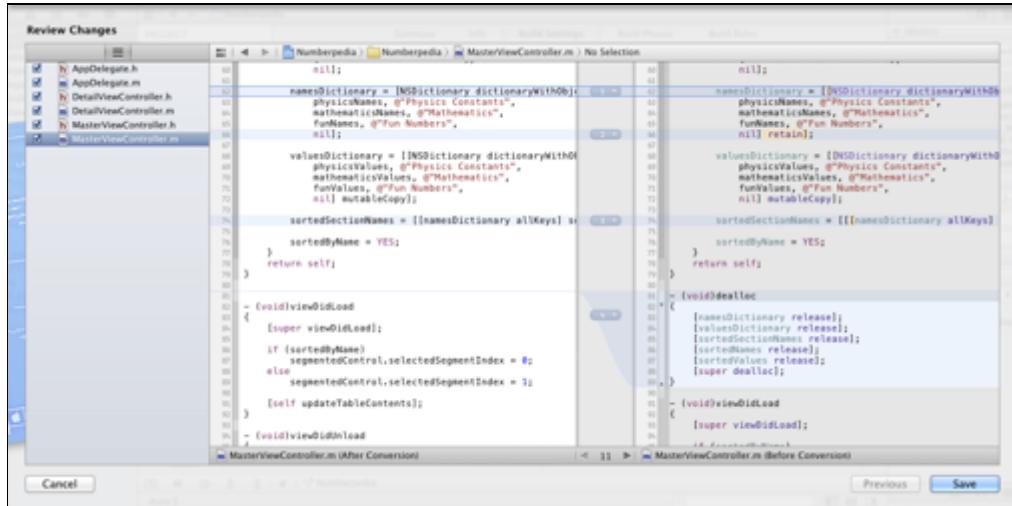
Make it automatic (ARC)

The most obvious place to start simplifying the Numberpedia source code is memory management. iOS 5 introduced a wonderful new technology called Automatic Reference Counting, or ARC for short. This takes Objective-C to a whole new level of easy. Where previously you had to type `retain`, `release` and `autorelease` to manage the lifetime of your objects, now you simply have to do... nothing. It doesn't get much easier than that.

To switch a project to ARC, you flip the build setting "Objective-C Automatic Reference Counting" to Yes and remove all the `retain`, `release` and `autorelease` statements from your code. It's possible to make these changes by hand, of course, but Xcode has a built-in conversion tool that makes the process much less painful.

There's really no reason not to be using ARC these days – it simplifies your code and makes it much less likely to contain memory leaks or crash issues. So let's try this out!

Go to the Edit menu and pick Refactor\Convert to Objective-C ARC... This presents you with a series of dialogs and then shows a preview of the changes that Xcode will make to your project:



(If Xcode gives an “ARC readiness error” instead, then make sure the Xcode scheme selector is set to “iPhone 6.0 Simulator” and not “iOS Device”, and try again.)

As you can see in the panel on the left, the `retain` statements will be removed, since they are no longer needed. The compiler is smart enough to automatically add `retain` statements in the proper places when it builds the app. Also note that the `dealloc` method gets removed. Because it’s no longer necessary to call `release` either, this particular `dealloc` method has nothing left to do, so Xcode strips it out.

Note: While ARC makes `dealloc` unnecessary for most code, you can still use `dealloc` if your code requires it. For instance, you might want to do some clean up, such as removing observers, when your class/object is deallocated. This would still require the use of a `dealloc` method.

Click the Save button to make the suggested changes. Congratulations, you now have a project that is ARC-enabled! Forever gone are the troubles of manual memory management. As an added bonus, the code is shorter because it has less to do.

The ARC conversion tool made a few other changes as well. Notably, the `@property` declarations now use the keyword `strong` rather than `retain`:

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

That is how ARC knows which objects to retain and which ones to release at any given time. You now express the relationships between your objects using `strong` or `weak` references.

A `strong` reference implies ownership and keeps the pointed-to object alive until the reference goes away. A `weak` reference is simply a pointer, with the added benefit

that it becomes `nil` when the pointed-to object has no more owners and is deallocated.



For a full introduction to ARC, refer to our book *iOS 5 by Tutorials*, as well as the free tutorials on raywenderlich.com:

- <http://www.raywenderlich.com/5677/beginning-arc-in-ios-5-part-1>
- <http://www.raywenderlich.com/5773/beginning-arc-in-ios-5-tutorial-part-2>

Fun with instance variables

Take a look at **MasterViewController.h**:

```
@interface MasterViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate,
DetailViewControllerDelegate>
{
    // Instance variables for outlets
    UITableView *tableView;
    UISegmentedControl *segmentedControl;

    . . .

}

@property (nonatomic, strong) IBOutlet UITableView *tableView;
@property (nonatomic, strong) IBOutlet UISegmentedControl
*segmentedControl;

. . .

@end
```

Usually when you declare a property, you want to have it “backed” by an instance variable that stores the actual value for that property. For example, `self.tableView`

(the property) actually reads and writes the value from the `tableView` instance variable.

In the `@interface` above, you can see that the author has declared both the property and its backing instance variable. The instance variable sits inside the `{ }` section, the property on its own line below that. (Sometimes you will see the `IBOutlet` specifier on the instance variable instead of the property. It doesn't really matter where it goes.)

When you do this, you're essentially writing the same thing twice. Here's the thing: this hasn't been necessary for ages! The explicit declaration of the instance variable was only necessary for the iPhone 2.x Simulator because it used an older version of the Objective-C runtime. Quite some time ago now, the Simulator switched to the "modern" runtime, which is also what the actual iPhone uses, and this workaround became unnecessary.

When you `@synthesize` a property, the compiler automatically creates that instance variable for you. That's what `@synthesize` is for, after all. So there is no need to type the same thing again.

Note: In some older versions of Xcode, it used to be that if you allowed the compiler to auto-create instance variables, you couldn't see the instance variables in the debugger. Happily, this is no longer the case, and you can now see the instance variables in the debugger as expected. So feel free to auto-generate!

So, go ahead and remove these two instance variable declarations from **MasterViewController.h**, and everything should work as before.

Also go ahead and remove all the instance variable declarations from **DetailViewController.h**. Same story. Each of those instance variables just exists for the sake of the property with the same name. Get rid of 'em.

The new, simplified `@interface` section from **DetailViewController.h** should look like this:

```
@interface DetailViewController : UIViewController

@property (nonatomic, strong) IBOutlet UINavigationBar *navigationBar;
@property (nonatomic, strong) IBOutlet UITextField *textField;

@property (nonatomic, weak) id <DetailViewControllerDelegate> delegate;

@property (nonatomic, copy) NSString *sectionName;
@property (nonatomic, assign) NSUInteger indexInSection;
```

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

- (IBAction)cancel:(id)sender;
- (IBAction)done:(id)sender;

@end
```

Build and run, and everything should work just as before!

But wait, there's more...

You're not done with the instance variables just yet. Currently the `@interface` of **MasterViewController.h** still declares several instance variables:

```
@interface MasterViewController : UIViewController <. . .>
{
    // Private instance variables
    NSDictionary *namesDictionary;
    NSMutableDictionary *valuesDictionary;
    NSArray *sortedSectionNames;

    // For the "sorted by value" screen
    BOOL sortedByName;
    NSArray *sortedNames;
    NSArray *sortedValues;
}
```

As the comment indicates, these are “private” instance variables. They are used internally by this view controller only, and are not supposed to be accessed by any objects outside this class. (Sometimes developers put a `@private` declaration in there as well.)

Wouldn’t it be better if outside objects didn’t know anything about these instance variables at all? In other words, why do they need to be exposed in the `@interface` section in the header file? There are good historical reasons why that was necessary, once upon a time – Objective-C having been built on top of the C language, to name one – but as of Xcode 4.2, the compiler no longer requires this. It is now possible to place instance variable declarations inside your implementation (.m) files instead.

Cut the instance variable section out of the header file and paste it directly below the `@implementation` line in **MasterViewController.m**.

The `@interface` section in **MasterViewController.h** should now look like this:

```
@interface MasterViewController : UIViewController <. . .>
```

```
@property (nonatomic, strong) IBOutlet UITableView *tableView;
@property (nonatomic, strong) IBOutlet UISegmentedControl
*segmentedControl;

- (IBAction)sortChanged:(UISegmentedControl *)sender;

@end
```

While the `@implementation` section in **MasterViewController.m** should now look like this:

```
@implementation MasterViewController
{
    NSDictionary *namesDictionary;
    NSMutableDictionary *valuesDictionary;
    NSArray *sortedSectionNames;

    // For the "sorted by value" screen
    BOOL sortedByName;
    NSArray *sortedNames;
    NSArray *sortedValues;
}
```

That is a lot cleaner! Instance variables are typically only necessary inside the .m file, so that's where they belong.

Note: You may wonder why some instance variables in this app have properties, and some do not. This is mostly a matter of style – some people like to create properties for everything, and some people don't like to create properties at all. I only tend to create properties for things that must be accessible from outside a class, and for `IBOutlets`.

To synthesize, or not to synthesize

Countless books and tutorials have probably drilled this rule into you: if you have a `@property` you need to `@synthesize` it, at least if you want it to be backed by an instance variable. It is also possible to create your own getter and setter methods or to use `@dynamic` properties, but most of the time you use `@synthesize`.

Well, thanks to the automatic `synthesize` feature in Xcode 4.5, you don't have to bother writing `@synthesize` statements anymore! The compiler will notice your `@property` statement and automatically `synthesize` the property and create the backing instance variable for you. Nice, eh?

Try this out by removing the `@synthesize` statements from **AppDelegate.m**, **MasterViewController.m** and **DetailViewController.m**. That shaves about nine or ten lines from the source code. Now build the app.

Whoops, the compiler isn't happy! It gives a number of errors in the code for **MasterViewController.m**, in this method:

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

    if (sortedByName)
        segmentedControl.selectedSegmentIndex = 0; // error!
    else
        segmentedControl.selectedSegmentIndex = 1; // error!

    [self updateTableContents];
}
```

The offending lines are the ones that refer to `segmentedControl`. This used to work before you removed `@synthesize`, so what's the big deal?

As it turns out, this is an example of programmer sloppiness. If you declare a property for something, then best practice says you should always refer to it as `self.property` and not directly through its backing instance variable. Using `self.property` invokes the proper getter and setter methods, but direct access through the backing instance variable skips those. That may cause issues if your getter or setter does anything special, beyond changing the backing variable.

It's best to always write `self.property` so you don't have to worry about any of this. Here, however, the programmer forgot to use "self" and just wrote `segmentedControl`. The fix is to simply add `self.` to the references to `segmentedControl`, as follows:

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

    if (sortedByName)
        self.segmentedControl.selectedSegmentIndex = 0;
    else
        self.segmentedControl.selectedSegmentIndex = 1;

    [self updateTableContents];
}
```

This still doesn't answer the question of why this code compiled without problems before you removed `@synthesize`. That `synthesize` statement looked like this:

```
@synthesize segmentedControl;
```

The statement above created a backing instance variable with the same name as the property: in this case, a variable also named `segmentedControl`. So before, you weren't actually going through the property (`self.segmentedControl`) – you were accessing the instance variable directly (`segmentedControl`). This is an easy mistake to make, since the property and instance variable have the same name.

You may have seen a variation of the `synthesize` statement that looks like:

```
@synthesize segmentedControl = _segmentedControl;
```

The above notation allows you to specify a different name for the instance variable. This is a good practice, because it makes it harder to make the above mistake – you access the property with `self.segmentedControl`, and the instance variable with `_segmentedControl`.

Also, when you do this the compiler helps you out, just like you saw here. Since your program referenced `segmentedControl` without `self`, the compiler gave an error because that is neither a valid way to access a property nor the name of an existing variable. It should either be `self.segmentedControl` or `_segmentedControl`, not just `segmentedControl`.

By renaming the instance variable, you prevent the situation where you're (mistakenly) using the backing instance variable directly, when you intended to use the property.

And that is exactly what auto-synthesize does: it creates a new backing instance variable named after the property, but prefixed with an underscore, just as if you had typed this:

```
@synthesize segmentedControl = _segmentedControl;
```

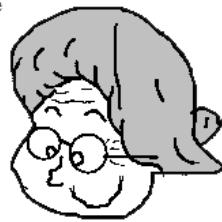
To verify this for yourself, change the offending lines to:

```
_segmentedControl.selectedIndex = . . . ;
```

Now the code should compile successfully. Even though you never declared this variable anywhere yourself, it still exists because of auto-synthesize. (You probably should change it back to use the property before you continue, though.)

Build and run, and everything should work as usual!

I've been @synthesizing
my properties since
before you were in
diapers, kid!



Tip: Your apps don't need to be iOS 6-only to take advantage of auto-synthesize. Apps compiled with this feature will still work all the way back to iOS 4. Nice!

What lurks in the shadows...

There is now another small fix you can make to the code. Some of the table view data source and delegate methods do the following:

```
- (UITableViewCell *)tableView:(UITableView *)theTableView  
    cellForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    UITableViewCell *cell = [theTableView  
        dequeueReusableCellWithIdentifier:@"NumberCell"];
```

Notice the name of the first parameter: `theTableView` instead of just `tableView`. This was done for the following reason: the author had an instance variable for the table view called `tableView`, but the method also took a parameter called `tableView`.

Because of this, there was a conflict – the parameter was said to “shadow” the instance variable, and the compiler generated a warning. To resolve the warning, the author renamed the parameter from `tableView` to `theTableView`.

This works, but it’s ugly! Luckily, since the compiler auto-synthesized your properties so that their backing instance variables begin with a prefix, this workaround is no longer necessary. Now the instance variable begins with an underscore (`_tableView`) and no longer conflicts with the parameter name.

So fix this up. Everywhere `theTableView` appears in the code, simply rename it `tableView`. That’s another wart removed!

Private categories and class extensions

The following code appears at the top of **MasterViewController.m**:

```
@interface MasterViewController (Private)
```

```
- (void)updateTableContents;
- (void)sortByValue;
@end
```

This is a *category*. Categories are an Objective-C feature that is used to add new methods to an existing class, even if you did not write that class yourself. In this case, the category is used to define *forward declarations* of certain methods.

In C and Objective-C, you're not allowed to call a method that hasn't been declared. Because the compiler reads source files from top to bottom, your code can only call methods that have been declared **above** the call. The `updateTableContents` method is called from `viewDidLoad`, but is defined somewhere below it in the source file.

To solve this conundrum, you can either shuffle the methods around, or tell the compiler about the existence of a method by placing a forward declaration in the `@interface`. For private methods, you don't want to place these forward declarations in the public header, so a common trick is to make a "Private" category and put the declarations in there.

That's the old school way of doing things. For some time now, Objective-C has had a special kind of category known as the *class extension* (also known as a *class continuation* or a *continuation class*) that serves exactly this purpose, plus it allows you to do a bunch of other cool stuff that categories don't.

Change the code at the top of **MasterViewController.m** to:

```
@interface MasterViewController ()
- (void)updateTableContents;
- (void)sortByValue;
@end
```

The only difference here is that there is no longer a name between the `()` parentheses. That makes it a class extension.

So how is this different from a regular category? First, the compiler will check to make sure that these methods are actually implemented in the `@implementation` section of the class (and throw a warning otherwise) – something it doesn't do for categories. Second, you can declare properties inside a class extension. You will see an example of that right now.

Earlier you moved the instance variables from the `@interface` section in the header to the `@implementation` section. You did this because there is no reason that instance variables, which are used only within the class, should be exposed to external objects. But what about the outlet properties - why would any other object in the program need to know about those?

Outlet properties exist to connect items from the nib or Storyboard to the view controller, but that is also an internal affair. It would be good if you could take

those properties out of the public interface and limit their visibility to within the class. You can do this by adding the Outlets to the class extension.

And don't worry – as long as these properties are marked as `IBOutlets`, Interface Builder and the Storyboard Editor are smart enough to find them even though they're in the class extension in the .m file. In other words, you still can connect stuff to them.

Remove the lines for the `IBOutlet` properties from **MasterViewController.h** and place them in the class extension at the top of **MasterViewController.m**:

```
@interface MasterViewController ()  
@property (nonatomic, strong) IBOutlet UITableView *tableView;  
@property (nonatomic, strong) IBOutlet UISegmentedControl  
                      *segmentedControl;  
  
- (void)updateTableContents;  
- (void)sortByValue;  
@end
```

Do the same thing for `DetailViewController`. You should now have a new class extension at the top of **DetailViewController.m**:

```
@interface DetailViewController ()  
@property (nonatomic, strong) IBOutlet UINavigationBar  
                      *navigationBar;  
@property (nonatomic, strong) IBOutlet UITextField *textField;  
@end
```

Note that you're only moving the `IBOutlet` properties, not the regular properties. You should leave the other properties inside the public `@interface` because you do want these to be accessible to other objects. After all, that is how the different view controllers communicate with each other. `MasterViewController`'s `prepareForSegue:` method sets these properties on `DetailViewController`. So the properties belong in a public `@interface`, but outlets usually don't.

Besides properties, it is also possible to put instance variables in the class extension. If you want to, you can even mark these variables as simply `IBOutlets`, freeing you from needing to make properties for your outlets. The class extension then looks like this:

```
@interface DetailViewController ()  
{  
    IBOutlet UINavigationBar *navigationBar;  
}  
  
@property (nonatomic, strong) IBOutlet UITextField *textField;
```

```
@end
```

Personally, I'm not a big fan of that. I like my outlets to be properties, because that makes it immediately clear whether the relationship with that outlet is strong or weak.

Forward declarations

Remember how I said that in C and Objective-C, you can't call methods that the compiler hasn't seen yet? Well, as I'm happy to inform you, that's no longer true. As of Xcode 4.3, the compiler is smart enough to figure out that your method exists, even if it's further ahead in the source file. What does that mean for you? Less code!

You can simply remove any forward declarations for your private methods from your class extension. The class extension from **MasterViewController.m** now becomes:

```
@interface MasterViewController ()  
@property (nonatomic, strong) IBOutlet UITableView *tableView;  
@property (nonatomic, strong) IBOutlet UISegmentedControl  
                      *segmentedControl;  
@end
```

The class extension now has only the properties, no longer the method names. Forward declarations for private methods are history as of Xcode 4.3. Another big time saver!

Note: This is only true for Objective-C methods. If you define your own C functions, you still need to provide forward declarations for those.

You may come across older code that has forward declarations for private methods in the public `@interface`, rather than in a "Private" category or class extension. Obviously, these are no longer necessary either (they didn't belong there in the first place). Be gone, methods!

Now don't go tossing everything from your public interface just yet – you do need to keep the methods that you want other classes to see. Most of the changes that you're making in this chapter are about simplifying the `@interface` section in the header file, so that it only contains the things you want other objects to know about.

Anything that is private to the class, that has to do with how the class works internally, gets moved into the .m file. That's the main difference between "old-fashioned" and modern Objective-C.

So you no longer have any private method declarations in your `@interface`, do you? Actually, you might. What about this line in **MasterViewController.h**?

```
- (IBAction)sortChanged:(UISegmentedControl *)sender;
```

That is the action method that gets called when the user interacts with the segmented control at the bottom of the screen. But is it something that other classes in the app need to know about? Not really: it's an implementation detail of the view controller. Therefore, this method declaration ought to go into the .m file.

But before you go moving it, check this: you just learned that forward declarations are no longer necessary. That's also true for `IBAction` methods. Interface Builder and the Storyboard Editor are smart enough to recognize that your `@implementation` already has this method, so you can do away with this line completely. Another method vanquished!

The new `@interface` section in **MasterViewController.h** looks like this:

```
@interface MasterViewController : UIViewController  
    <UITableViewDataSource, UITableViewDelegate,  
    DetailViewControllerDelegate>  
  
@end
```

That's a lot simpler already!

The same thing is true for **DetailViewController**. You can remove the following two lines from its header file:

```
- (IBAction)cancel:(id)sender;  
- (IBAction)done:(id)sender;
```

There's nothing left in my `@interface`!

Your quest to simplify the header file isn't over yet. There's almost nothing left in `MasterViewController.h`, but there are still some implementation details that remain exposed. Is it necessary for the user of `MasterViewController` to know that it conforms to the `DetailViewControllerDelegate`, table view data source and table view delegate protocols?

For some protocols this might be true, but in this case, all three protocols are important only to `MasterViewController` itself. You do need to declare somewhere that the `MasterViewController` class conforms to these protocols, but it doesn't have to be in the header.

You can simplify the header file to:

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

@interface MasterViewController : UIViewController

@end
```

That's right, all anyone else in the system needs to know is that `MasterViewController` is a `UIViewController`. Anything other than that is `MasterViewController`'s own business and no one else's.

So where do these protocol names go? Into the class extension, of course:

```
@interface MasterViewController () <UITableViewDataSource,
    UITableViewDelegate, DetailViewControllerDelegate>

@property (nonatomic, strong) IBOutlet UITableView *tableView;
@property (nonatomic, strong) IBOutlet UISegmentedControl
    *segmentedControl;
@end
```

Interface Builder and the Storyboard Editor still recognize that `MasterViewController` can act as the data source and delegate for the table view.

Because the `@interface` no longer references `DetailViewControllerDelegate`, you can even remove the `#import` statement for `DetailViewController.h` from the header. Put that `#import` in **MasterViewController.m** instead.

Speaking of `#imports`, you can even remove the `#import` for `UIKit.h` from the header files. The standard Xcode templates put that import there, but your projects also include a *pre-compiled header* file named "`<projectname>-Prefix.pch`" (where `<projectname>` is the actual name of your project) that already imports `UIKit` for the whole project. There is no reason to keep importing that framework in all your own headers.

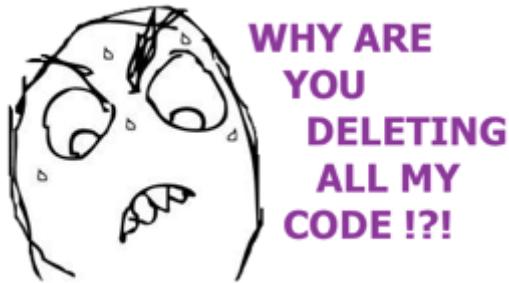
That means the entire source code for **MasterViewController.h** is now reduced to:

```
@interface MasterViewController : UIViewController

@end
```

It doesn't get much simpler than that.

If you like clean code as much as I do, then also remove the `#import` for `UIKit` from the other header files (except for `Numberpedia-Prefix.pch`, of course).



Handling low-memory situations

What I'm about to cover in this section isn't a new feature of Objective-C, but it is an important change with iOS 6 nonetheless.

Previously, if your app received a low-memory warning, the views of any view controllers that weren't visible at that point would get unloaded. The view controller was notified of this with the `viewDidUnload` message, and it was responsible for releasing any references to outlets so that they could get properly deallocated, in order to free up as much memory as possible.

This approach has changed a bit with iOS 6, and `viewDidUnload` will no longer be called (this method is now deprecated). This means your views will no longer be automatically unloaded for you – it's up to you if you want to do this, by doing so in `didReceiveMemoryWarning`.

You'll get to that in a bit, but first fix up your outlets. Right now all of the outlets are marked as `strong`; hence, there is a bunch of code in `viewDidUnload` that sets them to `nil` so that when the view is unloaded, there are no references to the subviews keeping them alive.

Switch the outlets to `weak` instead of `strong`, so that when the view is deallocated, nothing else will have a strong reference to the subviews, so they can be deallocated as well. The instance variables for the subviews will be automatically set to `nil` when the subviews are deallocated. (That's what `weak` properties do!)

To make these changes, first turn the outlets into weak pointers, in **MasterViewController.m**:

```
@interface MasterViewController () <UITableViewDataSource,  
UITableViewDelegate, DetailViewControllerDelegate>  
  
@property (nonatomic, weak) IBOutlet UITableView *tableView;  
@property (nonatomic, weak) IBOutlet UISegmentedControl  
*segmentedControl;  
@end
```

And in **DetailViewController.m**:

```
@interface DetailViewController ()  
@property (nonatomic, weak) IBOutlet UINavigationBar  
                      *navigationBar;  
@property (nonatomic, weak) IBOutlet UITextField *textField;  
@end
```

Remove `viewDidUnload` from **DetailViewController.m**, as it just sets these properties to nil and therefore no longer serves any purpose. Because the outlets are now weak references, they will become nil automatically when the main view is deallocated.

However, `viewDidUnload` in **MasterViewController.m** does a little more than that:

```
- (void)viewDidUnload  
{  
    [super viewDidUnload];  
  
    self.tableView = nil;  
    self.segmentedControl = nil;  
  
    sortedNames = nil;  
    sortedValues = nil;  
}
```

It not only sets the outlets to nil, but also the `sortedNames` and `sortedValues` instance variables. These two arrays contain the data for the "Sort by Value" tab on the app's main screen. The app only sorts this data the very first time you switch to this tab (a type of *lazy loading*). Essentially, `sortedNames` and `sortedValues` contain a cached version of the app's data that can be re-created when it is needed again.

In order to save as much memory as possible, `viewDidUnload` throws away this cached data. But because this method will no longer be called on iOS 6, you have to do this in `didReceiveMemoryWarning` instead.

Remove `viewDidUnload` and add this new method in its place to **MasterViewController.m**:

```
- (void)didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
  
    if ([self isViewLoaded] && self.view.window == nil)  
    {  
        self.view = nil;
```

```
    sortedNames = nil;
    sortedValues = nil;
}
}
```

First this method checks whether the view is still loaded. This is necessary because accessing the `self.view` property will always load the view if it isn't, and that's something you don't want to happen here! If the view is no longer part of the window, then you can safely unload it (and its subviews) by setting `self.view` to `nil`.

Second, this method gets rid of the cached data from the two arrays, replicating the old `viewDidUnload` behavior. The app should again behave the way it did on iOS 5 and earlier.

This sounds all well and good in theory, but I'd like to make sure that it actually does what it's supposed to. Fortunately, the simulator has a handy function that allows you to put low-memory situations to the test. Put some `NSLog()` statements in both `viewDidLoad` and `didReceiveMemoryWarning`, like this:

```
- (void)viewDidLoad
{
    NSLog(@"viewDidLoad");
    . . .
}

- (void)didReceiveMemoryWarning
{
    NSLog(@"didReceiveMemoryWarning");

    [super didReceiveMemoryWarning];

    if ([self isViewLoaded] && self.view.window == nil)
    {
        NSLog(@"will unload view");

        . . .
    }
}
```

Run the app. The debug output pane should now say something like:

```
Numberpedia [...] viewDidLoad
```

While still on the main screen, choose Hardware\Simulate Memory Warning from the simulator's menu. The output pane now says:

```
Numberpedia [...] Received memory warning.
```

```
Numberpedia[...] didReceiveMemoryWarning
```

It will not say “will unload view” because you added that check to make sure you don’t unload a visible view (its `window` is not nil). Now tap a row from the table so that the detail screen opens. Again, choose Hardware\Simulate Memory Warning. Now the output pane will say:

```
Numberpedia[...] Received memory warning.  
Numberpedia[...] didReceiveMemoryWarning  
Numberpedia[...] will unload view
```

Excellent! The app recognizes that the view from `MasterViewController` is no longer needed and can be unloaded. Just for fun, do another Simulate Memory Warning. The app will not say “will unload view” again, because there is no longer a view to unload.

Tap Cancel or Done to close the detail screen. Because the app now returns to the `MainViewController`, `UIKit` has to re-load its view and `viewDidLoad` should be called a second time:

```
Numberpedia[...] viewDidLoad
```

Of course, this requires that your `viewDidLoad` can handle such situations. That is why `Numberpedia` keeps track of the selection status of the segmented control using an instance variable, `sortedByName`. That way, the app can restore the proper segment when its view gets reloaded:

```
if (sortedByName)
    self.segmentedControl.selectedSegmentIndex = 0;
else
    self.segmentedControl.selectedSegmentIndex = 1;
```

Try it out. Select the “Sort by Value” tab and tap on a row. From the detail screen, fake a memory warning, and go back to the main screen. The Sort by Value tab should still be selected. To the user, it appears as if nothing ever changed, but behind the scenes, the app unloaded the view and restored it to its old state when it was needed again.



I don't like
*** WEAK ***
properties!

Now give me
20 push-ups!

Fun with blocks

Any chapter on the recent changes to Objective-C cannot neglect to mention blocks. If you've been shying away from them – they can look a little scary, it's true – then now is the time to learn how to use them. Blocks can simplify your code and make it a lot more expressive. Do more with less code™.

Did you know that blocks can be a good replacement for some delegates? If you've been programming with iOS for a while, then you've seen delegates before – probably in a nightmare or two. 😊

Delegates are a great design pattern, but they have the disadvantage of saddling you with snippets of code all over the place. That can sometimes make it hard to follow the flow of events.

For example, Numberpedia's detail screen is launched by a segue, so it is configured in `MasterViewController`'s `prepareForSegue:` method. But the communication from `DetailViewController` back to `MasterViewController` takes place using two delegate methods that are located in a completely different location in the source file. That makes the relationship between opening the detail screen and returning from it unclear. It would be nice to put all of this code in one place, and blocks can help with that.

In this part of the chapter, you will replace the `DetailViewControllerDelegate` protocol with a block. Remove the following lines from **DetailViewController.h**:

```
@class DetailViewController;

@protocol DetailViewControllerDelegate <NSObject>

- (void)detailViewControllerDidCancel:
    (DetailViewController *)controller;
- (void)detailViewControllerDidClose:
    (DetailViewController *)controller;

@end
```

Replace them with this line:

```
typedef void (^DetailViewControllerCompletionBlock)(BOOL
                                                 success);
```

This statement defines a new data type, `DetailViewControllerCompletionBlock`, that describes the block you will be using. A block is like a C function; it has a return type (in this case `void`) and zero or more parameters (here just one, a `BOOL` named `success`). The `typedef` makes it a bit easier to refer to this block in the code.

Remove the line for the delegate property and replace it with:

```
@property (nonatomic, copy) DetailViewControllerCompletionBlock  
completionBlock;
```

Thanks to the `typedef`, this looks just like any other property declaration. Without the `typedef`, it would have looked like this:

```
@property (nonatomic, copy) void (^completionBlock)(BOOL  
success);
```

That works, but it's also a bit harder to read.

Notice that the `completionBlock` property is marked `copy`. Due to the way blocks work, they need to be copied if you want to hold onto them. ARC will in many cases already do that for you behind-the-scenes, but I like my properties to be explicit about it.

In **DetailViewController.m**, the `cancel:` and `done:` action methods currently do the following:

```
- (IBAction)cancel:(id)sender  
{  
    [self.delegate detailViewControllerDidCancel:self];  
}  
  
- (IBAction)done:(id)sender  
{  
    . . .  
  
    [self.delegate detailViewControllerDidClose:self];  
}
```

That obviously no longer works because there is no longer a delegate for them to call. Replace the above lines with:

```
- (IBAction)cancel:(id)sender  
{  
    if (self.completionBlock != nil)  
        self.completionBlock(NO);  
}  
  
- (IBAction)done:(id)sender  
{  
    . . .  
  
    if (self.completionBlock != nil)  
        self.completionBlock(YES);  
}
```

Not only do blocks look like C functions, you also call them like C functions. The only trick is that you need to check for `nil` first. While it is okay in Objective-C to send messages to `nil` pointers, it is not allowed to call a block that is `nil`. Usually the caller of `DetailViewController` will have given it a proper `completionBlock`, but it's smart to do some defensive programming in any case.

Great, so this will call the completion block with a `success` value of `NO` if the user cancelled the screen, and `YES` if he pressed Done.

Now you need to change `MasterViewController` to pass along a block instead of using the delegate methods. Remove the `DetailViewControllerDelegate` protocol name from the class extension in **MasterViewController.m**, so that it reads:

```
@interface MasterViewController () <UITableViewDataSource,  
UITableViewDelegate>
```

The biggest change happens in `prepareForSegue`:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue  
           sender:(id)sender  
{  
    if ([segue.identifier isEqualToString:@"ShowDetail"])  
    {  
        DetailViewController *controller =  
            segue.destinationViewController;  
  
        controller.completionBlock = ^ (BOOL success)  
        {  
            if (success)  
            {  
                NSMutableArray *valuesArray = [[valuesDictionary  
                    objectForKey:controller.sectionName]  
                    mutableCopy];  
  
                [valuesArray  
                    replaceObjectAtIndex:controller.indexInSection  
                    withObject:controller.value];  
  
                [valuesDictionary setObject:valuesArray  
                    forKey:controller.sectionName];  
  
                // This will cause the table of values to be  
                // resorted if necessary.  
                sortedNames = nil;  
            }  
        }  
    }  
}
```

```
[self updateTableContents];
}

[self dismissModalViewControllerAnimated:YES];
};

UITableViewCell *cell = sender;
NSIndexPath *indexPath = . . .;
```

What you did here is move the contents from `detailViewControllerDidCancel:` and `detailViewControllerDidClose:` into the `prepareForSegue:`. As a result, you can remove these two methods, as they are no longer used.

And that's it. Now the code to present the detail screen and process the results is all in the same place, making it much easier to see what's going on.

Another small improvement you can make here is to replace the call to `dismissModalViewControllerAnimated:` (in `prepareForSegue:`) with:

```
[self dismissViewControllerAnimated:YES completion:nil];
```

The former was deprecated with iOS 5 already, but I still see a lot of people using it. Now that we have iOS 6, it's time to switch to the new method.

Note: It doesn't always make sense to use blocks instead of delegates. If a delegate has many different methods, then it's probably better to keep it that way, rather than requiring the client to pass in many different blocks. Replacing a delegate protocol with blocks works best when there are only one or two delegate methods.

Blocks for enumerating

Beyond being substitutes for delegates, blocks can be used in other ways. They are also handy for enumerating through collections – e.g., dictionaries and arrays. Currently the app has a few places where it loops through a dictionary or array using the `for`-statement, such as in `sortByValue` in **MasterViewController.m**:

```
- (void)sortByValue
{
    // First put all the values into one big array.
    NSMutableArray *allValues = [NSMutableArray
        arrayWithCapacity:50];
    for (NSString *key in valuesDictionary)
```

```

{
    NSArray *array = [valuesDictionary objectForKey:key];
    [allValues addObjectsFromArray:array];
}

...

```

You can simplify this using `NSDictionary`'s `enumerateKeysAndObjectsUsingBlock:` method:

```

// First put all the values into one big array.
NSMutableArray *allValues = [NSMutableArray
                             arrayWithCapacity:50];

[valuesDictionary enumerateKeysAndObjectsUsingBlock:
 ^(id key, id obj, BOOL *stop)
 {
     NSArray *array = obj;
     [allValues addObjectsFromArray:array];
 } ];

```

The dictionary loops through all its key-value pairs and calls the block for each one. The block receives three parameters: the key, the value (in the variable `obj`), and an output parameter `stop` that you'll encounter a bit later.

The above code adds the value object to the `allValues` array. The difference with the `for`-loop is that you don't have to do `[dictionary objectForKey:]` to get the value object. However, you can make it even simpler by changing the data types of the block's parameters to the classes that you're expecting:

```

[valuesDictionary enumerateKeysAndObjectsUsingBlock:
 ^(NSString *key, NSArray *array, BOOL *stop)
 {
     [allValues addObjectsFromArray:array];
 } ];

```

This works because `id` is a pointer to any object, but you do have to be sure that your dictionary really does have strings as keys and arrays as values, or the app will crash.

If you also replace the other loops with block-based enumeration, then `sortByValue` becomes:

```

- (void)sortByValue
{
    // First put all the values into one big array.
    NSMutableArray *allValues = [NSMutableArray

```

```
                                arrayWithCapacity:50];
[valuesDictionary enumerateKeysAndObjectsUsingBlock:
 ^(NSString *key, NSArray *array, BOOL *stop)
{
    [allValues addObjectFromArray:array];
}];

// Also put all the names into a big array. The order
// of the names in this array corresponds to the order
// of values in "allValues".
NSMutableArray *allNames = [NSMutableArray
                            arrayWithCapacity:50];
[namesDictionary enumerateKeysAndObjectsUsingBlock:
 ^(NSString *key, NSArray *array, BOOL *stop)
{
    [allNames addObjectFromArray:array];
}];

// Sort the array of values.
sortedValues = [allValues sortedArrayUsingSelector:
                @selector(compare:)];

// We have to put the names in the same order as the
// sorted values. For each sorted value, find its index
// in the un-sorted allValues array, then use that
// index to find the name from allNames.
NSMutableArray *theSortedNames = [NSMutableArray
                                  arrayWithCapacity:[sortedValues count]];

[sortedValues enumerateObjectsUsingBlock:
 ^(id obj, NSUInteger idx, BOOL *stop)
{
    NSUInteger index = [allValues indexOfObject:obj];
    [theSortedNames addObject:allNames[index]];
}];

sortedNames = theSortedNames;
}
```

Notice that `sortedValues`'s call to `enumerateObjectsUsingBlock:` (towards the end of the method) takes a block with different parameters. That's because `sortedValues` is an array, not a dictionary. The new block gives you both the index in the array and the object at that index. The method uses the index to look up the name from the `allNames` array in order to add it to the list of sorted names.

There is another place in the code that could benefit from block-based enumeration, and that is in `prepareForSegue:`. Currently it does the following at the bottom:

```
for (NSString *sectionName in namesDictionary)
{
    NSArray *array = [namesDictionary objectForKey:sectionName];
    NSInteger index = [array indexOfObject:name];
    if (index != NSNotFound)
    {
        controller.sectionName = sectionName;
        controller.indexInSection = index;
        break;
    }
}
```

This code loops through the `namesDictionary` in order to find the section and index that contain the string from the `name` object. As soon as the index is found, it exits the loop because there's no point in searching further.

Replace that block of code with the following:

```
[namesDictionary enumerateKeysAndObjectsUsingBlock:
 ^(NSString *sectionName, NSArray *array, BOOL *stop)
{
    NSInteger index = [array indexOfObject:name];
    if (index != NSNotFound)
    {
        controller.sectionName = sectionName;
        controller.indexInSection = index;
        *stop = YES;
    }
}];
```

This does exactly the same thing, but with a little less code. In order to exit the loop prematurely, you set the value of the `stop` parameter to `YES`. Because this is not a regular `BOOL` but a pointer to a `BOOL`, you have to use the notation `*stop` to change the actual value (`stop` is what's known as an *output parameter* since it's a parameter that's used to return a value to the caller).

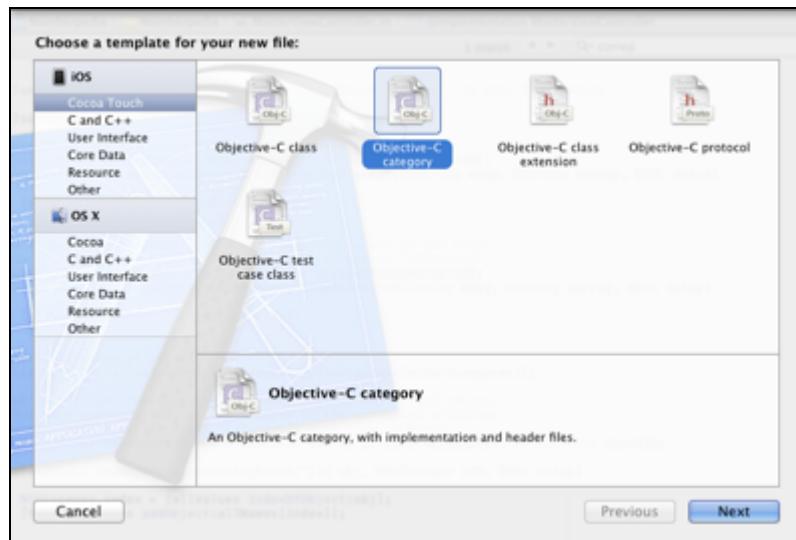
Categories: an oldie but goodie

Hardly a new feature of Objective-C, *categories* have been around for a while. They are great for extending existing classes with new functionality. Even though they are almost as old as the language itself, I mention them in this chapter because in my opinion, they aren't being used enough.

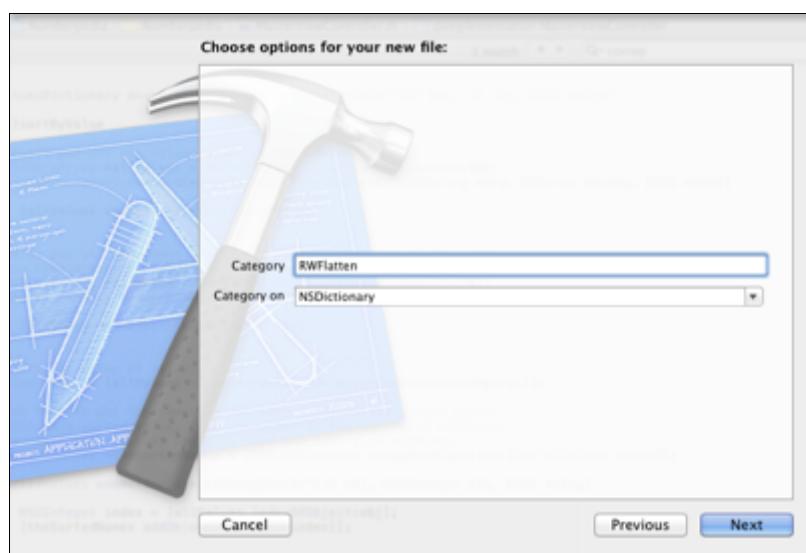
Categories are great! They allow you to extend classes to provide new functionality that can make your code easier to read and write.

For example, there is a common bit of functionality in `sortByValue` that you can extract into a category in order to make the code more readable. The first two sections of `sortByValue` do exactly the same thing, but on different dictionaries: they loop through a dictionary of arrays and put the values from these arrays into a new, “flattened” array. It’s only a few lines of code, but you might as well put them into a category anyway – no one said a category has to be big.

Go to the File menu and choose New\File... From the dialog, choose “Objective-C category”:



Press Next. In the next screen that appears, choose **RWFlatten** for the name of the category and **NSDictionary** for the class to make the category on:



Click **Next** again and choose a location for saving the new source files – usually, your project folder. This will add two new files to your project, `NSDictionary+RWFlatten.h` and `NSDictionary+RWFlatten.m`.

Replace the contents of **NSDictionary+RWFlatten.h** with:

```
@interface NSDictionary (RWFlatten)

- (NSArray *)rw_flattenIntoArray;

@end
```

And **NSDictionary+RWFlatten.m** with:

```
#import "NSDictionary+RWFlatten.h"

@implementation NSDictionary (RWFlatten)

- (NSArray *)rw_flattenIntoArray
{
    NSMutableArray *allValues = [NSMutableArray
        arrayWithCapacity:[self count]];

    [self enumerateKeysAndObjectsUsingBlock:
        ^(NSString *key, NSArray *array, BOOL *stop)
    {
        [allValues addObjectFromArray:array];
    }];
}

return allValues;
}

@end
```

You've done exactly the same thing you did before in `sortByValue`, except here you've created a new method in `NSDictionary` that you can use with any `NSDictionary` instance (as long as you include the category you created in the same project, of course).

Note: You may wonder why you added the `rw` prefix to the name of the category and `rw_` to the name of the method. This is to compensate for Objective-C's lack of namespaces (maybe in iOS 7?).

It is not inconceivable that your category method could conflict with a private method, with a method that might be added to this class in the future, or even with someone else's category, if you're using third-party libraries in your app.

To avoid such naming conflicts, it's a wise precaution to prefix your category and method names with a unique two- or three-character symbol, such as your initials. It's not exactly pretty, but better safe than sorry...

With this category in place, you can simplify `sortByValue` even further. First, import the new category in **MasterViewController.m**:

```
#import "NSDictionary+RWFlatten.h"
```

Then change `sortByValue` to:

```
- (void)sortByValue
{
    NSArray *allValues = [valuesDictionary rw_flattenIntoArray];
    NSArray *allNames = [namesDictionary rw_flattenIntoArray];

    // Sort the array of values.
    sortedValues = [allValues sortedArrayUsingSelector:
                    @selector(compare)];

    // We have to put the names in the same order as the
    // sorted values. For each sorted value, find its index
    // in the un-sorted allValues array, then use that
    // index to find the name from allNames.
    NSMutableArray *theSortedNames = [NSMutableArray
        arrayWithCapacity:[sortedValues count]];

    [sortedValues enumerateObjectsUsingBlock:
        ^(id obj, NSUInteger idx, BOOL *stop)
    {
        NSUInteger index = [allValues indexOfObject:obj];
        [theSortedNames addObject:allNames[index]];
    }];
}

sortedNames = theSortedNames;
}
```

I like adding these small categories to my projects. You could have added the "flatten into array" method to the view controller itself, but it just makes more sense to add it as extended functionality to `NSDictionary`, since you can re-use the same category in another project where you might require the same functionality.

If you give the new method a good name, it really helps to improve the readability of the code.

New Objective-C literals

It's time for some more new language features!

You no doubt know the difference between this bit of code:

```
"This is a string."
```

And this one:

```
@"This is a string."
```

The first is a C string literal and the second is an `NSString` literal. The C string is just an array of characters, but the `NSString` value is an actual object, permitting you to send messages to it, such as:

```
[@"This is a string." lowercaseString];
```

Historically, Objective-C hasn't had any support for defining literals for other objects, such as `NSNumber`... until now, that is! As of Xcode 4.5, you can write the following:

```
NSNumber *number = @1234;
```

If there is a number behind the `@` sign, the compiler converts it into an `NSNumber` object. This also works for numbers with a decimal point, character literals, and even `@YES` and `@NO`.

Since Numberpedia uses quite a few `NSNumber` objects, you can use this new notation to simplify the code quite a bit. **MasterViewController.m's** `init` method currently does this:

```
NSArray *physicsValues = [NSArray arrayWithObjects:
    [NSNumber numberWithDouble:6.02214129e23],
    [NSNumber numberWithDouble:1.3806503e-23],
    [NSNumber numberWithDouble:6.626068e-34],
    [NSNumber numberWithDouble:1.097373e-7],
    nil];
```

Replace this with the new literal notation:

```
NSArray *physicsValues = [NSArray arrayWithObjects:
    @6.02214129e23,
```

```
@1.3806503e-23,  
@6.626068e-34,  
@1.097373e-7,  
nil];
```

That is a lot easier to read! Also modify the `mathematicsValues` and `funValues` arrays:

```
NSArray *mathematicsValues = [NSArray arrayWithObjects:  
    @2.71828183,  
    @3.14159265,  
    @1.414213562,  
    @6.2831853,  
    nil];  
  
NSArray *funValues = [NSArray arrayWithObjects:  
    @-273.15,  
    @90210,  
    @1.618,  
    @214,  
    @13,  
    nil];
```

Note that some of these numbers are integers, while others are floats. The compiler automatically calls `[NSNumber numberWithInt:]` or `numberWithFloat:` as necessary – that is no longer something you need to worry about!

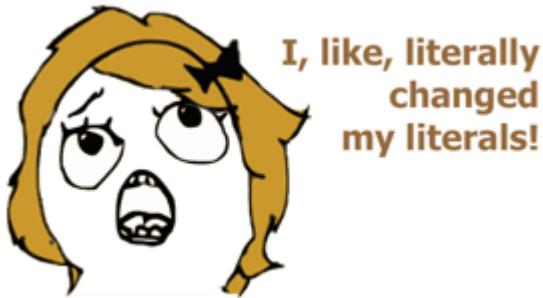
There is also a line in **DetailViewController.m** that you can simplify. The `done:` method currently does this:

```
self.value = [NSNumber numberWithInt:0];
```

That can simply become:

```
self.value = @0;
```

Having the ability to specify literal `NSNumber` objects a bit more succinctly is nice, but you're not done yet. There are also new shorter syntaxes for `NSArray` and `NSDictionary` objects!



Switch back to **MasterViewController.m** and replace the code that does `[NSArray arrayWithObjects:]` with:

```
NSArray *physicsNames = @[
    @"Avogadro",
    @"Boltzman",
    @"Planck",
    @"Rydberg"];
```

```
NSArray *physicsValues = @[
    @6.02214129e23,
    @1.3806503e-23,
    @6.626068e-34,
    @1.097373e-7];
```

```
NSArray *mathematicsNames = @[
    @"e",
    @"Pi ( $\pi$ )",
    @"Pythagoras' constant",
    @"Tau ( $\tau$ )"];
```

```
NSArray *mathematicsValues = @[
    @2.71828183,
    @3.14159265,
    @1.414213562,
    @6.2831853];
```

```
NSArray *funNames = @[
    @"Absolute Zero",
    @"Beverly Hills",
    @"Golden Ratio",
    @"Number of Human Bones",
    @"Unlucky Number"];
```

```
NSArray *funValues = @[
    @-273.15,
```

```
@90210,  
@1.618,  
@214,  
@13];
```

Anything in between @[and] brackets becomes an item in an NSArray. Also notice that it is no longer necessary to add the terminating nil sentinel in the array declaration. The nil sentinel was necessary with [NSArray arrayWithObjects:] in order to let NSArray know what the end of the list was, but this new syntax does away with it.

Dictionaries have a similar syntax, although now the brackets are @{@" ... }.

Where you used to do this:

```
namesDictionary = [NSDictionary dictionaryWithObjectsAndKeys:  
    physicsNames, @"Physics Constants",  
    mathematicsNames, @"Mathematics",  
    funNames, @"Fun Numbers",  
    nil];
```

You can now simply do this:

```
namesDictionary = @{@"  
    @"Physics Constants" : physicsNames,  
    @"Mathematics" : mathematicsNames,  
    @"Fun Numbers" : funNames  
};
```

The @{@" indicates to the compiler that this starts a new NSDictionary. Inside the brackets, each key-value pair is specified as key : value. Notice that this is the other way around from dictionaryWithObjectsAndKeys:, where the order was value, key, value, key, and so on. Again, no nil sentinel is required at the end of the list.

It is important to realize that @[] and @{@" } create immutable arrays and dictionaries, respectively. To make a mutable dictionary, which is what the valuesDictionary needs to be, you have to call mutableCopy first:

```
valuesDictionary = [@{  
    @"Physics Constants" : physicsValues,  
    @"Mathematics" : mathematicsValues,  
    @"Fun Numbers" : funValues  
} mutableCopy];
```

Calling mutableCopy turns the NSDictionary into an NSMutableDictionary. Note that it's okay to mix these array and dictionary literals in a single statement. For example, this works just fine:

```
valuesDictionary = [@{  
    @"Physics Constants" : physicsValues,  
    @"Mathematics" : mathematicsValues,  
    @"Fun Numbers" : @[@-273.15, @90210, @1.618, @214, @13]  
} mutableCopy];
```

(You're also not limited to putting literals inside these arrays and dictionaries – any object will do. In fact, the code above is putting real `NSArray` objects into the dictionary.)

There are some limitations to this new notation, though. For `NSStrings`, you can do this to create a static (global) object:

```
static NSString *myString = @"A static string";
```

That notation unfortunately doesn't work for the new `NSNumber`, `NSArray` and `NSDictionary` literals. You'll get a compiler error if you try.

Note: The new literal syntax can also “box” the values of expressions. The term *boxing* means taking a primitive value, such as an `int` or `float`, and converting it into an object. For example:

```
int x = 123;  
NSNumber *y = @(x + 321); // boxes the result of the calculation into an  
NSNumber
```

First the expression between the parentheses is evaluated and then the result is placed into a new `NSNumber` object.

The cool thing is that this new syntax is backwards-compatible with previous versions of iOS, so you can also use it in apps that need to work on iOS 5 and older. For detailed info on the new literals, check out the official compiler documentation at <http://clang.llvm.org/docs/ObjectiveCLiterals.html>.

Grabbing the goods with subscripting

Creating arrays and dictionaries has become a bit easier, and even better than that... so has using them! In other languages, you can often do the following to get or set an item from an array:

```
array[index] = . . .;
```

But Objective-C has always required you to use verbose methods such as:

```
[array objectAtIndex:...]
```

Happy days have arrived, for the creators of Objective-C have finally brought us the goodies. From now on, you can simply use [] brackets to index arrays, dictionaries, and even your own classes!



MasterViewController.m calls `objectAtIndex:` in quite a few places, for example in `tableView:titleForHeaderInSection:`. That method currently looks like this:

```
- (NSString *)tableView:(UITableView *)tableView
                  titleForHeaderInSection:(NSInteger)section
{
    if (sortedByName)
        return [sortedSectionNames objectAtIndex:section];
    else
        return nil;
}
```

You can now simplify it to the following:

```
- (NSString *)tableView:(UITableView *)tableView
                  titleForHeaderInSection:(NSInteger)section
{
    if (sortedByName)
        return sortedSectionNames[section];
    else
        return nil;
}
```

That looks a lot more natural, especially if you have programmed in other languages before. (It's also the syntax C uses for regular C arrays.)

Another example is `tableView:numberOfRowsInSection:`:

```
- (NSInteger)tableView:(UITableView *)tableView
                  numberOfRowsInSection:(NSInteger)section
{
    if (sortedByName)
```

```

{
    NSString *sectionName = [sortedSectionNames
                             objectAtIndex:section];
    return [[namesDictionary objectForKey:sectionName]
           count];
}
else
{
    return [sortedValues count];
}
}

```

You can simplify this to:

```

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    if (sortedByName)
    {
        NSString *sectionName = sortedSectionNames[section];
        return [namesDictionary[sectionName] count];
    }
    else
    {
        return [sortedValues count];
    }
}

```

Notice that two things changed here. First the line that gets the `sectionName`, as before, but also the line that accesses the `namesDictionary`. Instead of doing `objectForKey:` on a dictionary, you can use the `[]` brackets instead. On an array the brackets mean “get the object at that index”; on a dictionary they mean, “get the object for that key”.

Now go through `MasterViewController` and change every instance of `objectAtIndex:` and `objectForKey:` to use bracket notation. It’s a dozen or so places that need to change, but that’s good practice to get the hang of this notation. The next time you find yourself typing `objectAtIndex:` or `objectForKey:`, simply use `[]` brackets instead.

You can use `[]` not only for reading from an array or dictionary, but also for inserting or replacing values. In `prepareForSegue:` the code currently does the following:

```

[valuesArray replaceObjectAtIndex:controller.indexInSection
                           withObject:controller.value];

```

```
[valuesDictionary setObject:valuesArray  
    forKey:controller.sectionName];
```

With bracket notation that becomes:

```
valuesArray[controller.indexInSection] = controller.value;  
  
valuesDictionary[controller.sectionName] = valuesArray;
```

You tell me which is easier to read. ☺

Note: If you have a lot of code that you wish to convert to this new syntax, then use the Refactor\Convert to Modern Objective-C Syntax command from Xcode's Edit menu. Just like the ARC conversion utility, this tool will go through your source files and automatically make all the changes. It converts your code to use both the new literals and subscripting for arrays and dictionaries.

Go ahead and run this now for practice – if you made all the updates listed so far, it should say "No source changes necessary." But next time you have a big project you want to convert, this should save you a lot of time!

One further point: just like the new literals, the new subscripting notation also works on older versions of iOS, so you can use this technique in apps for iOS 5 and earlier. Yay!

Refactoring the code

You have to admit, the source code is already much simpler – and much lighter, with fewer lines and characters! You were able to remove a lot of boilerplate code and clean up the public interfaces of the classes. However, it is far from perfect. The app can still be improved by putting the data model into its own class (or classes), rather than manipulating arrays and dictionaries directly.

Note: It's best practice to have your data model in its own classes rather than using `NSArray` or `NSDictionary` directly. It makes your code much easier to read and maintain over time, and lets the compiler work for you and help find typos and mistakes. If you're still not convinced, compare the project the way it is now to what it will look like after this section, and see which is easier to read and understand!

Add a new file to the project, using the Objective-C class template. Name it "Item" and make it a subclass of NSObject. Replace the contents of **Item.h** with the following:

```
@interface Item : NSObject

+ (id)itemWithName:(NSString *)name value:(NSNumber *)value;

- (id)initWithName:(NSString *)name value:(NSNumber *)value;

@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSNumber *value;

@end
```

And **Item.m** with:

```
#import "Item.h"

@implementation Item

+ (id)itemWithName:(NSString *)name value:(NSNumber *)value
{
    return [[self alloc] initWithName:name value:value];
}

- (id)initWithName:(NSString *)name value:(NSNumber *)value
{
    if ((self = [super init]))
    {
        _name = name;
        _value = value;
    }
    return self;
}

- (NSComparisonResult)compare:(Item *)otherItem
{
    return [_value compare:otherItem.value];
}

@end
```

The `Item` class combines the `NSString` for the name and the `NSNumber` for the value into one object. See how easy that was? No need to make any instance variables,

and no `@synthesize`. The header file only contains what you want other classes to see and nothing more.

In **MasterViewController.m**, first import the new class:

```
#import "Item.h"
```

Then change the `@implementation` section to:

```
@implementation MasterViewController
{
    NSDictionary *dictionary;
    NSArray *sortedSectionNames;

    BOOL sortedByName;
    NSArray *sortedItems;
}
```

Before, you needed two different dictionaries (one for names, and one for values), because you were storing the information separately. Now that you are storing the information properly, together in a single object, you only need one dictionary and array.

Change the `init` method to create and populate these new objects:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        NSArray *physics = @{
            [Item itemWithName:@"Avogadro" value:@6.02214129e23],
            [Item itemWithName:@"Boltzman" value:@1.3806503e-23],
            [Item itemWithName:@"Planck" value:@6.626068e-34],
            [Item itemWithName:@"Rydberg" value:@1.097373e-7]
        };

        NSArray *mathematics = @{
            [Item itemWithName:@"e" value:@2.71828183],
            [Item itemWithName:@"π" value:@3.14159265],
            [Item itemWithName:@"Pythagoras' constant"
                value:@1.414213562],
            [Item itemWithName:@"Tau (Π)" value:@6.2831853]
        };

        NSArray *fun = @{
            [Item itemWithName:@"Absolute Zero" value:@-273.15],
            [Item itemWithName:@"Beverly Hills" value:@90210],
        };
    }
}
```

```
[Item itemWithName:@"Golden Ratio" value:@1.618],  
[Item itemWithName:@"Number of Human Bones" value:@214],  
[Item itemWithName:@"Unlucky Number" value:@13]  
];  
  
dictionary = @{  
    @"Physics Constants" : physics,  
    @"Mathematics" : mathematics,  
    @"Fun Numbers" : fun,  
};  
  
sortedSectionNames = [[dictionary allKeys]  
    sortedArrayUsingSelector:@selector(compare:)];  
  
sortedByName = YES;  
}  
return self;  
}
```

It's already starting to look better – now it's easy to see the numbers that correspond to each name.

Next you have to go through `MasterViewController` and modify all of the code that uses the old dictionaries to the new one.

Start with the table view methods. Replace the `numberOfRowsInSection` method with:

```
- (NSInteger)tableView:(UITableView *)tableView  
    numberOfRowsInSection:(NSInteger)section  
{  
    if (sortedByName)  
    {  
        NSString *sectionName = sortedSectionNames[section];  
        return [dictionary[sectionName] count];  
    }  
    else  
    {  
        return [sortedItems count];  
    }  
}
```

That was a very straightforward change; only the names of the dictionary and array are different. The change to `cellForRowAtIndexPath` is slightly more involved:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
```

```
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"NumberCell"];

    Item *item;

    if (sortedByName)
    {
        NSString *sectionName =
            sortedSectionNames[indexPath.section];
        NSArray *itemsArray = dictionary[sectionName];
        item = itemsArray[indexPath.row];
    }
    else
    {
        item = sortedItems[indexPath.row];
    }

    cell.textLabel.text = item.name;
    cell.detailTextLabel.text = [item.value description];
    return cell;
}
```

Here you look up the correct item to display in either the dictionary or sorted array, based on whether you're sorting by name or value. Once you have the correct item, it's easy to pull out the number and description and display it in the cell.

Next up are `updateTableContents` and `sortByValue`. The first method determines whether the list of sorted items is still `nil` if the user is switching to the "Sort by Value" tab. If so, it lazily creates this list by calling `sortByValue`.

This method now becomes very simple. Because the `Item` object has a `compare:` method, it can simply put the contents of the dictionary into a new array (using the `RWFlatten` category you made earlier) and then sort this array:

```
- (void)updateTableContents
{
    // Lazily sort the list by value if we haven't
    // done that yet.
    if (!sortedByName && sortedItems == nil)
    {
        [self sortByValue];
    }

    [self.tableView reloadData];
}
```

```
- (void)sortByValue
{
    NSArray *allItems = [dictionary rw_flattenIntoArray];

    sortedItems = [allItems
        sortedArrayUsingSelector:@selector(compare:)];
}
```

This leaves two methods that do not compile yet, `prepareForSegue:` and `didReceiveMemoryWarning`. For now, comment out the `prepareForSegue:` method (you'll be rewriting it soon) and replace `didReceiveMemoryWarning` with:

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

    if ([self isViewLoaded] && self.view.window == nil)
    {
        self.view = nil;
        sortedItems = nil;
    }
}
```

Build and run the app. There should be no warnings or compiler errors. The app should still work as before, except that the segue for tapping a row is no longer handled properly.

To fix that, you first have to change `DetailViewController`. Rather than passing along the section name and index of the item to edit, you can now give it the actual `Item` object.

Replace the contents of **DetailViewController.h** with:

```
@class Item;

typedef void (^DetailViewControllerCompletionBlock)(BOOL
                                                 success);

@interface DetailViewController : UIViewController

@property (nonatomic, copy) DetailViewControllerCompletionBlock
                           completionBlock;
@property (nonatomic, strong) Item *itemToEdit;

@end
```

Except for `completionBlock`, all the properties have been replaced with the `itemToEdit` property.

There are also a few changes you need to make to **DetailViewController.m**. First, import the Item class:

```
#import "Item.h"
```

Then, change `viewDidLoad` to:

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

    self.navigationBar.topItem.title = self.itemToEdit.name;
    self.textField.text = [self.itemToEdit.value description];

    [self.textField becomeFirstResponder];
}
```

And change the `done:` action method to:

```
- (IBAction)done:(id)sender
{
    NSNumberFormatter *formatter = [[NSNumberFormatter alloc]
                                   init];
    [formatter setNumberStyle:NSNumberFormatterDecimalStyle];
    NSNumber *newValue = [formatter
                          numberFromString:self.textField.text];

    self.itemToEdit.value = (newValue != nil) ? newValue : @0;

    if (self.completionBlock != nil)
        self.completionBlock(YES);
}
```

Only the line that places the value into `self.itemToEdit` has changed. That's all for the Detail View Controller.

Now you can fix `prepareForSegue:` in **MasterViewController.m** to put the Item object from the row that the user tapped into the `itemToEdit` property. The new `prepareForSegue:` becomes:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                  sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"ShowDetail"])
}
```

```
{  
    DetailViewController *controller =  
        segue.destinationViewController;  
  
    controller.completionBlock = ^(BOOL success)  
    {  
        if (success)  
        {  
            // This will cause the table of values to be  
            // resorted if necessary.  
            sortedItems = nil;  
  
            [self updateTableContents];  
        }  
        [self dismissViewControllerAnimated:YES  
            completion:nil];  
    };  
  
    UITableViewCell *cell = sender;  
    NSIndexPath *indexPath = [self.tableView  
        indexPathForCell:cell];  
  
    if (sortedByName)  
    {  
        NSString *sectionName =  
            sortedSectionNames[indexPath.section];  
        NSArray *itemsArray = dictionary[sectionName];  
        controller.itemToEdit = itemsArray[indexPath.row];  
    }  
    else  
    {  
        controller.itemToEdit = sortedItems[indexPath.row];  
    }  
}
```

That's a lot simpler than it was before. There is no need to update the dictionary because the Detail View Controller has already put the new `NSNumber` into the Item object.

At this point, you do have to clear out the list of sorted items, because changing a number might have an impact on the sort order. That's why you set `sortedItems` to `nil` and call `updateTableContents` to reload the table, re-sorting the list when necessary.

Build and run, and verify that the detail screen works again. All is well!

Public read-only, private read-write properties

Class extensions allow you to have public properties that are read-only, so that users of the class can't change the thing that the property points to, but that are writeable to the class itself. That is often a useful feature to have.

You'll now add a new class to the project, `Section`, that you'll use to organize the list into sections. Currently the dictionary contains one `NSArray` for each section, but from now on it will have `Section` objects instead.

Add a new file to the project using the Objective-C class template. Name it `Section`, and make it a subclass of `NSObject`. Replace **Section.h** with the following:

```
@interface Section : NSObject

@property (nonatomic, strong, readonly) NSArray *items;

- (id)initWithArray:(NSArray *)array;

@end
```

Notice that the `items` property is marked `readonly`. Users of this class cannot assign a new object to the `items` pointer, only use the existing object.

Put the following into **Section.m**:

```
#import "Section.h"

@interface Section ()
@property (nonatomic, strong, readwrite) NSMutableArray *items;
@end

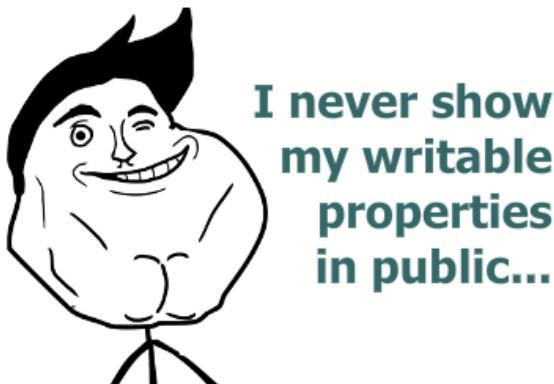
@implementation Section

- (id)initWithArray:(NSArray *)array
{
    if ((self = [super init]))
    {
        self.items = [array mutableCopy];
    }
    return self;
}

@end
```

The class extension re-declares the same `items` property, but this time marks it as `readwrite`. That's one of the special powers that class extensions have that regular categories don't – you can override property declarations internally.

In this example, inside the class you can write `self.items = object;` to change the object that `items` points to. That doesn't work outside the class because to outsiders, `items` is considered read-only.



Note: There is some controversy over whether it's a good idea to access properties in your `init` and `dealloc` methods. Apple officially recommends against it, so if you follow their guidelines, you would put the following in `init` instead:

```
_items = [array mutableCopy];
```

Rather than going through the setter for the property, you now assign the new object to the `_items` instance variable directly.

The reason for not using properties here is that setters and getters can cause side effects, and that's generally undesirable while your object is still being constructed (in the `init` method) or being destroyed (in `dealloc`). For a simple property such as this, I don't think it's a big deal. But you have been duly warned. ☺

In **MasterViewController.m**, import the new class:

```
#import "Section.h"
```

In the `init` method, change the creation of the `NSArray` to make `Section` objects instead:

```
Section *physics = [[Section alloc] initWithArray:@[
    [Item itemName:@"Avogadro" value:@6.02214129e23],
```

```
[Item itemWithName:@"Boltzman" value:@1.3806503e-23],
[Item itemWithName:@"Planck" value:@6.626068e-34],
[Item itemWithName:@"Rydberg" value:@1.097373e-7]
]];

Section *mathematics = [[Section alloc] initWithArray:@[
[Item itemWithName:@"e" value:@2.71828183],
[Item itemWithName:@"π" value:@3.14159265],
[Item itemWithName:@"Pythagoras' constant"
value:@1.414213562],
[Item itemWithName:@"Tau (τ)" value:@6.2831853]
]];

Section *fun = [[Section alloc] initWithArray:@[
[Item itemWithName:@"Absolute Zero" value:@-273.15],
[Item itemWithName:@"Beverly Hills" value:@90210],
[Item itemWithName:@"Golden Ratio" value:@1.618],
[Item itemWithName:@"Number of Human Bones" value:@214],
[Item itemWithName:@"Unlucky Number" value:@13]
]];
```

The `numberOfRowsInSection` method now becomes:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    if (sortedByName)
    {
        NSString *sectionName = sortedSectionNames[section];
        Section *section = dictionary[sectionName];
        return [section.items count];
    }
    else
    {
        return [sortedItems count];
    }
}
```

Because the dictionary now contains `Section` objects instead of `NSArrays`, it first has to get the `Section` object and then ask its `items` property – which *is* an array – for its element count.

Something similar needs to happen in `cellForRowAtIndexPath`:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

```

{
    . . .

    if (sortedByName)
    {
        NSString *sectionName =
            sortedSectionNames[indexPath.section];

        Section *section = dictionary[sectionName];
        item = section.items[indexPath.row];
    }
    else
    {
        . . .
    }
}

```

And in `prepareForSegue:`:

```

- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender
{
    . . .

    if (sortedByName)
    {
        NSString *sectionName =
            sortedSectionNames[indexPath.section];

        Section *section = dictionary[sectionName];
        controller.itemToEdit = section.items[indexPath.row];
    }
    else
    {
        . . .
    }
}

```

Run the app to see if it still works. Unfortunately... it doesn't. When you switch to the Sort by Value tab, the app crashes with a SIGABRT and the message "array argument is not an NSArray." That makes sense, because the RWFlatten category that `sortByValue` uses assumes that the contents of the dictionary are NSArray objects, but now they are Section objects.

Replace `sortByValue` with:

```

- (void)sortByValue
{
    NSMutableArray *allItems = [NSMutableArray
        arrayWithCapacity:50];
}

```

```
[dictionary enumerateKeysAndObjectsUsingBlock:  
    ^(NSString *key, Section *section, BOOL *stop)  
    {  
        [allItems addObjectsFromArray:section.items];  
    }];  
  
sortedItems = [allItems  
    sortedArrayUsingSelector:@selector(compare:)];  
}
```

That's better. If you run the app now, it should work as before.

Now why did you go through all this trouble to make `Section` objects? Well, first of all it's often a good idea to create classes that fit your data model better than plain `NSArray` and `NSDictionary` objects, as mentioned earlier.

But secondly, this allows me to demonstrate another useful feature that was added to Objective-C: adding subscripting into your own classes.

Subscripting in your own classes

You have seen that `NSArray` and `NSDictionary` can now work with `[]` notation to get and set elements. But you can also add this functionality to your own classes simply by implementing one or two methods with certain special names.

Let's try it with the `Section` class. Add the following method signature to **Section.h**:

```
- (id)objectAtIndexedSubscript:(NSUInteger)idx;
```

That's the method for doing array-style or *indexed* subscripting, i.e. where you put a number between the `[]` brackets. Add the method body to **Section.m**:

```
- (id)objectAtIndexedSubscript:(NSUInteger)idx  
{  
    return self.items[idx];  
}
```

This is only a simple example app, so you'll just use it as shorthand for accessing the `items` array.

In **MasterViewController.m**, you can now change the lines where it does this:

```
item = section.items[indexPath.row];
```

Into this:

```
item = section[indexPath.row];
```

Cool, huh?



In `cellForRowAtIndexPath` you can make the same change, so that it reads like this:

```
NSString *sectionName = sortedSectionNames[indexPath.section];
Section *section = dictionary[sectionName];
item = section[indexPath.row];
```

You could actually shorten this to:

```
NSString *sectionName = sortedSectionNames[indexPath.section];
item = dictionary[sectionName][indexPath.row];
```

Or even:

```
item = dictionary[sortedSectionNames[indexPath.section]]
[indexPath.row];
```

Although that may be going a bit overboard... It gets hard to tell which thing you're accessing where.

Let's add another data model class to the project, named simply `DataModel`. This will be the top-level model object and contain the main dictionary, so that can move out of the view controller.

Add a new file to the project using the Objective-C class template, named `DataModel`, and make it a subclass of `NSObject`. Put the following in **DataModel.h**:

```
@interface DataModel : NSObject

@property (nonatomic, strong, readonly) NSArray
    *sortedSectionNames;
@property (nonatomic, strong, readonly) NSArray *sortedItems;

- (void)sortByValue;
- (void)clearSortedItems;

- (id)objectForKeyedSubscript:(id)key;
```

```
@end
```

You should recognize most of this as coming from the `MasterViewController`. Put the following in `DataModel.m`:

```
#import "DataModel.h"
#import "Item.h"
#import "Section.h"

@interface DataModel ()
@property (nonatomic, strong, readwrite) NSArray
    *sortedSectionNames;
@property (nonatomic, strong, readwrite) NSArray *sortedItems;
@end

@implementation DataModel
{
    NSDictionary *dictionary;
}

- (id)init
{
    if ((self = [super init]))
    {
        Section *physics = [[Section alloc] initWithArray:@[
            [Item itemWithName:@"Avogadro" value:@6.02214129e23],
            [Item itemWithName:@"Boltzman" value:@1.3806503e-23],
            [Item itemWithName:@"Planck" value:@6.626068e-34],
            [Item itemWithName:@"Rydberg" value:@1.097373e-7]
        ]];

        Section *mathematics = [[Section alloc] initWithArray:@[
            [Item itemWithName:@"e" value:@2.71828183],
            [Item itemWithName:@"π" value:@3.14159265],
            [Item itemWithName:@"Pythagoras' constant"
                value:@1.414213562],
            [Item itemWithName:@"Tau (τ)" value:@6.2831853]
        ]];

        Section *fun = [[Section alloc] initWithArray:@[
            [Item itemWithName:@"Absolute Zero" value:@-273.15],
            [Item itemWithName:@"Beverly Hills" value:@90210],
            [Item itemWithName:@"Golden Ratio" value:@1.618],
            [Item itemWithName:@"Number of Human Bones" value:@214],
            [Item itemWithName:@"Unlucky Number" value:@13]
        ]];
    }
}
```

```
];
dictionary = @{
    @"Physics Constants" : physics,
    @"Mathematics" : mathematics,
    @"Fun Numbers" : fun,
};

self.sortedSectionNames = [[dictionary allKeys]
    sortedArrayUsingSelector:@selector(compare:)];
}

return self;
}

- (void)sortByValue
{
    NSMutableArray *allItems = [NSMutableArray
        arrayWithCapacity:50];

    [dictionary enumerateKeysAndObjectsUsingBlock:
        ^(NSString *key, Section *section, BOOL *stop)
    {
        [allItems addObjectsFromArray:section.items];
    }];
}

self.sortedItems = [allItems
    sortedArrayUsingSelector:@selector(compare:)];
}

- (void)clearSortedItems
{
    self.sortedItems = nil;
}

- (id)objectForKeyedSubscript:(id)key
{
    return dictionary[key];
}

@end
```

Again, the class extension re-declares the read-only properties as read-write. The rest of the code comes straight from `MasterViewController`. There is a new method named `clearSortedItems` that sets the `sortedItems` pointer to `nil`. It's necessary to do that in a method because the Master View Controller can no longer change `sortedItems` directly (it's now a read-only property).

The `objectForKeyedSubscript:` method is also new. This is a special method that, just like `objectAtIndexedSubscript:`, allows other objects to use the `[]` brackets with instances of `DataModel`. However, this time it is not an indexed subscript but a **keyed subscript**, meaning that an object, not a number, is used as the subscript – just as with an `NSDictionary`. In this case, it will be the section name, which is an `NSString`.

That's it for the `DataModel` class. What's left is changing `MasterViewController` to use it. That is mostly a matter of removing redundant code. Begin by importing `DataModel` into **MasterViewController.m**:

```
#import "DataModel.h"
```

Then change the instance variables to:

```
@implementation MasterViewController
{
    DataModel *dataModel;
    BOOL sortedByName;
}
```

And replace the init method with:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        dataModel = [[DataModel alloc] init];
        sortedByName = YES;
    }
    return self;
}
```

In both `didReceiveMemoryWarning` and `prepareForSegue:`, instead of setting `sortedItems` to `nil`, do:

```
[dataModel clearSortedItems];
```

There is another change you need to make in `prepareForSegue:`, because the `sortedSectionNames`, `dictionary`, and `sortedItems` instance variables no longer exist. Replace that part of the method with:

```
if (sortedByName)
{
    NSString *sectionName =
        dataModel.sortedSectionNames[indexPath.section];
```

```
    Section *section = dataModel[sectionName];
    controller.itemToEdit = section[indexPath.row];
}
else
{
    controller.itemToEdit = dataModel.sortedItems[indexPath.row];
}
```

You can now get the Section object simply by asking the `DataModel` with the notation `dataModel[sectionName]`. Pretty sweet.

There's more to be done before everything compiles without errors. Change the if-statement in `updateTableContents` to:

```
if (!sortedByName && dataModel.sortedItems == nil)
{
    [dataModel sortByValue];
}
```

Remove the `sortByValue` method.

Replace `numberOfSectionsInTableView:` and `titleForHeaderInSection:` with:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    if (sortedByName)
        return [dataModel.sortedSectionNames count];
    else
        return 1;
}

- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    if (sortedByName)
        return dataModel.sortedSectionNames[section];
    else
        return nil;
}
```

Change the `numberOfRowsInSection` method to:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
```

```

if (sortedByName)
{
    NSString *sectionName =
        dataModel.sortedSectionNames[section];

    Section *section = dataModel[sectionName];
    return [section.items count];
}
else
{
    return [dataModel.sortedItems count];
}
}

```

And finally, the if-statement in `cellForRowAtIndexPath`:

```

if (sortedByName)
{
    NSString *sectionName =
        dataModel.sortedSectionNames[indexPath.section];

    Section *section = dataModel[sectionName];
    item = section[indexPath.row];
}
else
{
    item = dataModel.sortedItems[indexPath.row];
}

```

Now the code goes through the `DataModel` object wherever before it used instance variables directly, keeping all logic related to the data model nicely centralized and reusable. You should be able to build and run again.

The `DataModel` class uses `[]` subscripts for returning a `Section` object with a particular name (keyed subscripting) and the `Section` class uses `[]` subscripts for returning the `Item` object at a specific index (indexed subscripting). However, there is no reason you can't make `DataModel` use indexed subscripting as well.

A class can do both keyed and indexed subscripting at the same time, because these features are provided by two different methods, `objectForKeyedSubscript:` and `objectAtIndexedSubscript:`, respectively. The compiler looks at the data type of the thing between the `[]` brackets to determine which of these two methods to call.

Add the following method signature to `DataModel.h`:

```
- (id)objectAtIndexedSubscript:(NSUInteger)idx;
```

And its implementation to **DataModel.m**:

```
- (id)objectAtIndexedSubscript:(NSUInteger)idx
{
    return self.sortedSectionNames[idx];
}
```

Now `dataModel[number]` returns the name of the section at that index, and `dataModel[string]` runs the `Section` object with that name.

So everywhere in `MasterViewController` where it does:

```
NSString *sectionName = dataModel.sortedSectionNames[number];
```

You can replace that with simply:

```
NSString *sectionName = dataModel[number];
```

For example, the if-statement in `cellForRowAtIndexPath` becomes:

```
if (sortedByName)
{
    NSString *sectionName = dataModel[indexPath.section];
    Section *section = dataModel[sectionName];
    item = section[indexPath.row];
}
else
{
    . . .
}
```

I'm not advocating using subscripts everywhere, always. If indexed or keyed subscripting is a good fit for your (data model) classes, then by all means use it. As you've seen, it can make the code a lot simpler to read.

However, if you overdo it, the code can actually become a lot less transparent and harder to understand. For example, `DataModel` contains two different arrays: the list of section names, but also the list of sorted items. It may not be immediately obvious to a user of the `DataModel` class which of these two lists gets indexed with `[]`.

With great power comes great responsibility – so use with care! ☺

Tip: In this chapter you have only used `[]` on `DataModel` and `Section` to retrieve data, but you can also use `[]` notation to change data in your own classes. To pull that off, you need to implement one or more of the following methods:

```
- (void)setObject:(id)obj atIndex:(NSUInteger)idx;  
- (void)setObject:(id)obj forKeyedSubscript:(id <NSCopying>)key;
```

For more information, refer to the official compiler documentation at
<http://clang.llvm.org/docs/ObjectiveCLiterals.html>

There is another document at the LLVM website that lists all the language extensions provided by the Clang compiler, including those for C and C++. See here: <http://clang.llvm.org/docs/LanguageExtensions.html>

If you're curious about which Objective-C features work with which versions of iOS or Mac OS X, then look up the document "Objective-C Feature Availability Index" on the Apple Developer Portal.

Where to go from here?

Congratulations, you have fully refactored this project to use the latest and greatest Objective-C techniques and coding style!

In the process, you have reduced the lines of code from 366 to 342 (7% savings) and the total characters from 11,974 to 9,739 (19% savings). And more importantly – you have made your code much easier to read, understand, and maintain.

Note: Curious about how we got these stats?

We used the open source Count Lines of Code (cloc) Perl script from <http://cloc.sourceforge.net/> for lines of code, and the Unix word count tool (wc) for characters.

Handy tools to have in your arsenal!

Here's a summary of what you've learned in this chapter. If you took the challenge at the beginning of the chapter, check to see how many you caught. If you caught less than 5 you're an Objective-C Padawan, if you caught 5-9 you're an Objective-C Jedi Knight, and if you caught 10 or more you're an Objective-C Jedi Master!

1. Use ARC to stop worrying about memory management.
2. You no longer need to declare instance variables for your properties.
3. You no longer need to `@synthesize` your properties. Auto-synthesize will automatically make the backing instance variable and prefix it with an underscore.

4. It's better to place your instance variables in the `@implementation` section than in the public interface.
5. You no longer need forward declarations for your private methods and `IBAction` methods.
6. Rather than making a "(Private)" category, use a class extension.
7. Put your `IBOutlet` properties in the class extension.
8. You can list the protocols that your class conforms to in the class extension, rather than in the public interface.
9. The class extension can re-declare a `readonly` property to be a `readwrite` property inside the .m file.
10. You don't need to `#import` UIKit and other system frameworks if you already import them in the Prefix.pch file.
11. Use categories to add new functionality to existing classes, even those from UIKit and Foundation.
12. Use blocks for enumerating through arrays and dictionaries, and to replace delegates where that makes sense.
13. You can now use the `@` symbol for number literals, `@[]` for arrays and `@{ }` for dictionaries.
14. You can use `[]` notation to subscript arrays and dictionaries, and even implement this functionality in your own classes.

Enjoy simplifying your code!

Chapter 3: Beginning Auto Layout

By Matthijs Hollemans

Have you ever been frustrated trying to make your apps look good in both portrait and landscape orientation? Is making screen layouts that support both the iPhone and iPad driving you to the brink of madness? Despair no longer, I bring you good news!

It's not hard to design a user interface for a screen that is always guaranteed to be the same size, but if the screen's frame can change, the positions and sizes of your UI elements also have to adapt to fit into these new dimensions.

Until now, if your designs were reasonably complex, you had to write a lot of code to support such adaptive layouts. You will be glad to hear that this is no longer the case – iOS 6 brings an awesome new feature to the iPhone and iPad: Auto Layout.

Not only does Auto Layout makes it easy to support different screen sizes in your apps, as a bonus it also makes internationalization almost trivial. You no longer have to make new nibs or storyboards for every language that you wish to support, and this includes right-to-left languages such as Hebrew or Arabic.

This chapter shows you how to get started with Auto Layout using Interface Builder. The next chapter builds on this knowledge and shows you how to unleash the full power of Auto Layout via code.

Note that this is a long chapter – the longest chapter in the book in fact. This chapter is long because Auto Layout is a big topic and there are a lot of subtle aspects to cover. But if you only have a little time and want to get a feel for just the basics, I will point out two natural breaking points in the chapter that would be good spots to take a break or move onto other topics.

So grab a snack and your favorite caffeinated beverage, and get ready to become an Auto Layout master!

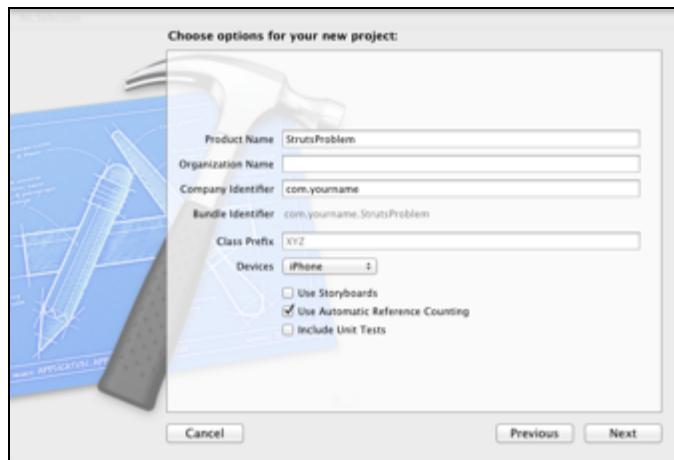
The problem with springs and struts

You are no doubt familiar with *autosizing masks* – also known as the “springs and struts” model. The autosizing mask determines what happens to a view when its superview changes size. Does it have flexible or fixed margins (the struts), and what happens to its width and height (the springs)?

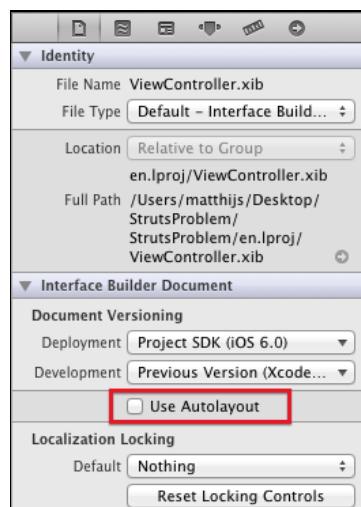
For example, with a flexible width the view will become proportionally wider if the superview also becomes wider. And with a fixed right margin, the view’s right edge will always stick to the superview’s right edge.

The autosizing system works well for simple cases, but it quickly breaks down when your layouts become more intricate. Let’s look at an example where springs and struts simply don’t cut it.

Open Xcode and create a new project based on the Single View Application template. Call the app “StrutsProblem”, choose iPhone and disable Storyboards:



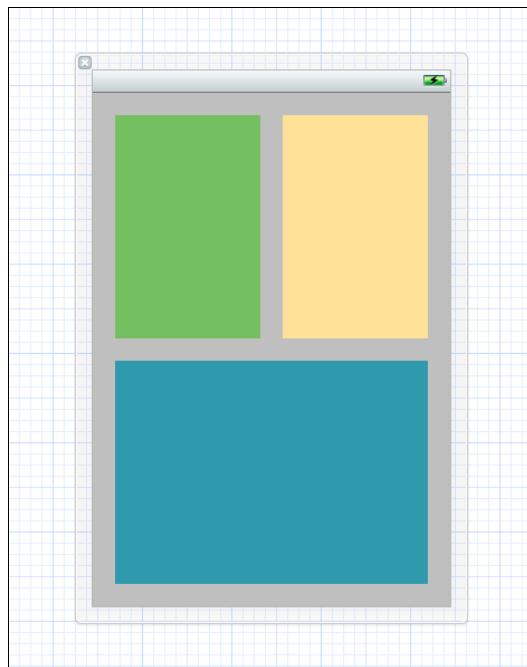
Click on **ViewController.xib** to open it in Interface Builder. Before you do anything else, first disable Auto Layout for this nib. You do that in the File inspector:



Uncheck the “Use Autolayout” box. Now the nib uses the old struts-and-springs model.

Note: Any new nib or storyboard files that you create with Xcode 4.5 or better will have Auto Layout activated by default. Because Auto Layout is an iOS 6 feature only, if you want to use Xcode 4.5 to make apps that are compatible with iOS 5, you need to disable Auto Layout on any new nibs or storyboard files by unchecking the “Use Autolayout” checkbox.

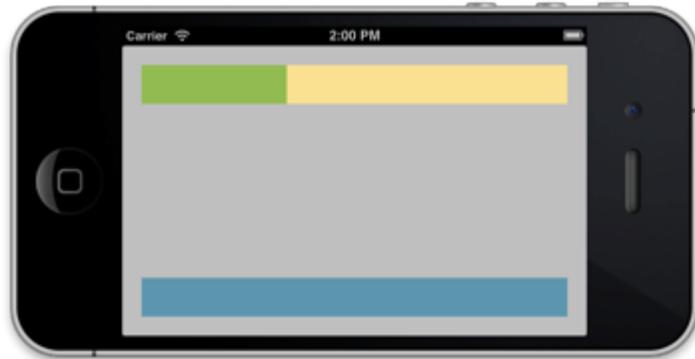
Drag three new views on to the main view and line them up like this:



For clarity, give each view its own color so that you can see which is which.

Each view is inset 20 points from the window’s borders; the padding between the views is also 20 points. The bottom view is 280 points wide and the two views on top are both 130 points wide. All views are 200 points high.

Run the app and rotate the simulator or your device to landscape. That will make the app look like this, not quite what I had in mind:

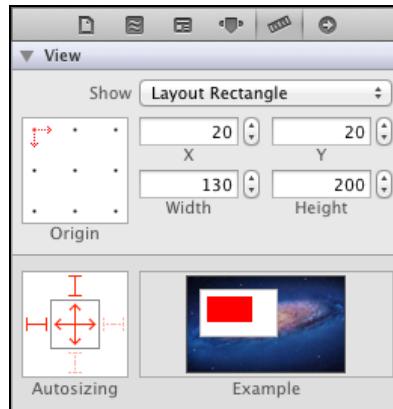


Note: You can rotate the simulator using the **Hardware\Rotate Left** and **Rotate Right** menu options, or by holding down **Cmd** and tapping the left or right arrow keys.

Instead, I want the app to look like this in landscape:



Obviously, the autosizing masks for all three views leave a little something to be desired. Change the autosizing settings for the top-left view to:

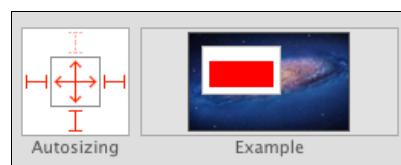


This makes the view stick to the top and left edges (but not the bottom and right edges), and resizes it both horizontally and vertically when the superview changes its size.

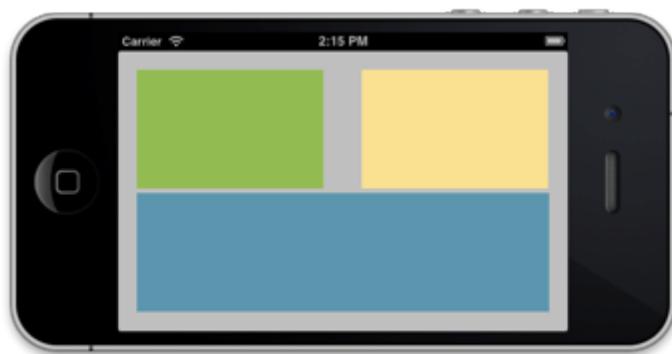
Similarly, change the autosizing settings for the top-right view:



And for the bottom view:



Run the app again and rotate to landscape. It should now look like this:



Close, but not quite. The padding between the views is not correct. Another way of looking at it is that the sizes of the views are not 100% right. The problem is that the autosizing masks tell the views to resize when the superview resizes, but there is no way to tell them *by how much* they should resize.

You can play with the autosizing masks – for example, change the flexible width and height settings (the “springs”) – but you won’t get it to look exactly right with a 20-point gap between the three views.

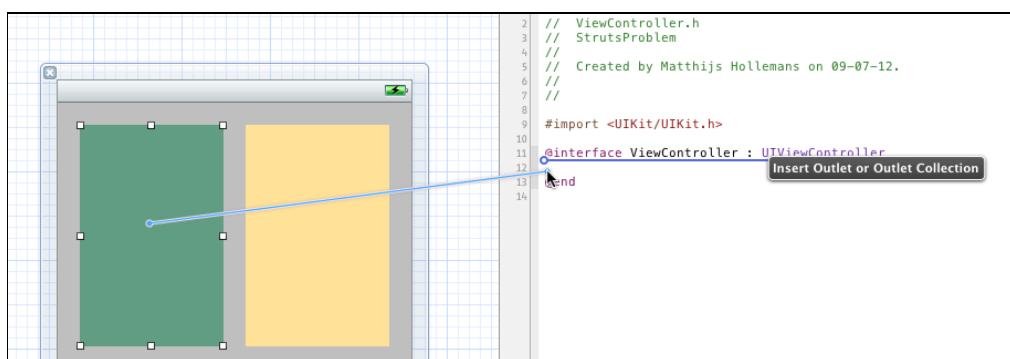


To solve this layout problem with the springs and struts method, unfortunately you will have to write some code.

UIKit sends several messages to your view controllers before, during and after rotating the user interface. You can intercept these messages to make changes to the layout of your UI. Typically you would override `willAnimateRotationToInterfaceOrientation:duration:` to change the frames of any views that need to be rearranged.

But before you can do that, you first have to make outlet properties to refer to the views to be arranged.

Switch to the Assistant Editor mode (middle button on the Editor toolset on the Xcode toolbar) and Ctrl-drag from each of the three views onto **ViewController.h**:



Connect the views to these three properties, respectively:

```
@property (weak, nonatomic) IBOutlet UIView *topLeftView;
@property (weak, nonatomic) IBOutlet UIView *topRightView;
@property (weak, nonatomic) IBOutlet UIView *bottomView;
```

Add the following code to **ViewController.m**:

```
- (void)willAnimateRotationToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
    duration:(NSTimeInterval)duration
{
    [super willAnimateRotationToInterfaceOrientation:
        toInterfaceOrientation duration:duration];

    if (toInterfaceOrientation ==
        UIInterfaceOrientationLandscapeLeft
    || toInterfaceOrientation ==
        UIInterfaceOrientationLandscapeRight)
    {
        CGRect rect = self.topLeftView.frame;
        rect.size.width = 210;
    }
}
```

```
rect.size.height = 120;
self.topLeftView.frame = rect;

rect = self.topRightView.frame;
rect.origin.x = 250;
rect.size.width = 210;
rect.size.height = 120;
self.topRightView.frame = rect;

rect = self.bottomView.frame;
rect.origin.y = 160;
rect.size.width = 440;
rect.size.height = 120;
self.bottomView.frame = rect;
}

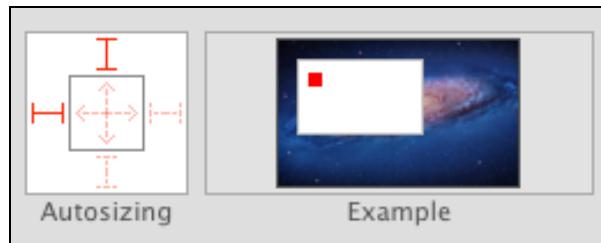
else
{
    CGRect rect = self.topLeftView.frame;
    rect.size.width = 130;
    rect.size.height = 200;
    self.topLeftView.frame = rect;

    rect = self.topRightView.frame;
    rect.origin.x = 170;
    rect.size.width = 130;
    rect.size.height = 200;
    self.topRightView.frame = rect;

    rect = self.bottomView.frame;
    rect.origin.y = 240;
    rect.size.width = 280;
    rect.size.height = 200;
    self.bottomView.frame = rect;
}
}
```

This callback occurs when the view controller is rotating to a new orientation. It looks at the orientation the view controller is rotating to and resizes the views appropriately – in this case with hardcoded offsets based on the known screen dimensions of the iPhone. This callback occurs within an animation block, so the changes in size will animate.

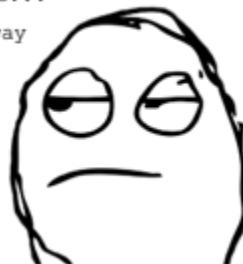
Don't run the app just yet. First you have to restore the autosizing masks of all three views to the following, or the autosizing mechanism will clash with the positions and sizes you set on the views in `willAnimateRotation:`



That should do it. Run the app and flip to landscape. Now the views line up nicely. Flip back to portrait and verify that everything looks good there as well.

It works, but that was a lot of code you had to write for a layout that is actually pretty simple. Imagine the effort it takes for layouts that are truly complex, especially dynamic ones where the individual views change size, or the number of subviews isn't fixed.

There must be...
...another way



Note: Another approach you can take is to make separate nibs for the portrait and landscape orientations. When the device rotates you load the views from the other nib and swap out the existing ones. But this is still a lot of work and it adds the trouble of having to maintain two nibs instead of one.

Auto Layout to the rescue!

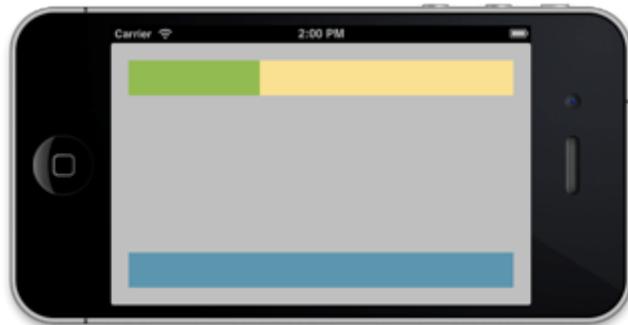
You will now see how to accomplish this same effect with Auto Layout. First, remove `willAnimateRotationToInterfaceOrientation:duration:` from **ViewController.m**, because you're now going to do this without writing any code.

Select **ViewController.xib** and in the File inspector panel, check the "Use Autolayout" box to enable Auto Layout for this nib file:

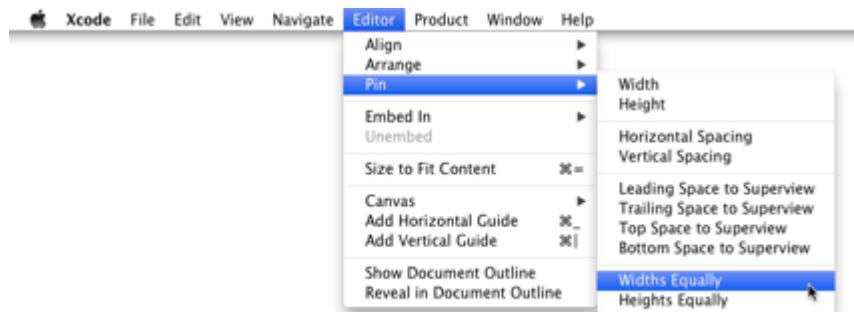


Note: Auto Layout is always enabled for the entire nib or storyboard file. All the views inside that nib or storyboard will use Auto Layout if you check that box.

Run the app and rotate to landscape. It should give the same messed up layout that it did earlier:



Let's put Auto Layout into action. Hold down the **Cmd** key while you click on the two views on the top (the green and yellow ones), so that both are selected. From Xcode's **Editor** menu, select **Pin\Widths Equally**:

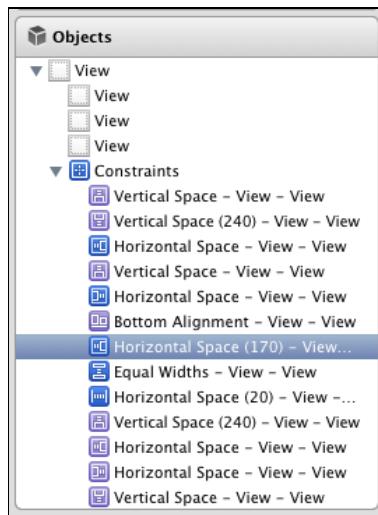


Select the same two views again and choose **Editor\Pin\Horizontal Spacing**. (Even though the two views appear selected after you carry out the first Pin action,

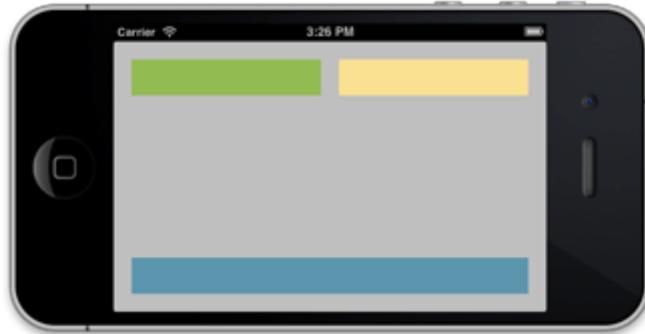
do note that they are in a special layout relationship display mode. So you do have to reselect the two views.)

In the Document Outline on the left, you'll notice a new section named "Constraints". This section was added when you enabled Auto Layout for the nib. You will learn all about what these constraints are and how they operate in the next section.

For now, locate the one named "Horizontal Space (170)" and delete it from the list:



Run the app and rotate to landscape. That looks better already – the views at the top now have the proper widths and padding – but you're not quite there yet:

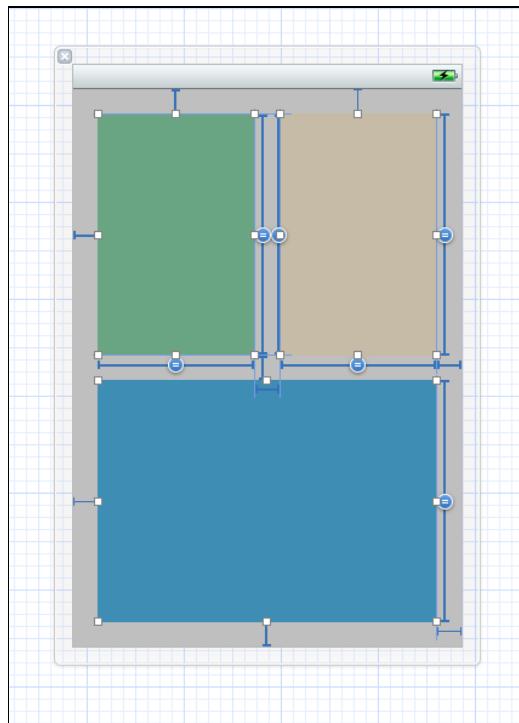


Hold down **Cmd** and select all three views. From the Editor menu, choose **Pin\Heights Equally**.

Now select the top-left corner view and the bottom view (using Cmd as before), and choose **Editor\Pin\Vertical Spacing**.

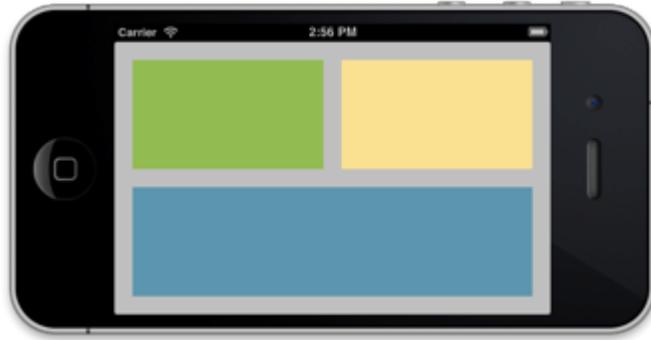
Finally, remove the "Vertical Space (240)" constraint from the list.

If you select all three views at the same time, Interface Builder should show something like this:



The blue “T-bar” shaped things represent the constraints between the views. It might look a bit scary, but it is actually quite straightforward once you learn what it all means.

Run the app and... voila, everything looks good again, all without writing a single line of code!



Cool, but what exactly did you do here? Rather than requiring you to hard-code how big your views are and where they are positioned, Auto Layout lets you express how the views in your layout relate to each other.

You have put the following relationships – what is known as constraints – into the layout:

- The top-left and top-right views always have the same width (that was the first pin widths equally command).

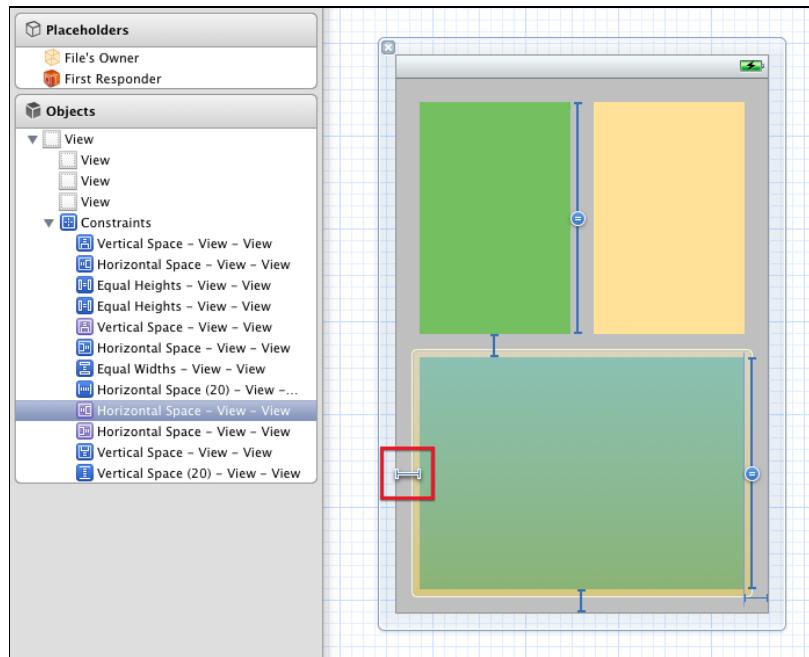
- There is a 20-point horizontal padding between the top-left and top-right views (that was the pin horizontal spacing).
- All the views always have the same height (the pin heights equally command).
- There is a 20-point vertical padding between the two views on top and the one at the bottom (the pin vertical spacing).

And that is enough to express to Auto Layout where it should place the views and how it should behave when the size of the screen changes.

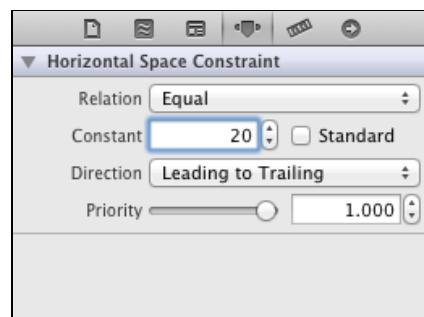


Note: There are also a few other constraints that were brought over from the springs-and-struts layout when you toggled the “Use Autolayout” checkbox. For each of the margins between the views and the edges of the screen there is now a constraint that basically says: “this view always sits at a 20-points distance from the top/bottom/left/right edge.”

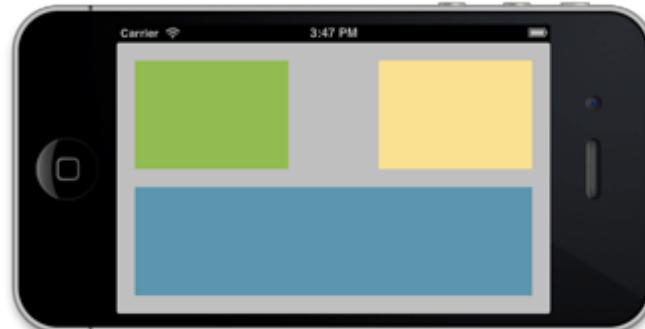
You can see all your constraints in the Document Outline. If you click on a constraint in the Document Outline, Interface Builder will highlight where it sits on the view by drawing a white outline around the constraint and adding a shadow to it so that it stands out:



Constraints are real objects (of class `NSLayoutConstraint`) and they also have attributes. For example, select the constraint that creates the padding between the two top views (it is named “Horizontal Space (20)”) and then switch to the Attributes inspector. There you can change the size of the margin by editing the Constant field.



Set it to 100 and run the app again. Now the margin is a lot wider:



Auto Layout is a lot more expressive than springs and struts when it comes to describing the views in your apps. In the rest of this chapter, you will learn all about constraints and how to apply them in Interface Builder to make different kinds of layouts.

How Auto Layout works

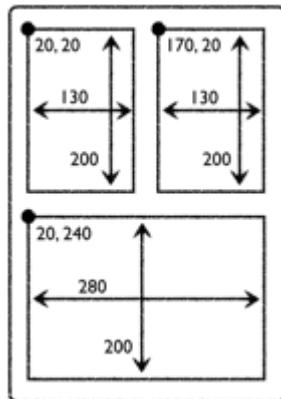
As you've seen in the test drive above, the basic tool in Auto Layout is the *constraint*. A constraint describes a geometric relationship between two views. For example, you might have a constraint that says:

"The right edge of label A is connected to the left edge of button B with 20 points of empty space between them."

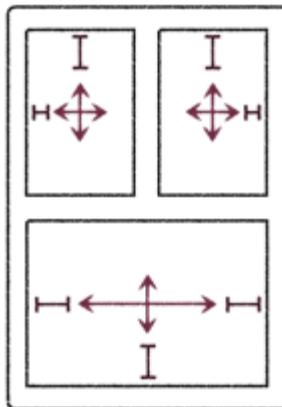
Auto Layout takes all of these constraints and does some mathematics to calculate the ideal positions and sizes of all your views. You no longer have to set the frames of your views yourself – Auto Layout does that for you, entirely based on the constraints you have set on those views.

Before Auto Layout, you always had to hard-code the frames of your views, either by placing them at specific coordinates in Interface Builder, by passing a rectangle into `initWithFrame:`, or by setting the view's `frame`, `bounds` or `center` properties.

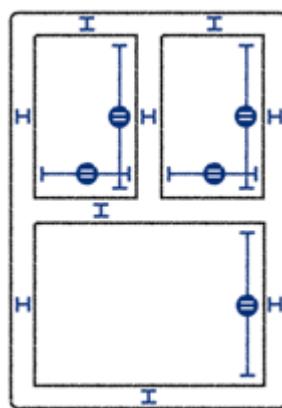
For the app that you just made, you specifically set the frames to:



You also set autosizing masks on each of these views:



That is no longer how you should think of your screen designs. With Auto Layout, all you need to do is this:



The sizes and positions of the views are no longer important; only the constraints matter. Of course, when you drag a new button or label on to the canvas it will have a certain size and you will drop it at a certain position, but that is only a design aid that you use to tell Interface Builder where to put the constraints.

Designing like you mean it

The big advantage of using constraints is that you no longer have to fiddle with coordinates to get your views to appear in the proper places. Instead, you can describe to Auto Layout how the views are related to each other and Auto Layout will do all the hard work for you. This is called *designing by intent*.

When you design by intent, you're expressing *what* you want to accomplish but not necessarily *how* it should be accomplished. Instead of saying: "the button's top-left corner is at coordinates (20, 230)", you now say:

"The button is centered vertically in its superview, and it is placed at a fixed distance from the left edge of the superview."

Using this description, Auto Layout can automatically calculate where your button should appear, no matter how big or small that superview is.

Other examples of designing with intent (and Auto Layout can handle all of these instructions):

"These two text fields should always be the same size."

"These two buttons should always move together."

"These four labels should always be right-aligned."

This makes the design of your user interfaces much more descriptive. You simply define the constraints, and the system calculates the frames for you automatically.

You saw in the first section that even a layout with just a few views needs quite a bit of work to layout properly in both orientations. With Auto Layout you can skip all that effort. If you set up your constraints properly, then the layout should work without any changes in both portrait and landscape.

Another important benefit of using Auto Layout is internationalization. Text in German, for example, is infamous for being very long and getting it to fit into your labels can be a headache. Again, Auto Layout takes all this work out of your hands, because it can automatically resize your labels based on the content they need to display – and have everything else adapt with constraints.

Adding support for German, French, or any other language is now simply a matter of setting up your constraints, translating the text, and... that's it!



The best way to get the hang of Auto Layout is to play with it, so that's exactly what you will do in the rest of this chapter.

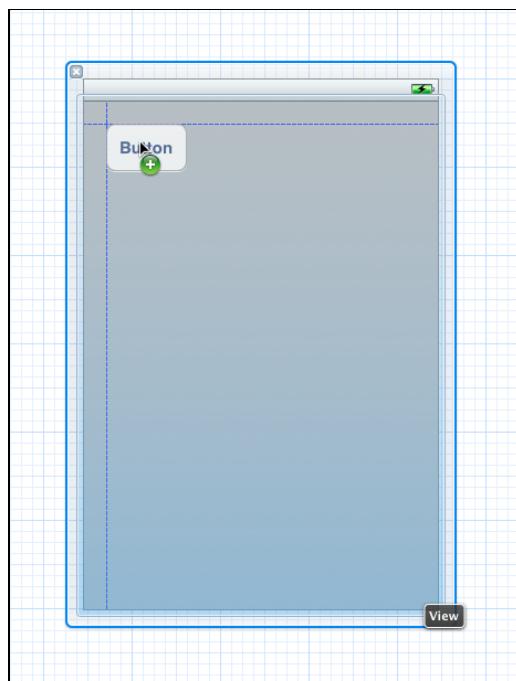
Note: Auto Layout is not just useful for rotation; it can also easily scale your UI up and down to accommodate different screen sizes. It is no coincidence that this technology was added to iOS at the same time that the iPhone 5 and its taller screen came out! Auto Layout makes it a lot easier to stretch your apps' user interfaces to fill up that extra vertical space on the iPhone 5. And who knows what will come of the rumored "iPad mini"... With Auto Layout you will be prepared for the future.

Courting constraints

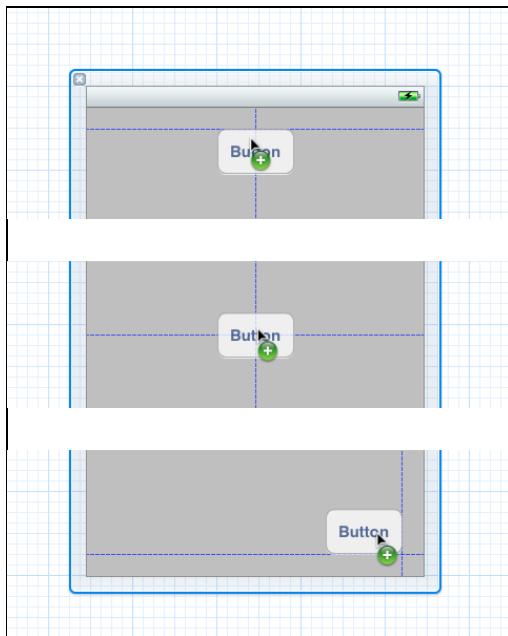
Close your current project and create a new project using the Single View Application template. Name it "Constraints". This will be an iPhone project that does not use storyboards, but it does use Automatic Reference Counting.

Any new projects that you create with Xcode 4.5 automatically assume that you will be using Auto Layout, so you do not need to do anything special to enable it.

Click on **ViewController.xib** to open Interface Builder. Drag a new Round Rect Button onto the canvas. Notice that while you're dragging, dashed blue lines appear. These lines are known as the *guides*:

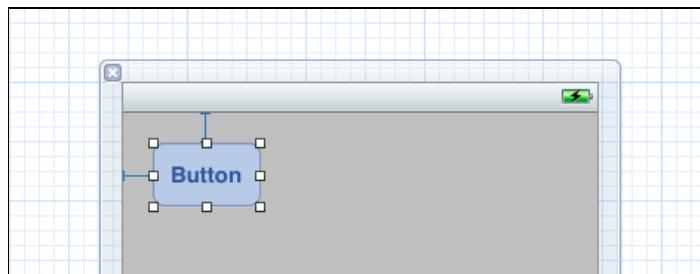


There are guides around the margins of the screen, as well as in the center:



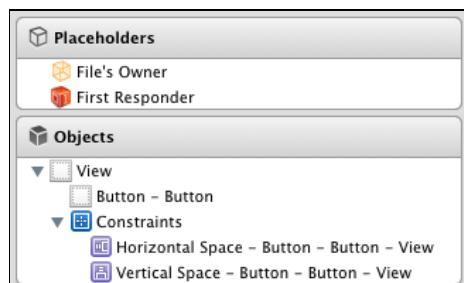
If you have used Interface Builder before, then you have no doubt seen these guides. They are helpful hints that make it easier to align stuff. With Auto Layout enabled, however, the guides have a different purpose. You still use them for alignment, but they also tell you where the new constraints will go.

Drop the button in the top-left corner against the blue guides. Now the nib looks like this:



There are two blue thingies attached to the button. These T-bar shaped objects are the constraints that are set on this button.

All the constraints are also listed in the Document Outline pane on the left-hand side of the Interface Builder window:



There are currently two constraints, a Horizontal Space between the button and the left edge of the main view, and a Vertical Space between the button and the top edge of the main view. The relationship that is expressed by these constraints is:

"The button always sits at the top-left corner in its superview."

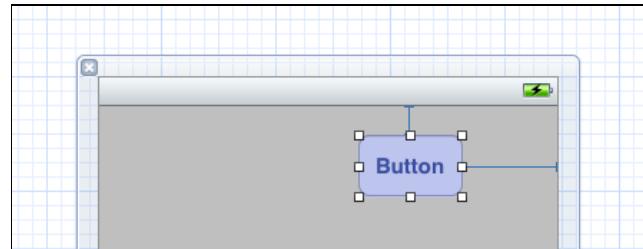
Now pick up the button and place it in the nib's top-right corner, again against the blue guides:



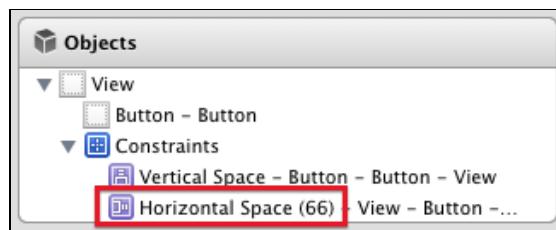
The Horizontal Space constraint has changed. It is no longer attached to the button's left side, but to its right.

When you place a button (or any other view) against the guides, you get a constraint with a standard size that is defined by the "HIG", Apple's iOS Human Interface Guidelines document. For margins around the edges, the standard size is a space of 20 points.

Even if you place the button where there is no guide, you still get a Horizontal or Vertical Space constraint to keep it in place. Try it out. Drag the button a bit to the left until you get something like this:

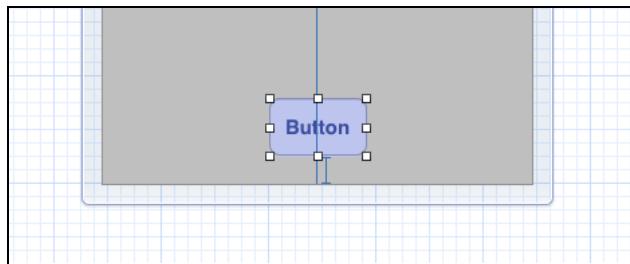


There is still a Horizontal Space constraint, but it's larger now. In the Document Outline, you can see that it no longer has a standard space:



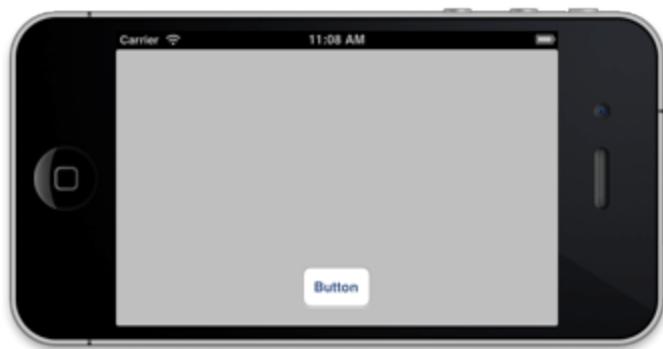
What constraints you get depend on where you place the button.

There is also a “center” constraint. Drag the button to the bottom center of the canvas, so that it snaps into place with the guides:



Notice that the Horizontal Space constraint has been replaced by a Center X Alignment constraint, which means that the button is always center-aligned with its superview, on the horizontal axis. There is still a Vertical Space constraint to keep the button away from the bottom of the view (again, using the standard margin).

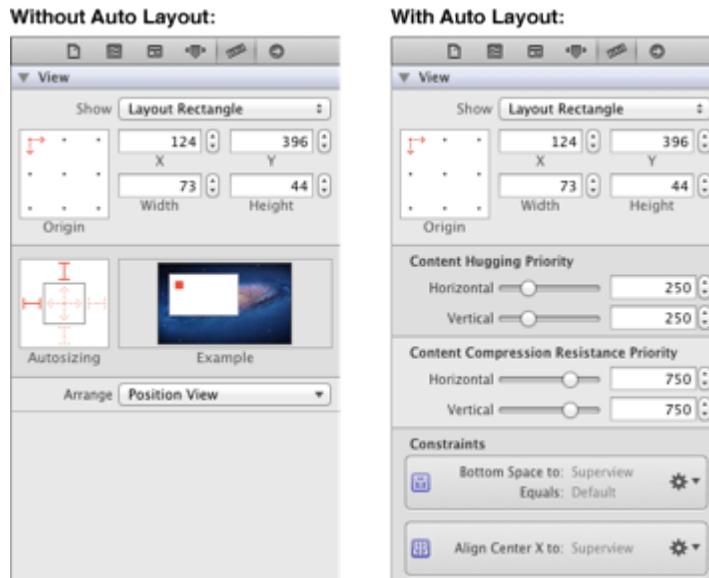
Run the app and rotate it to landscape. Even in landscape mode, the button stays at the bottom center of the screen:



That's how you express intent: “This button should always be at bottom center.” Notice that nowhere did you have to tell Interface Builder what the button's coordinates are, only where you want it anchored in the view.

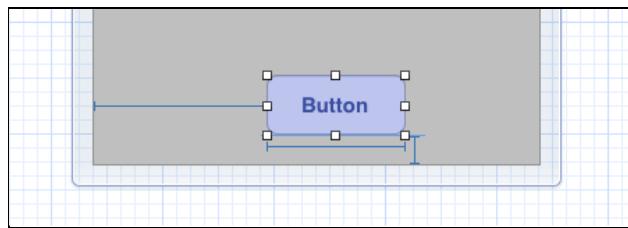
With Auto Layout, you're no longer supposed to care about the exact coordinates of where you place your views on the canvas or what their size is. Instead, Auto Layout derives these two things from the constraints that you set (or that Interface Builder sets for you).

You can see this paradigm shift in the Size inspector for the button, which is now quite different:



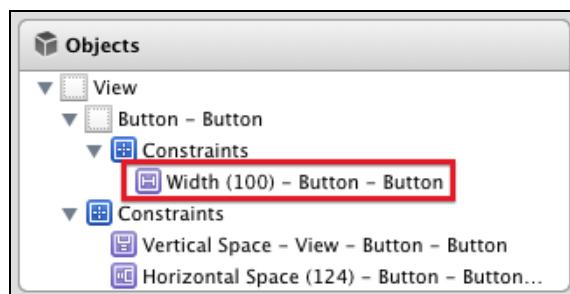
With Auto Layout disabled, typing into the X, Y, Width or Height fields will change the position and size of the selected view. With Auto Layout enabled, you can still type new values into these fields, but that may not always have the effect you want. The view will move, but Interface Builder will also calculate new constraints based on your new values.

For example, change the Width value to 100. The canvas turns into something like this:



The Center X Alignment constraint has disappeared, and in its place is a Horizontal Space that glues the button to the left edge of the screen, as well as a new constraint on the button itself that forces it to have a width of 100 points (the blue bar below the button).

You can also see this new Width constraint in the Document Outline on the left:



Unlike the other constraints, which are between the button and its superview, the Width constraint only applies to the button itself. You can think of this as a constraint between the button and... the button.

Drag the button so that it snaps with the Center X Alignment constraint again.

Tip: Because changing the position and size in the Size inspector may mess up your constraints, I advise against doing this. Instead, if you want to make changes to the layout, change the constraints.

You may wonder why the button did not have a Width constraint before. How did Auto Layout know how wide to make the button without it?

Here's the thing: the button itself knows how wide it must be. It calculates this based on its title text plus some padding for the rounded corners. If you set a background image on the button, it also takes that into account.

This is known as the *intrinsic content size*. Not all controls have this, but many do (`UILabel` is another example). If a view can calculate its own preferred size, then you do not need to set specific Width or Height constraints on it. You will see more of this later.



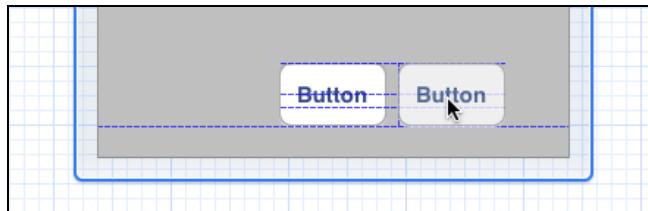
To return the button to its optimal size, select it and choose **Size to Fit Content** from the **Editor** menu. This gets rid of the explicit Width constraint and restores the button's intrinsic content size.

It takes two to tango

Guides do not appear only between a view and its superview, but also between views on the same level of the view hierarchy. To demonstrate this, drag a new round rectangular button onto the canvas.

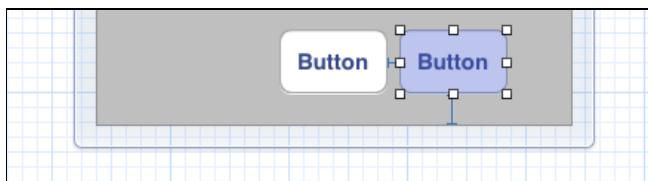
If you put it far enough away from the other button, then this new button gets its own constraints. However, if you drag them close to each other, then their constraints start to interact.

Put the new button next to the existing one so that it snaps into place:



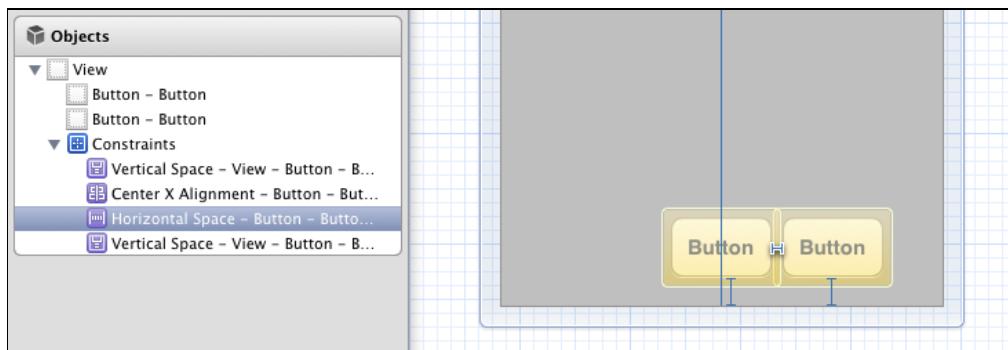
There are quite a few dotted guidelines here, but Interface Builder doesn't turn them all into constraints; that would be a bit much. It basically recognizes that these two buttons can align in different ways – at their tops, centers and baselines.

After dropping the button in place, the constraints look like this:



The new button has a Vertical Space to the bottom of the screen, but also a Horizontal Space that links it with the other button. Because this space is small (only 8 points), the T-bar may be a bit hard to see, but it is definitely there.

Click on the Horizontal Space constraint in the Document Outline to select it:

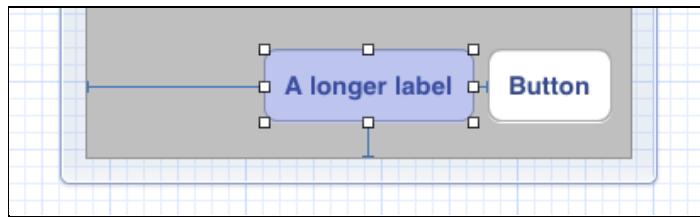


When you select a constraint, it lights up the controls it belongs to. This particular constraint sits between the two buttons. What you've done here is say:

"The second button always appears on the right of the first one, no matter where the first button is positioned or how big it is."

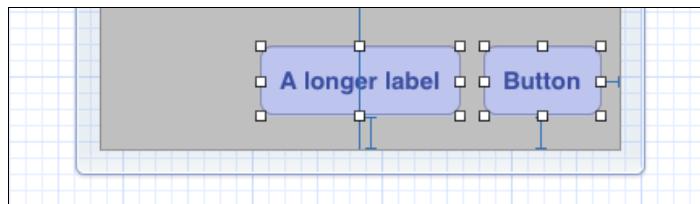
Select the button on the left and type something long into its label like "A longer label". When you're done, the button resizes to make room for the new text, and

the other button shifts out of the way. After all, it is attached to the first button's right edge, so that is exactly what you intended to happen:



Notice that Interface Builder replaced the Center X Alignment on the first button with a Horizontal Space again. Every time you make a change to the size (or position) of your controls, Interface Builder will recalculate a set of constraints that it thinks are optimal. Often it does the right thing, but sometimes it completely misses the boat. Obviously you just wanted to change the text in the button but keep it centered here.

Drag the button back so that it is centered again. Take a look at the constraints now:

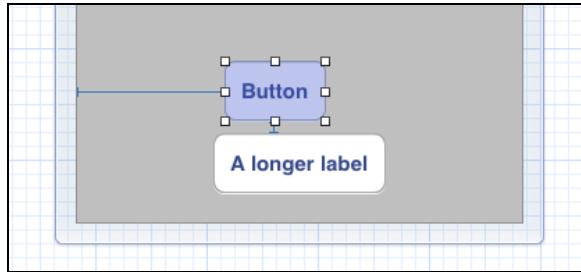


That is probably not what you wanted to happen. The two buttons are no longer connected to each other. Instead, the rightmost button now has a Horizontal Space to the right edge of the screen. There is no longer a Horizontal Space constraint between them.

Of course, you can reconnect these two buttons by snapping them together again, but this problem could have been avoided by not dragging views around.

First, press **Cmd-Z** to undo, so that the first button is no longer center-aligned. Now select that button and from the **Editor** menu select **Align\Horizontal Center in Container**. This time not only the first button moves to the center – the other button moves along with it. That is more like it!

Just to get a better feel for how this works, play with this some more. Select the smaller button and put it above the other one, so that they snap into place vertically (but don't try to align the left edges of the two buttons):



Because you snapped the two buttons together, there is now a Vertical Space between them. Again it has the standard spacing of 8 points that is recommended by the HIG.

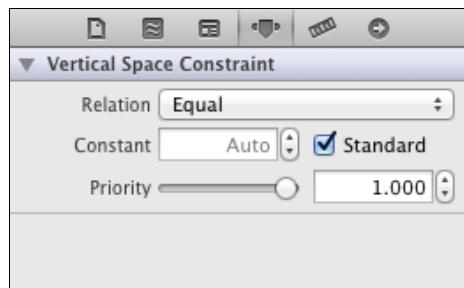
Note: The “HIG”, which is short for iOS Human Interface Guidelines, contains Apple’s recommendations for designing good user interfaces. It is mandatory reading for any iOS developer. The HIG explains which UI elements are appropriate to use under which circumstances, and best practices for using them.

You can find this document at:

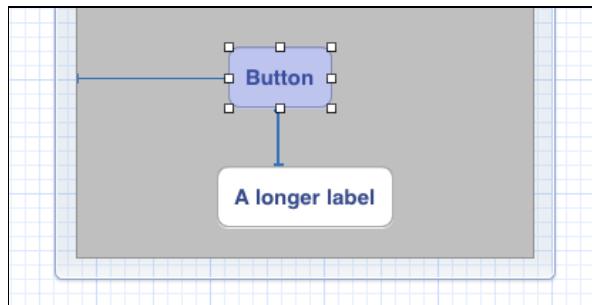
<http://developer.apple.com/library/ios/#DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html>

You are not limited to standard spacing between controls, though. Constraints are full-fledged objects, just like views, and therefore have attributes that you can change.

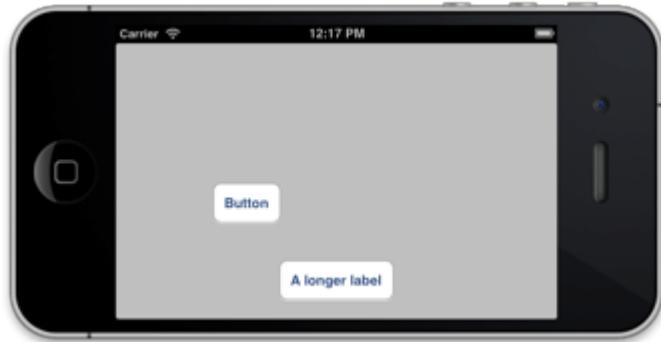
Select the Vertical Space constraint between the two buttons. You can do this in the canvas by clicking the T-bar, although that tends to be a bit finicky. By far the easiest method is to click on the constraint in the Document Outline. Once you have it selected, switch to the Attributes inspector:



By default the Standard attribute is checked. For a space constraint between two objects this is 8 points; for a margin around the edges of the superview it is 20 points. Type 40 into the Constant field to change how big the constraint is. Now the two buttons will be further apart, but they are still connected:

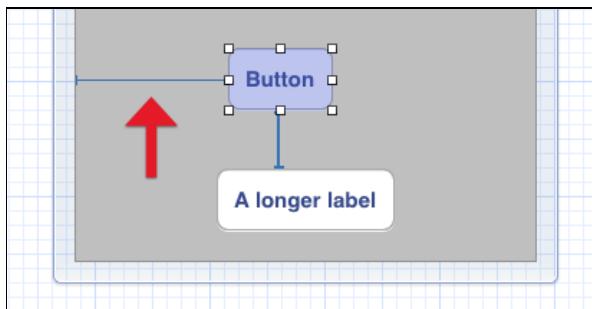


Run the app and flip to landscape to see the effect:



The buttons certainly keep their vertical arrangement, but not their horizontal one!

If you look at the nib, you'll see a Horizontal Space between the top button and the left edge of the canvas (at least if you placed the button roughly in the same spot that I did):



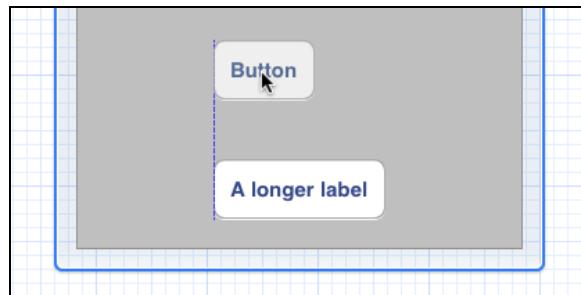
The bottom label is horizontally centered in the screen, but the top button isn't – it always keeps the same distance from the left edge.

That doesn't look very nice, so instead you are going to express the following intention:

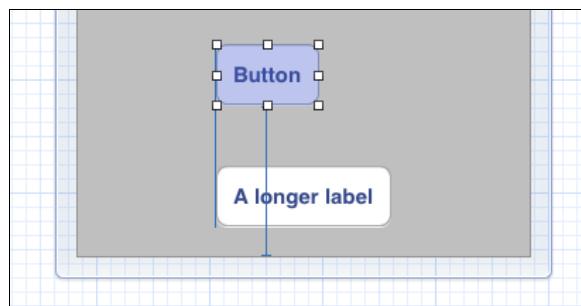
"The bottom button will always be horizontally centered, and the top button will align its left edge with the left edge of the bottom button."

You already have a constraint for the first condition, but not for the second.

Interface Builder shows guides for alignment, so you can drag the top button until its left edge snaps with the left edge of the bottom button:



Unfortunately, this also removes the Vertical Space between the two buttons (at least sometimes, depending on how things are dragged/placed). Interface Builder simply “forgets” it was there and replaces it with a Vertical Space to the bottom of the view:

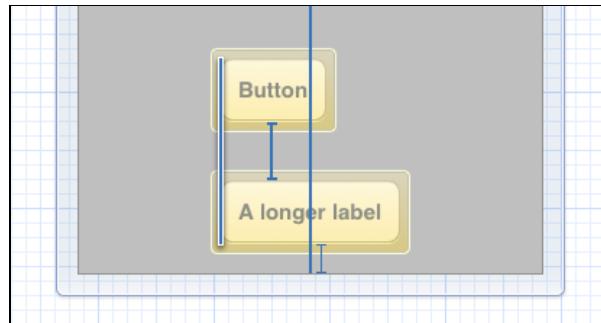


That is not quite what you want. Instead, there should be a Vertical Space between these two buttons, not one that extends all the way to the window edge. Here's a comic that shows how that might feel when this happens:



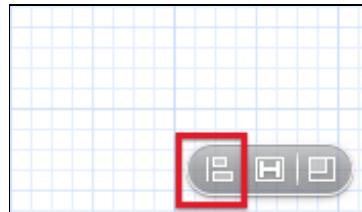
As I mentioned before, dragging around views is not a good idea if you want to keep your constraints intact. There is a better way to align these two buttons.

First, undo the change so that the top button moves back to its previous, unaligned position. Then hold down **Cmd** and click both buttons to select them. Now from the **Editor** menu, pick **Align\Left Edges**. This tells Interface Builder that you want to left-align these two controls, leaving the existing constraints intact:

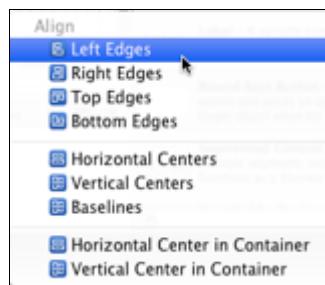


As you can see, your previous constraints are still there – the Center X Alignment on the bottom button and the Vertical Space between the two buttons – while a new “Leading Alignment” constraint was added to keep the top button left-aligned with the bottom one.

Tip: You don't always need to go to the **Editor** menu to pick an alignment option. Interface Builder has a shortcut menu in the bottom-right corner:

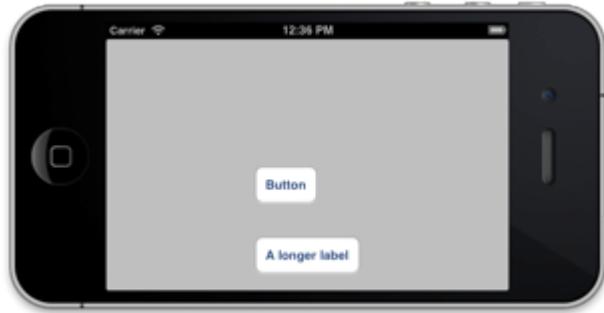


The left-most button opens the Align menu:



Because you'll be using these options a lot, using the shortcut menu will save you some time.

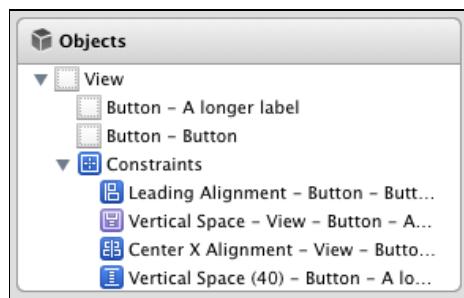
Run the app and rotate to landscape to verify that it works:



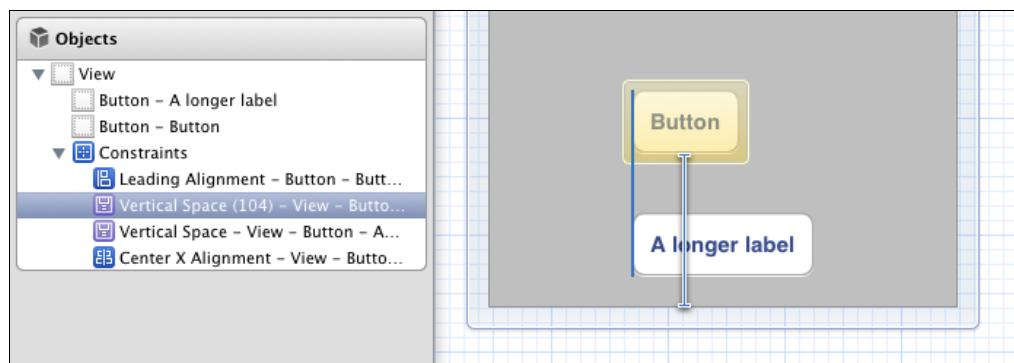
As bold as a user constraint

Maybe you noticed that some of the T-bars in the canvas are thicker than others. The bold ones are called *user constraints*, and unlike the thin ones you can delete them. However, when you delete a user constraint, Interface Builder will often put a non-deletable constraint in its place. You will soon see why.

In the Document Outline, user constraints have a blue icon:



Select the Vertical Space (40) constraint and tap the Delete key on your keyboard. The T-bar between the two buttons disappears and is replaced by a new Vertical Space constraint that goes all the way to the bottom:



This new constraint has a purple icon and does not have a bold line, meaning that you cannot delete this one. The two buttons are no longer connected vertically, although they are still left-aligned due to the Leading Alignment constraint.

Why does this happen? Why does Interface Builder attach a new Vertical Constraint to the button, even though you just told it to delete such a constraint? The answer is this:

For each view there must always be enough constraints to determine both its position and size.

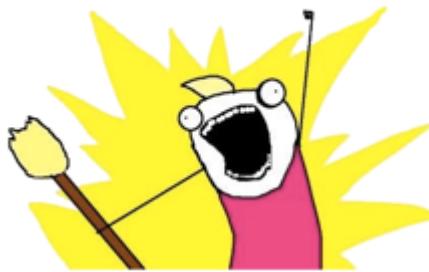
That is the most important rule to remember when it comes to using Auto Layout. If there aren't enough constraints, then Auto Layout will be unable to calculate where your views should be positioned or how big they should be. Such a layout is considered to be invalid. You will see examples of such invalid layouts later on.

Interface Builder tries very hard to prevent you from making layouts that are invalid. The size of these two buttons is known because buttons know how big they should be, based on their text, background image, and so on – intrinsic content size, remember? So that's not a problem. The X-position of the top button is also known because its left edge is aligned with the bottom button, and the bottom button is always horizontally centered. The only unknown is the Y-position.

Previously, the two buttons were connected with a Vertical Space. That was enough to determine the Y-position of the top button. But if you delete that Vertical Space, then the top button has nothing to anchor it vertically in the view. It cannot just float there because then Auto Layout has no way to determine what its Y-coordinate should be.

To prevent this from happening, Interface Builder needs to “pin” the button somewhere and the bottom edge is closest.

PIN ALL THE BUTTONS

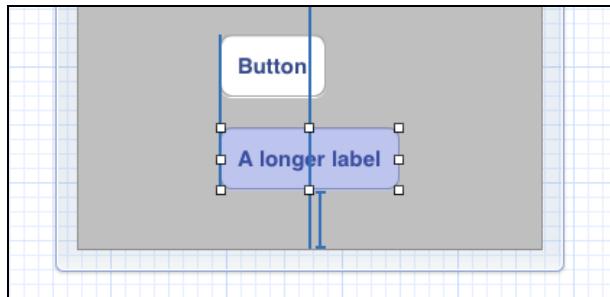


Funnily enough, if you run the app and flip to landscape, it still seems to work. The screen looks exactly the same as it did before. That is true, but your design is fundamentally different: both buttons are now connected to the bottom of the window. This means that if the bottom button moves, the top one doesn't move with it. (Note that either solution is fine, it just depends on what you want your app to do. But in this example, you want to have a vertical connection between the two buttons.)

To illustrate this, select the Vertical Space constraint between the lower button and the screen's edge. Go into the Attributes inspector. Its Constant should currently

read "Auto", and Standard is checked because this is a standard margin space. Change it to 40.

Because the buttons are not connected, only the lower button moves upward; the top button stays put:

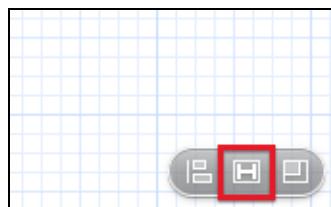


Notice that changing the Constant value of the constraint promoted it to a bold "user" constraint.

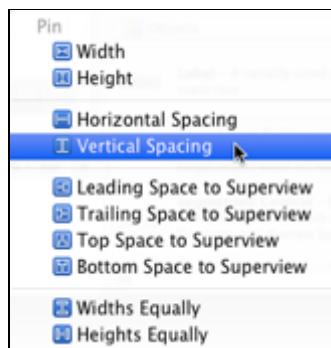
Needles and pins

Let's connect the two buttons again. So far you have made constraints by dragging the buttons on the canvas, but you can also make them afterwards. Hold down the **Cmd** key and click both buttons to select them. From the **Editor** menu, choose **Pin\Vertical Spacing**.

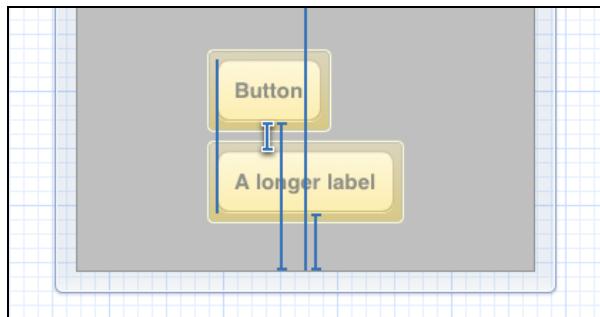
You can also use the little panel in the bottom-right corner to make this constraint:



It pops up the following menu:



Regardless of the method you choose, this adds a new constraint between the two buttons:

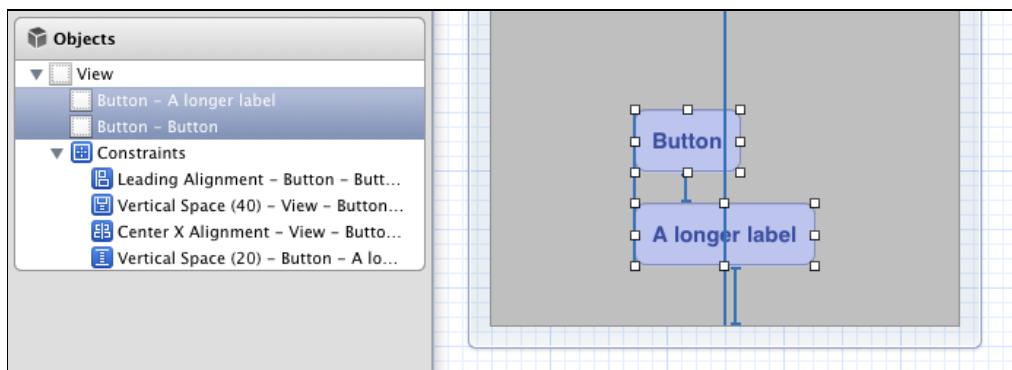


The new constraint is a Vertical Space constraint with a Constant of 20 points. That is because the distance between the two buttons was 20 points at the time you made this connection.

Notice that the old Vertical Space from the upper-most button to the bottom edge is still there. This constraint – the one that says Vertical Space (104) – is no longer needed, so delete it.

Previously when you deleted a blue constraint, a purple one took its place. Now that does not happen, because the remaining constraints are sufficient to position all the views. Interface Builder only adds new constraints when the existing ones are no longer adequate.

You should now have the following constraints:



Select the bottom Vertical Space (by clicking on the canvas) and change its Constant from 40 back to Standard. This should not only move the bottom button downwards, but the top button as well, because they are connected again.

A little runtime excursion

You've seen a bit of the basics now: you know how to place controls using the guides, how to align them relative to one another, and how to put space between controls. Over the course of this chapter you will also use the other options from the Align and Pin menus.

Playing with this in Interface Builder is all well and good, but let's see how this works at runtime. Add the following method to **ViewController.m**:

```
- (IBAction)buttonTapped:(UIButton *)sender
{
    if ([[sender titleForState:UIControlStateNormal]
          isEqualToString:@"X"])
        [sender setTitle:@"A very long title for this button"
          forState:UIControlStateNormal];
    else
        [sender setTitle:@"X" forState:UIControlStateNormal];
}
```

This simply toggles between a long title and a short title for the button that triggered the event. Connect this action method to both of the buttons in Interface Builder: Ctrl-drag from each button to File's Owner and select `buttonTapped:` in the popup.

Run the app and tap the buttons to see how it behaves. Perform the test in both portrait and landscape orientations.



Regardless of which button has the long title and which has the short title, the layout always satisfies the constraints you have given it:

- The lower button is always center-aligned in the window, horizontally.
- The lower button always sits 20 points from the bottom of the window.
- The top button is always left-aligned with the lower button.

That is the entire specification for your user interface.

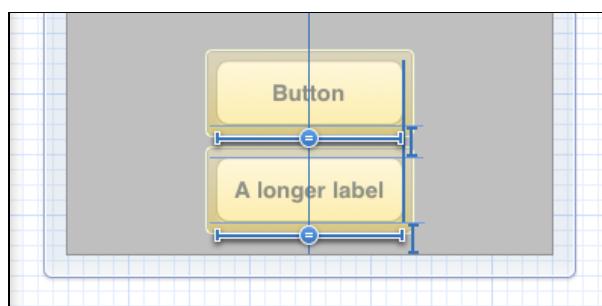
For fun, select both buttons in Interface Builder and from the Align menu pick **Right Edges**. Now run the app again and notice the differences.

Repeat, but now choose **Align\Horizontal Centers**. That will always center the top button with respect to the bottom button. Run the app and see how the buttons act when you tap them.

Fixing the width

The Pin menu has an option for Widths Equally. If you set this constraint on two views, then Auto Layout will always make both views equally wide, based on which one is the largest. Let's play with that for a minute.

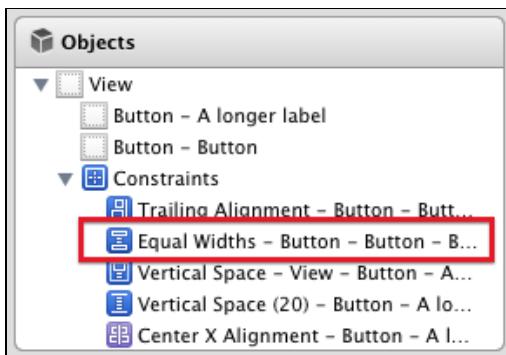
Select both buttons and choose **Pin\Widths Equally**. This adds a new constraint to both buttons:



Note: If you get an extra unintended constraint between one of the buttons and the superview, select the two buttons and select Align\Horizontal Centers again.

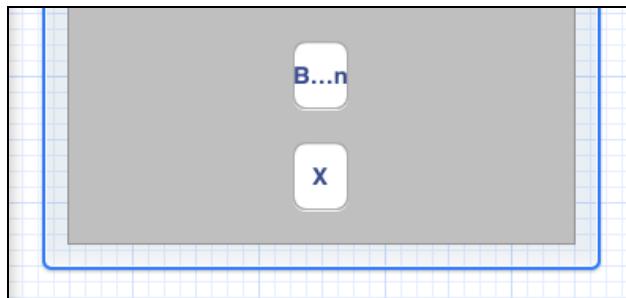
You have seen this type of constraint before, in the first section of the chapter. It looks like the usual T-bar but in the middle it has a circle with an equal sign.

In the Document Outline this shows up as a single Equal Widths constraint:

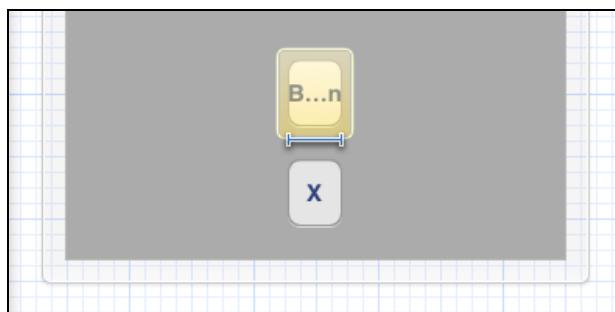
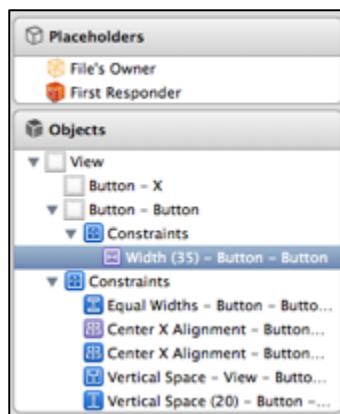


Changing the label text on one button will now change the size of the other one as well.

Change the bottom button's label to "X", just to make it really small. You will notice that the top button no longer fits its text:



So how does Interface Builder know which button's size to use for both of them? If you pay close attention, you'll see that a Width constraint was added to the button with the truncated text:



Interface Builder does this to force the button to become smaller than what it would ideally be, in order to comply with the Equal Widths constraint.

Obviously this is not what you want, so select the top button and choose **Size to Fit Content** from the **Editor** menu (or press **Cmd =**). Now the text fits inside the button again – or rather, the button fits around the text – and the Width constraint is gone.

Run the app and tap the buttons. The buttons always have the same width, regardless of which one has the largest label:



Of course, when both labels are very short, both buttons will shrink equally. After all, unless there is a constraint that prevents, buttons will size themselves to fit their content exactly, no more, no less. What was that called again? Right, the intrinsic content size.

Intrinsic Content Size

Before Auto Layout, you always had to tell buttons and other controls how big they should be, either by setting their `frame` or `bounds` properties or by resizing them in Interface Builder. But it turns out that most controls are perfectly capable of determining how much space they need, based on their content.

A label knows how wide and tall it is because it knows the length of the text that has been set on it, as well as the font size for that text. Likewise for a button, which might combine the text with a background image and some padding for the rounded corners.

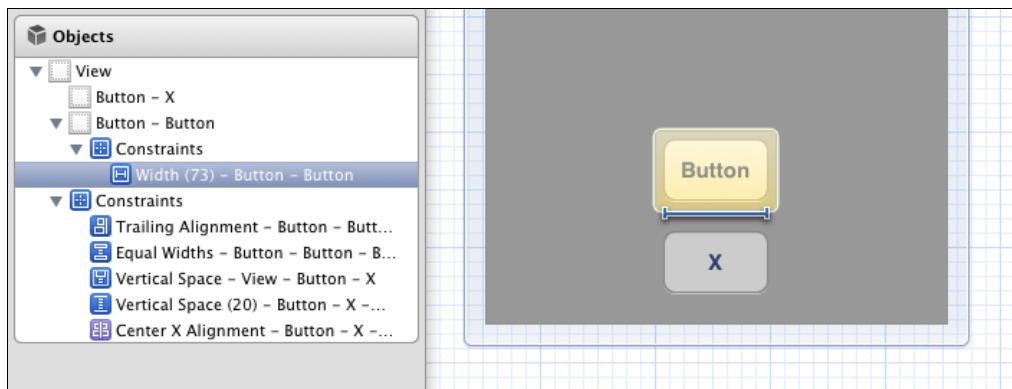
The same is true for segmented controls, progress bars, and most other controls, although some may only have a predetermined height but an unknown width.

This is known as the *intrinsic content size*, and it is an important concept in Auto Layout. You have already seen it in action with the buttons. Auto Layout asks your controls how big they need to be and lays out the screen based on that information.

You can prevent this by setting an explicit Width or Height constraint on a control. If you resize the control by hand, then Interface Builder will set such an explicit constraint for you. With the **Size to Fit Content** command, you remove any fixed Width or Height constraints and let the control determine its intrinsic content size again.

Usually you want to use the intrinsic content size, but there are some cases where you may *not* want to do that. Imagine what happens when you set an image on a `UIImageView` if that image is much larger than the screen. You usually want to give image views a fixed width and height and scale the content, unless you want the view to resize to the dimensions of the image.

So what happens when one of the buttons has a fixed Width constraint on it? Buttons calculate their own size, but you can override this by giving them a fixed width. Select the top button and choose **Pin\Width** from the menu. This adds a solid T-bar below the button:



Because this sort of constraint only applies to the button itself, not to its superview, it is listed in the Document Outline below the button object. In this case, you have fixed the button to a width of 73 points.

Run the app and tap the buttons. What happens? The button text does change, but it gets truncated because there is not enough room:



Because the top button has a fixed-width constraint and both buttons are required to be the same size, they will never shrink or grow.

Note: You probably wouldn't set a Width constraint on a button by design – it is best to let the button use its intrinsic size – but if you ever run into a layout problem where you expect your controls to change size and they don't, then double check to make sure Interface Builder didn't sneak a fixed Width constraint in there.

Play around with this stuff for a bit to get the hang of pinning and aligning views. Get a feel for it, because not everything is immediately obvious. Just remember that there must always be enough constraints so that Auto Layout can determine the position and size for all views.



Gallery example

You should now have an idea of what constraints are and how you can build up your layouts by forging relationships between the different views. In the following sections, you will see how to use Auto Layout and constraints to create layouts that meet real-world scenarios.

Let's pretend you want to make an app that has a gallery of your favorite programmers. It looks like this in portrait and landscape:

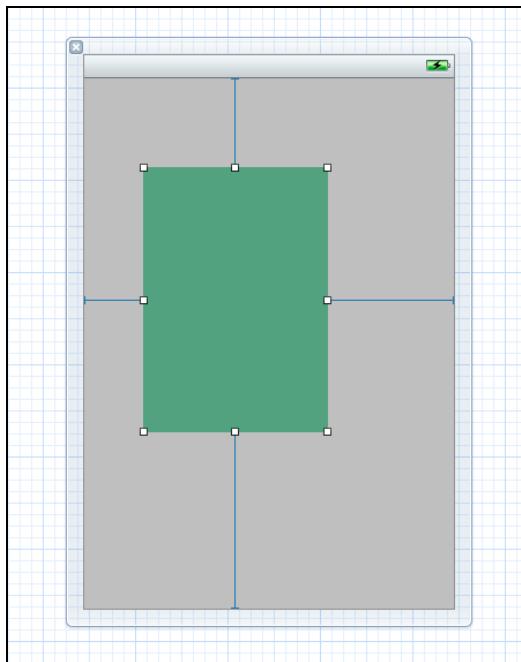


The screen is divided into four equal quarters. Each quarter has an image view and a label. How would you approach this?

Let's start by setting up the basic app. You can use your existing "Constraints" app by deleting the buttons and reusing the view.

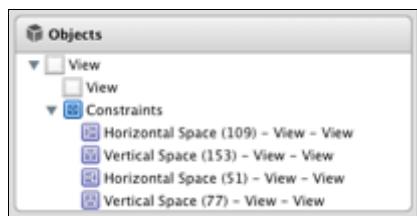
Or, you can create a new project using the Single View Application template and name it as you like, for instance, "Gallery". This will just use a nib, so disable the storyboards option.

Open **ViewController.xib**. From the Object Library, drag a plain view object onto the canvas. Resize the view so that it is 160 by 230 points, and change its background color to be something other than white (I made mine green):



This view has four constraints to keep it in place. Unlike a button or label, a plain `UIView` does not have an intrinsic content size. There must always be enough constraints to determine the position and size of each view, so this view also needs constraints to tell it what size it needs to be.

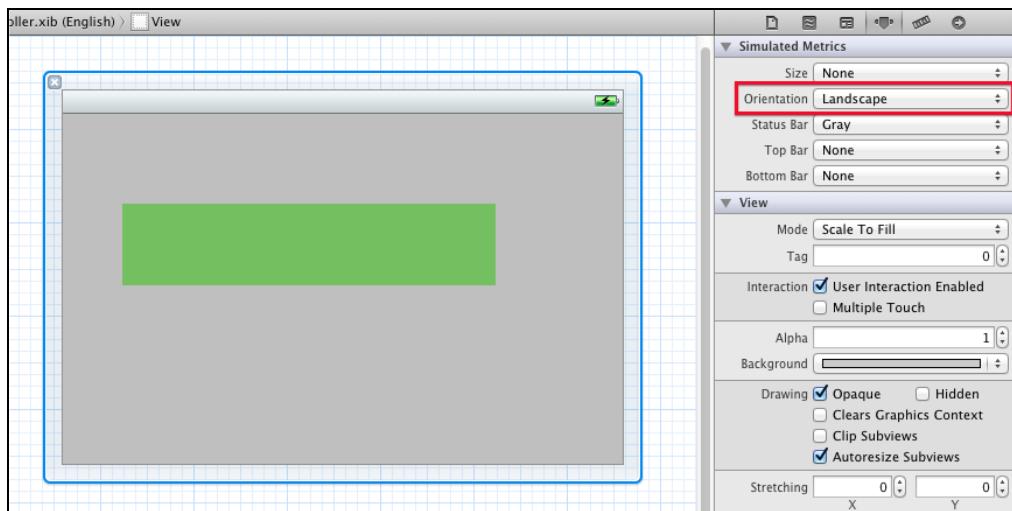
You may wonder, where are these size constraints? In this case, the size of the view is implied by the size of the superview. The constraints in this layout are two Horizontal Spaces and two Vertical Spaces, and these all have fixed lengths. You can see this in the Document Outline:



The width of the green view is calculated by the formula “width of superview minus $(109 + 51)$ ” and its height by the formula “height of superview minus $(153 + 77)$ ”. The space constraints are fixed, so the view has no choice but to resize. When you rotate the app, the dimensions of the superview change from 320x460 to 480x300. Plug this new width and height into these formulas, and you’ll get the new size of the green view.

You can see this for yourself when you run the app and flip to landscape, but you can also simulate it directly in Interface Builder.

Select the top-most view in the nib and go to the Attributes inspector. Under the Simulated Metrics section, change Orientation to Landscape:



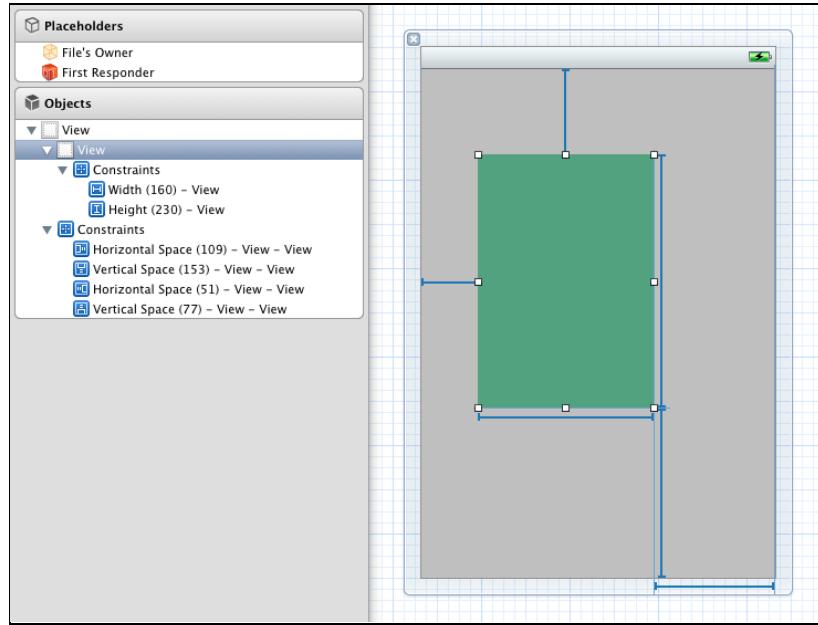
This gives you an instant preview of what the nib's layout will look like in landscape orientation. The green view has resized in order to satisfy its Horizontal and Vertical Space constraints.

Switch back to portrait orientation.

Note: There are two main reasons why you would drop a plain `UIView` onto a nib: a) You're going to use it as a container for other views, which helps with organizing the content of your nibs; or b) It is a placeholder for a custom view or control, and you will also set its `Class` attribute to the name of your own `UIView` or `UIControl` subclass.

You may not always want your `UIView` to resize when the device rotates, so you can use constraints to give the view a fixed width and/or height. Let's do that now. Select the green view and from the **Pin** menu, choose **Width**. Select the view again and choose **Pin\Height**.

You have now added two new constraints to the view, a 160 point Width constraint and a 230 point Height constraint:



Because Width and Height apply to just this view, they are located in the Document Outline under the View itself. Usually, constraints express a relationship between two different views – for example, the Horizontal and Vertical Space constraints are between the green view and its gray superview – but you can consider the Width and Height constraints to be a relationship between the view and itself.

Run the app. Yup, looks good in portrait. Now flip over to landscape. Whoops! Not only does it not look like you wanted – the view has changed size again – but the Xcode debug pane has dumped a nasty error message:

```
Gallery[68932:11303] Unable to simultaneously satisfy constraints.
```

Probably at least one of the constraints in the following list is one you don't want. Try this: (1) look at each constraint and try to figure out which you don't expect; (2) find the code that added the unwanted constraint or constraints and fix it. (Note: If you're seeing

`NSAutoresizingMaskLayoutConstraint` constraints that you don't understand, refer to the documentation for the `UIView` property

`translatesAutoresizingMaskIntoConstraints`)

(

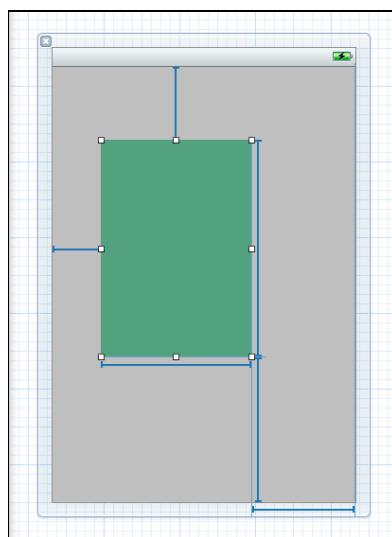
```
"<NSLayoutConstraint:0x754dac0 V:[UIView:0x754e510(230)]>",
"<NSLayoutConstraint:0x754eac0 V:|-(77)-[UIView:0x754e510]   (Names:
' | ':UIView:0x754e3a0 )>",
"<NSLayoutConstraint:0x754ea40 V:[UIView:0x754e510]-(153)-|   (Names:
' | ':UIView:0x754e3a0 )>",
"<NSAutoresizingMaskLayoutConstraint:0x7558cd0 h=-&- v=-&-
UIView:0x754e3a0.width == UIWindow:0x71156e0.width - 20>",
"<NSAutoresizingMaskLayoutConstraint:0x74128b0 h=--- v=---
H:[UIWindow:0x71156e0(320)]>"
)
```

```
Will attempt to recover by breaking constraint  
<NSLayoutConstraint:0x754dac0 V:[UIView:0x754e510(230)]>
```

Break on objc_exception_throw to catch this in the debugger.
The methods in the `UIConstraintBasedLayoutDebugging` category on `UIView` listed
in `<UIKit/UIView.h>` may also be helpful.

Remember when I said that there must be enough constraints so that Auto Layout
can calculate the positions and sizes of all the views? Well, this is an example
where there are *too many* constraints. Whenever you get the error "Unable to
simultaneously satisfy constraints", it means that your constraints are conflicting
somewhere.

Let's look at those constraints again:



There are six constraints set on the green view, the four Spacing constraints you
saw earlier and the new Width and Height constraints that you have just set on it.
So where is the conflict?

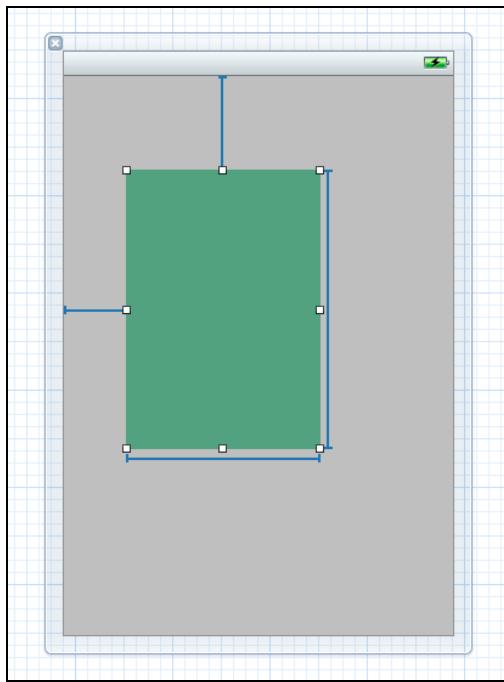
Well, in portrait mode there shouldn't be a problem because the math adds up. The
width of the superview is 320 points. If you add the lengths of the Horizontal Space
constraints and the Width of the view, then you should also end up at 320. The way
I have positioned the view, that is: $51 + 160 + 109 = 320$ indeed. Likewise, the
vertical constraints should add up to 460.

But when you rotate the device to landscape, the window (and therefore the
superview) is 480 points wide. That means $51 + 160 + 109 + ? = 480$. There are
160 extra points that need to go somewhere in that equation and Auto Layout
doesn't know where to get them. Likewise for the vertical axis.

The conflict here is that either the width of the view is fixed and one of the margins
must be flexible, or the margins are fixed and the width must be flexible. So one of

these constraints has to go. In the above example, you want the view to have the same width in both portrait and landscape, so the trailing Horizontal Space has got to go.

Remove the Horizontal Space at the right and the Vertical Space at the bottom. The nib should look like this:

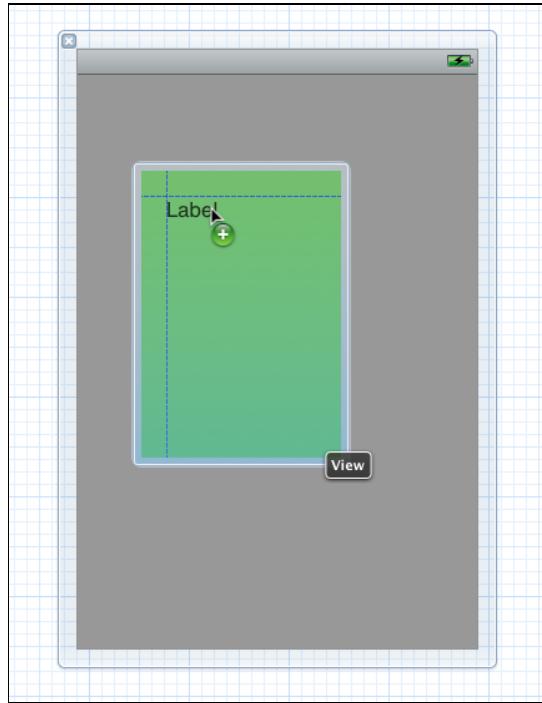


Now the view has just the right number of constraints to determine its size and position, no more, no less. Run the app and verify that the error message is gone and that the view stays the same size after rotating.

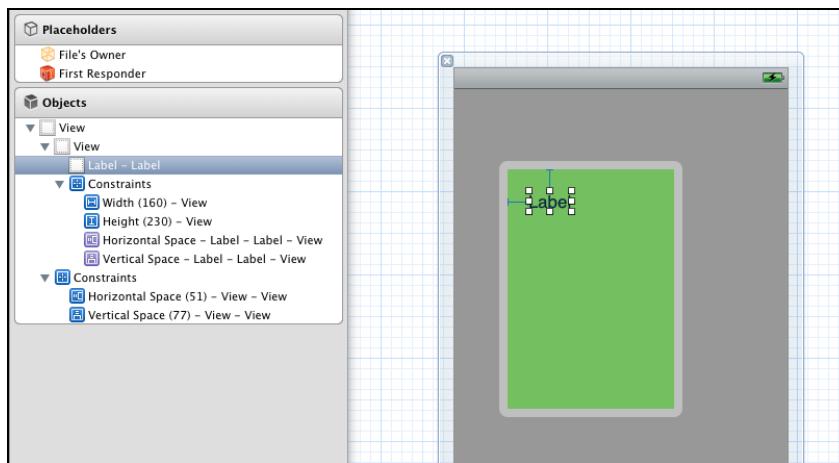
Note: Even though Interface Builder does its best to prevent you from making invalid layouts, it cannot perform miracles. At least Auto Layout spits out a detailed error message when something is wrong. You will learn more about analyzing these error messages and diagnosing layout problems in the next chapter.

Painting the portraits

Drag a label onto the green view. Notice that now the guides appear within that green view, because it will be the superview for the label.



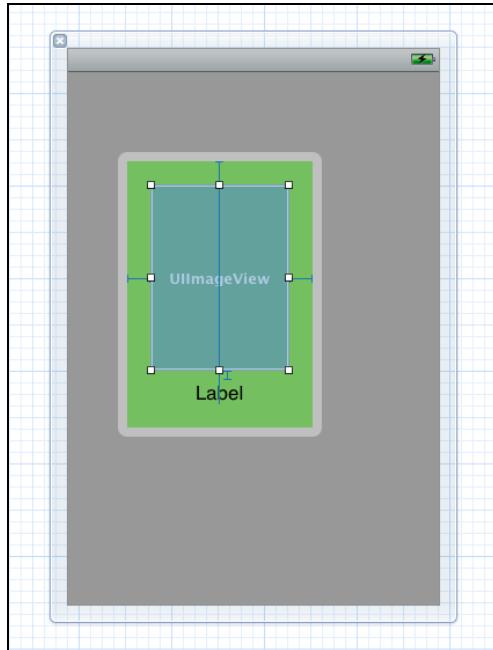
Position the label in the top-left corner against the guides. This will add two Space constraints to anchor the label in the top-left corner of the green view:



Notice that these two new Horizontal and Vertical Space constraints are listed under the green view's Constraints section, not in the main view.

Now move the green view around a bit. You'll see that only the constraints between the green view and its superview change, but those for the label don't. The label always stays put in the same place, relative to the green view.

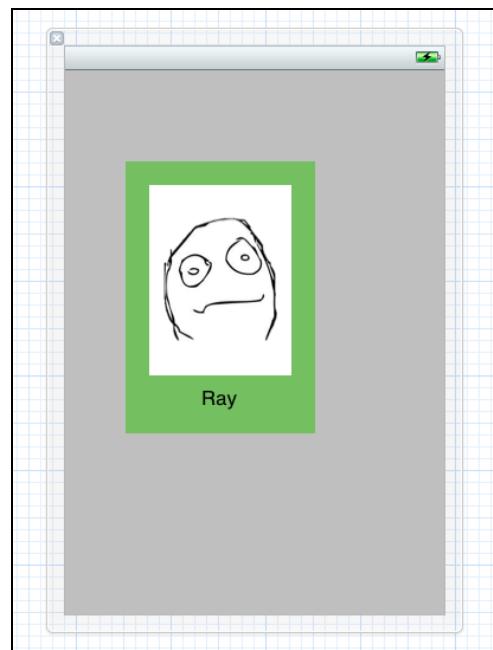
Select the label and place it against the bottom margin, horizontally centered. Then drag a new image view object on to the nib, and make the layout look like this:



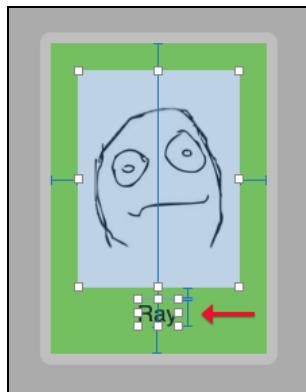
The image view is pinned to the top, left and right edges of its superview, but its bottom is connected to the top of the label with a standard spacing.

In the resources for this chapter you will find an **Images** folder – add this folder into your project. Set **Ray.png** as the image for the image view, change the image view's mode to Aspect Fit and set its background color to white. Change the label's text to say "Ray".

Your layout should now look like this:



Notice that Interface Builder has placed a Height constraint on the label now. This happened the moment you set the image on the image view.



Interface Builder tries to prevent what are known as *ambiguous layouts*. If neither the image view nor the label has a fixed height, then Auto Layout doesn't know how much to scale each if the height of the green view should change. (Interface Builder seems to ignore for now that the green view actually has a fixed Height constraint set on it.)

Let's say at some point in your app the green view becomes 100 points taller. How should Auto Layout distribute these new 100 points among the label and the image view? Does the image view become 100 points taller while the label stays the same size? Or does the label become taller while the image view stays the same? Do they both get 50 points extra, or is it split 25/75, 40/60, or in some other possible combination?

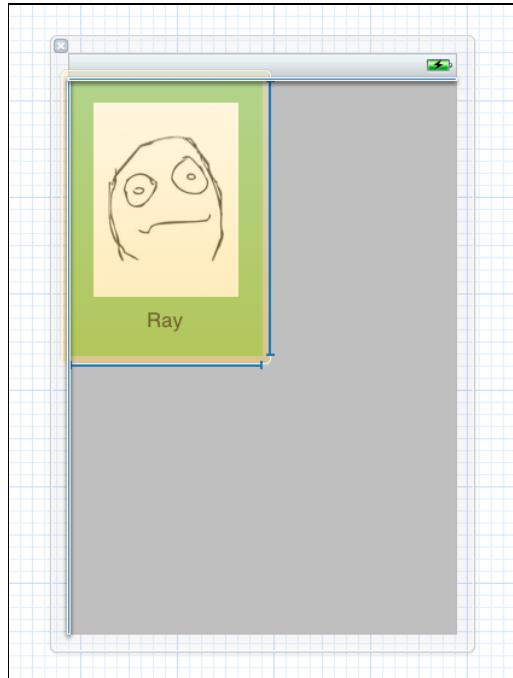
Auto Layout is not going to guess, so Interface Builder "fixes" this problem for us by giving the label a fixed height. It could also have given the image view a fixed height, but the label makes more sense.

For now, let's just live with the Height constraint on the label.

Note: The proper solution to this small layout problem is to change the "Content Compression Resistance Priority" of the label. You will learn more about that later on. If you can't wait, then go into the Size inspector for the label and set the vertical Content Compression Resistance Priority to 751. The Height constraint on the label should now disappear.

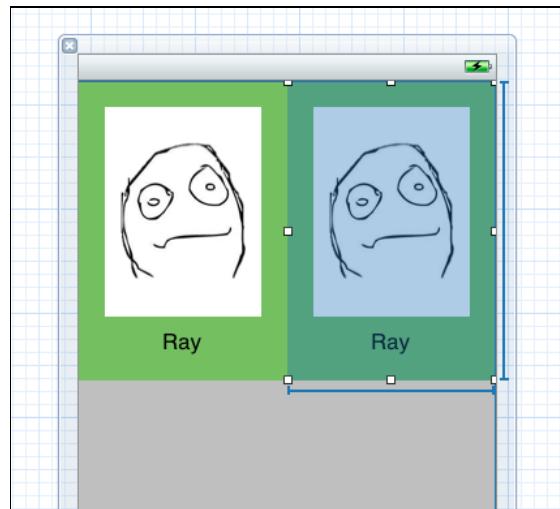
Adding the other heads

Move the green view onto the main view's top-left corner. Recall that the green view had Horizontal Space and Vertical Space constraints that determined its position in the parent view. It still has those, but they are now set to a value of 0 – they are represented by the thick blue lines (with white borders) at the top and left edges of the window:



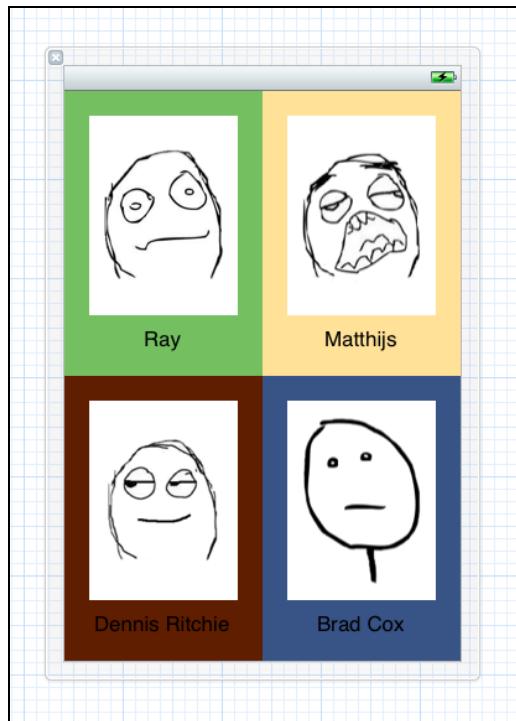
So even though the view sits completely in the corner, it still needs constraints to anchor it there. Think of these as margins with a value of 0.

Select the green view and tap **Cmd-D** to duplicate it. Move the duplicate into the top-right corner:



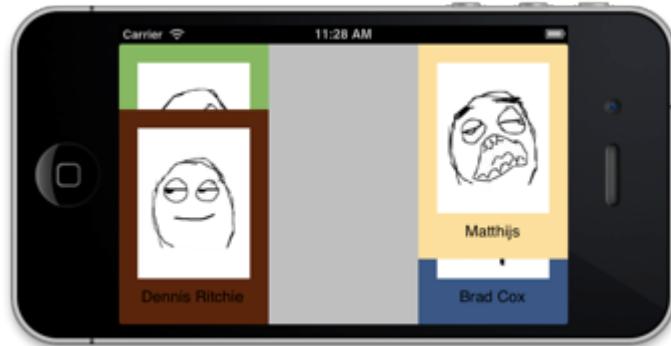
Duplicate two more times and put these copies in the bottom-left and bottom-right corners, respectively.

Change the screen design to the following:



Those are some good-looking programmers! :-)

Run the app. It looks good in portrait, but not so much in landscape:



It should be pretty obvious what went wrong: you've set a fixed width and height on the four brightly-colored container views, so they will always have those sizes, regardless of the size of their superview.

Select the Width (160) and Height (230) constraints from all four views and delete them. If you run the app now, you'll get something like this. Also not very good:



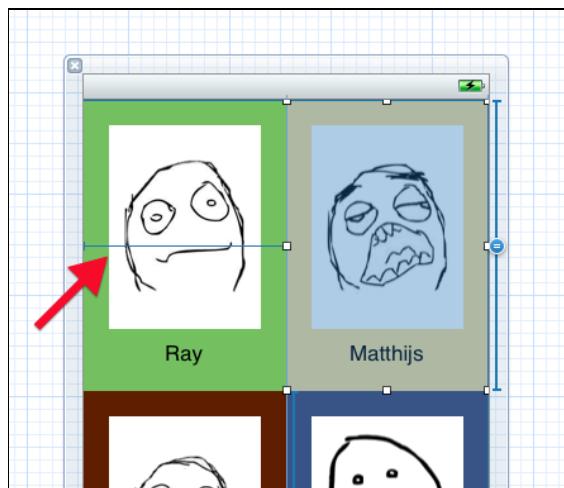
This looks very much like the problem we solved in the introduction, so if you think back to how we solved that, you'll recall that we gave the views equal widths and heights.

Select all four colored views and choose **Pin\Widths Equally**. Select the views again and choose **Pin\Heights Equally**.

Run the app again and rotate the device. Hmm... it still looks exactly the same as before. Why?

Well, if you look at the screenshot you'll see that all the views *do* have the same height, and they also appear to have the same width (the green and brown views are partially obscured by the yellow and blue ones), so our constraints are being met. It's just not the width and height that you want them to have. There must be other constraints that are getting in the way.

Sure enough, if you look at the constraints on these views, you'll see that they also have Horizontal and Vertical Space constraints that force them into place (look at list of constraints on the main view, not the four subviews):



What's worse, you can't even delete that constraint. Its T-bar is not bold and the constraint is not blue, so Interface Builder put it there in order to prevent a layout problem.

So why does it do that? Just saying that all four views must have equal sizes is not enough to determine what those sizes should actually be, because Auto Layout does not know how these four views are connected to each other. They appear side-by-side in the design, but there are no actual constraints between them. Auto Layout does not know that it needs to split the window width between the "Ray" and "Matthijs" boxes.

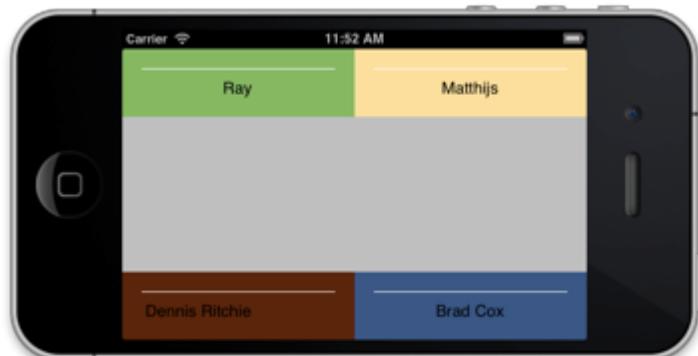
If Auto Layout can't figure this out by itself, you have to tell it.



Select the Ray and Matthijs boxes and choose **Pin\Horizontal Spacing**. Because the boxes are side-by-side, this adds a Horizontal Space constraint with size 0 between them, and that is enough to let Auto Layout know how these two views are related.

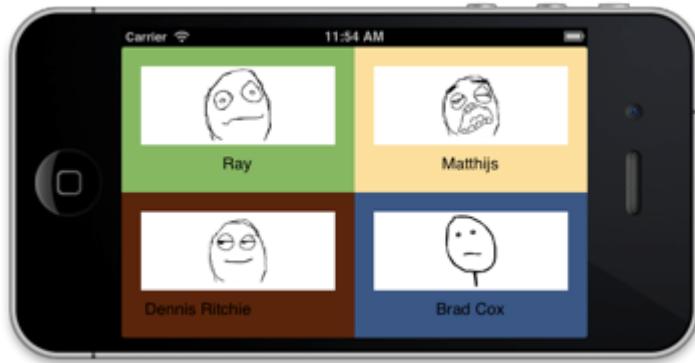
Important: Interface Builder does not automatically remove the leading Horizontal Space between the superview and the yellow box (the one from the screenshot above), but it did promote it to a user constraint (a fat bar). You can now delete this space. If you don't, you will get an "Unable to simultaneously satisfy constraints" error during runtime when you flip to landscape.

Run the app. It should now look like this:



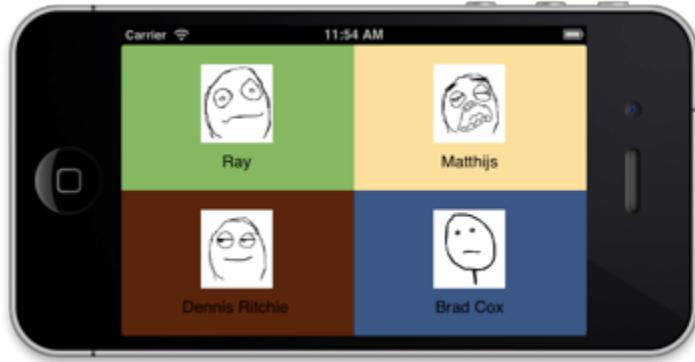
That looks a bit better already. The four boxes now have equal widths, but the heights are still wrong. The solution is similar: put a Vertical Space between the Ray and Dennis Ritchie boxes and remove the Vertical Space between the Dennis Ritchie box and the top of the window.

Run the app again, and this time it looks all right:



Notice that the “Dennis Ritchie” label is not centered below its image view. This originally happened to me when I typed that text into the label. The label was initially centered in the view, but Interface Builder decided it knew better and replaced that centering constraint with a Horizontal Space. If this happened to you, too, then select that label and choose **Align\Horizontal Center in Container** to fix it.

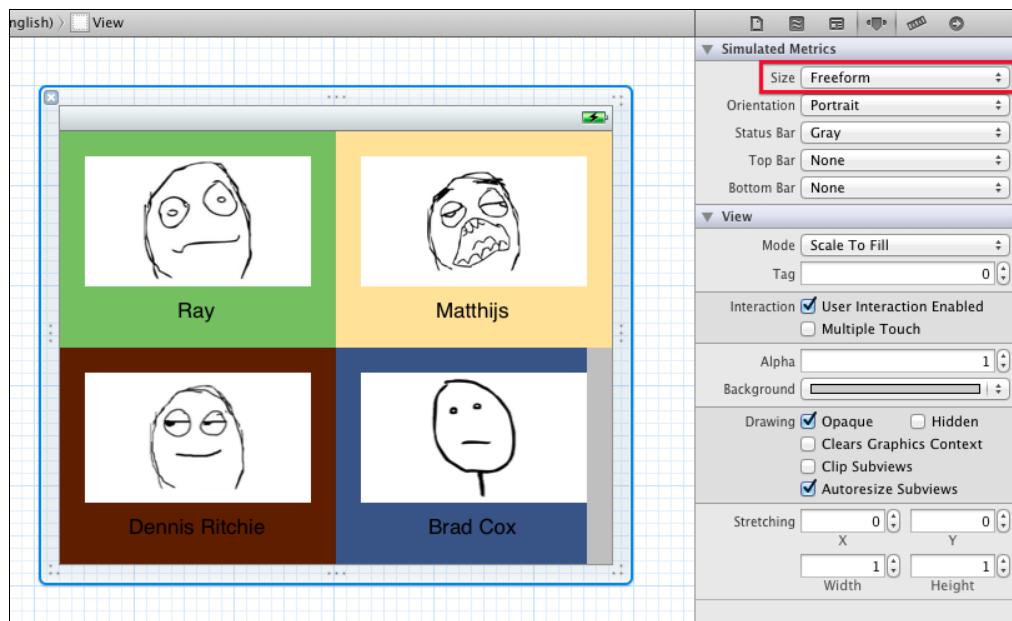
A quick note on the image views: they stretch out because you have not given them a fixed size. You may not know it, but that’s intentional on your part. ☺ The image views wouldn’t fit in landscape mode otherwise. However, if you want an image view to keep its original aspect ratio, then you’re out of luck. You cannot achieve the following effect using Interface Builder:



Unfortunately, Interface Builder does not currently provide a way to make constraints that keep the aspect ratio of a view intact. To do that, you need to create and set the constraints programmatically. You will learn how to do that in the next chapter.

Tip: You have seen that you can preview what the UI will look like in landscape by changing the Orientation setting under Simulated Metrics. You can also test the resizing behavior of your views directly in Interface Builder.

Select the main view. Under Simulated Metrics, set Size to Freeform. This adds resize handles around your nib that you can use to mold it into any shape you want. Auto Layout will recalculate the layout on-the-fly:



However, be careful with this. Sometimes Interface Builder will insert new constraints of its own when you're resizing, as it did here in the bottom-right corner (it added a Horizontal Space). It may also delete existing constraints when they fall outside of the nib bounds.

Note: If you've made it this far, congratulations – you now know what Auto Layout is all about, and have experimented with the basics! There's a lot left to learn – in the next section you'll use Auto Layout to create a more "real-world" detail screen. But if you just wanted to get a rough feel of Auto Layout for now, feel free to take a break at this point or move onto another chapter – you can always continue on from here later!

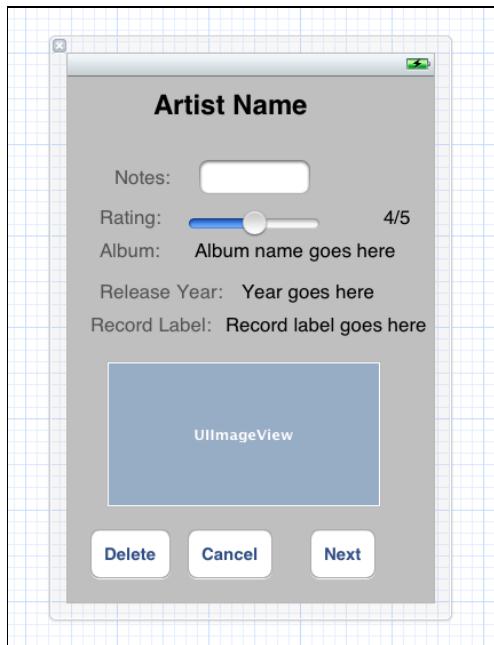
Details example

Let's move on to a more complex example. You are going to build a "detail screen" that you might see in a music player app. The screen that you will make shows the name of an artist and some album details:



Reuse the existing project again by deleting everything except for the main view from the nib file. Or if you can't bear to part with your Auto Layout masterpiece, create a new project from the Single View Application template and name it "ArtistDetails". No storyboards, but enable Automatic Reference Counting.

Open the nib file and drag a bunch of labels and buttons onto it, roughly as follows. Don't worry too much about the constraints yet; you will make it look good step-by-step over the course of this section.



The Artist Name label has a System Bold font, size 24. After typing into the label, press **Cmd =** (or **Editor\Size to Fit Content**) to resize it to its optimal size. The labels on the left have text color "Dark Gray Color".

When designing screens with Auto Layout, you no longer worry about placing your controls at specific coordinates, but you do think about the relationships between the controls.

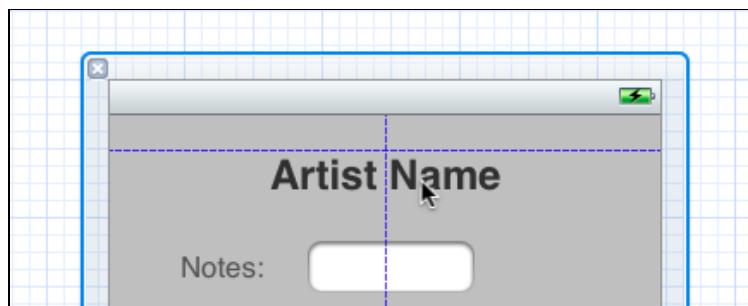
So how would you think about the relationships between these components? By just dropping the controls onto the canvas the way you did, Interface Builder will have given them all sorts of constraints that may or may not be suitable.

The Artist Name label

Let's start with the Artist Name label at the top. This will contain the name of the currently selected artist, which is why it is bold and has a bigger font. This label will probably look best if it is centered horizontally and offset from the top by the standard margin.

That last sentence is exactly how we're going to describe this label to Auto Layout: "horizontally centered in the window, standard margin from the top."

Select the Artist Name label. Drag it so that it aligns with the top margin and the horizontal center:



Alternatively, you can choose **Align\Horizontal Center in Container**, and change the size of its Vertical Space constraint to Standard (in the Attributes inspector). But in this case, dragging the label is just as easy.

This design may satisfy you, but let's put it to the test by simulating what might happen in a real app.

Add an outlet property for the label to **ViewController.m**:

```
@interface ViewController ()  
@property (nonatomic, weak) IBOutlet UILabel *artistNameLabel;  
@end
```

Also add an action method:

```
- (IBAction)nextButtonTapped:(id)sender
{
    static NSArray *artists;
    if (artists == nil)
    {
        artists = @[ @"Thelonious Monk", @"Miles Davis", @"Louis
Jordan & His Tympany Five", @"Charlie 'Bird' Parker", @"Chet
Baker" ];
    }

    static int index = 0;

    self.artistNameLabel.text = artists[index % 5];

    index++;
}
```

This method cycles through a list of artist names and sets them on the label, one by one.

Go back to the nib file and connect the Artist Name label to the `artistName` outlet (Ctrl-drag from File's Owner to the label).

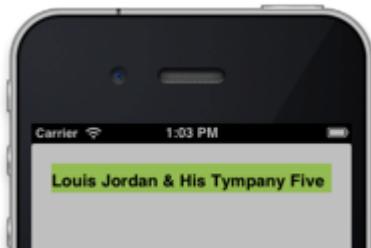
Also connect the Next button to `nextButtonTapped:` (Ctrl-drag from the button to File's Owner).

Before you run the app, give the Artist Name label a non-transparent background color. That makes it easier to see exactly how big the label is.

Run the app and tap the Next button a few times. You can see that the label changes size with each new artist name, but always stays centered. Unfortunately, some names are too big, causing the label to go outside the screen bounds:



That is a bit sloppy. It would be better if the label respected the standard margins at the side of the screen and truncated the text a bit better, or even resized the font to fit all the text, like this:

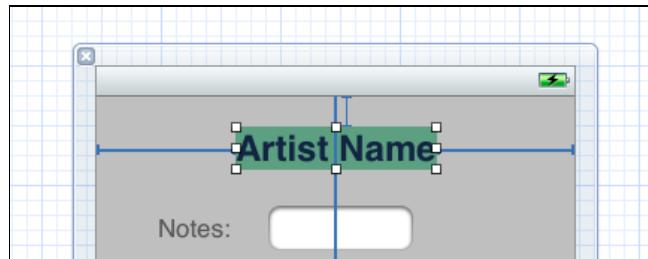


There are a couple of solutions you could try:

15. Give the label a smaller font. That would work for large titles, but doesn't look very nice for artists with smaller names.
16. Give the label a fixed width. If the standard margin is 20 points, then the width would be $320 - 20 - 20 = 280$ points. But that doesn't look very good if the app needs to work in landscape mode as well, because the label won't make optimal use of the extra space.
17. Give the label a leading Horizontal Space on the left and a trailing Horizontal Space on the right.

The first two options have obvious drawbacks, so let's go for number three.

Select the Artist Name label and choose **Pin\Leading Space to Superview**. Repeat and choose **Pin\Trailing Space to Superview**. Now the label has a new Horizontal Space constraint on either side:

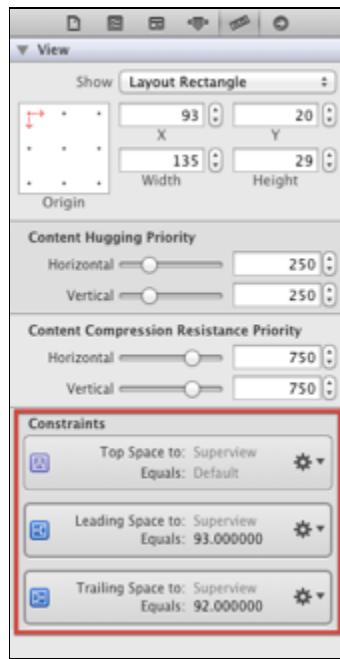


You can remove the Center X Alignment constraint; it is now superfluous.

Of course, if you were to run the app now, the Artist Name label would never grow any bigger than this, so you have to make the Horizontal Spaces smaller first. Select each Horizontal Space constraint and set it to the Standard size.

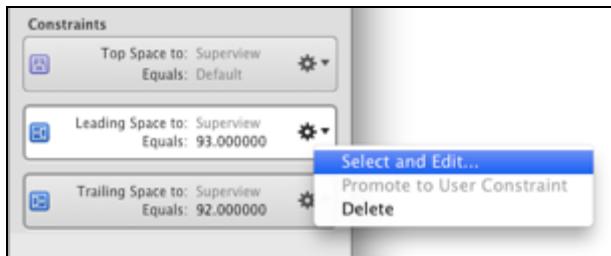
Because there are quite a few controls in the screen already, there are also a lot of constraints. Finding the ones you need can be tricky. Fortunately, Interface Builder makes it easy to see which constraints belong to a particular view.

With the Artist Name label selected, open the Size inspector. At the bottom you will find the constraints that are set on this label:



As you hover over each of these constraints, you will see them light up in the nib. That makes it easy to find the constraints you want.

Hover over the Leading Space to: Superview constraint and click the arrow next to the gear icon. From the menu, choose Select and Edit...

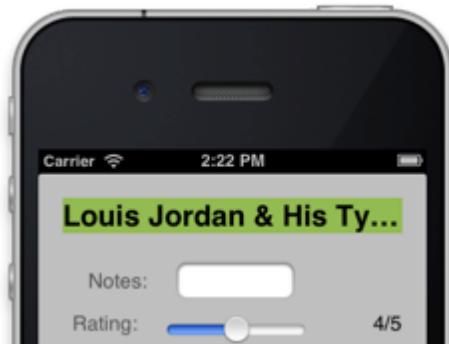


This selects the constraint and activates its Attributes inspector pane. Simply check the Standard box. Do the same thing for the Trailing Space constraint.

Note: Interface Builder sometimes refers to Horizontal Space constraints as Leading Space or Trailing Space, depending on which edge of the superview these constraints connect to. Why “leading” and “trailing” instead of “left” and “right”? This was done to make internationalization easier. After all, not all languages read from left-to-right. Hebrew and Arabic are two common right-to-left languages.

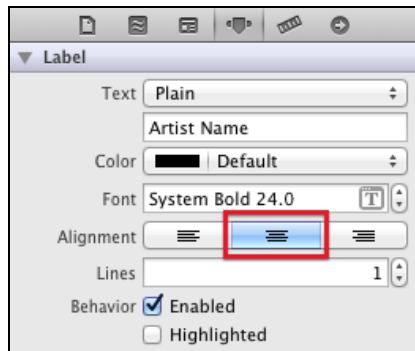
You will read more about this in the section on localization in the next chapter, but “leading” automatically gets translated to “on the right” for such right-to-left languages, and “trailing” becomes “on the left.” If you were to hard-code it as “left” and “right,” then these constraints wouldn’t flip sides when running the app on a Hebrew or Arabic iPhone.

Run the app to see what the label looks like now:



That’s better. The artist name no longer goes outside the margins, and when the text gets too wide, it is properly truncated. However, you may have noticed that the text in the label is no longer centered. This has nothing to do with the fact that you just removed the Center X Alignment constraint. That constraint centers the label as a whole, but has no impact on the text inside the label.

One solution is to set the label’s Alignment property to centered:



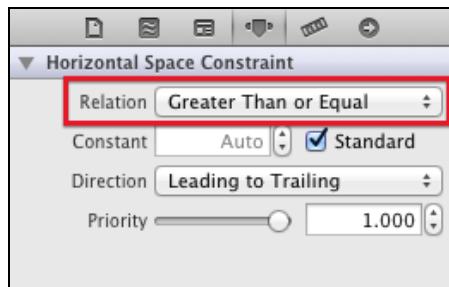
Now the label always takes up the entire width of the view (minus the margins), but it centers the text inside that space. It's a valid solution, and that's how you probably would have done it before Auto Layout, but I want to show you a different way to do it that is more powerful.

Inequalities and goodwill towards all men

So far, all of the constraints you used have had fixed sizes. Horizontal or Vertical Spaces were always X points big. It's also possible to make constraints that say, "I am at least X points, but I could be bigger," or the opposite, "I am at most X points, but I could be smaller." This lets you build some flexibility into your layouts.

Such constraints are sometimes called *inequalities* because they are "greater than or equal" or "less than or equal" to a specific value. You use them to set minimum or maximum values on constraints.

Select the Leading Space between the window edge and the artist name label. That's the one on the left. Go into the Attributes inspector and change the Relation attribute to Greater Than or Equal:



Keep the Standard field checked. This Horizontal Space constraint will now be at least 20 points wide (the standard margin), but it can grow larger if possible. You can also think of Greater Than or Equal meaning, "not smaller than" – this constraint will never shrink any smaller than the standard margin.

Do the same thing for the Trailing Space to the right of the label. Set its Relation to Greater Than or Equal as well.

If you look now at the label's constraints, you should see something like this:

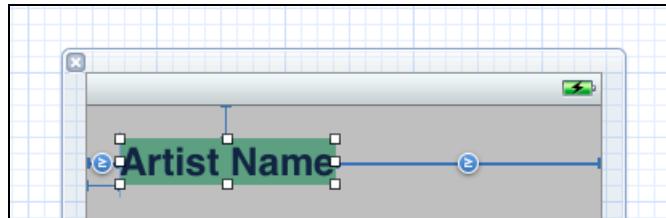


Besides the new \geq constraints, which represent the changes you have just made, the label now also has regular Horizontal Space constraints. That's not entirely what you intended.

Why did Interface Builder insert these extra constraints? You should know by now that when Interface Builder starts adding in new constraints, it means the ones you specified are by themselves not enough to place and size the view.

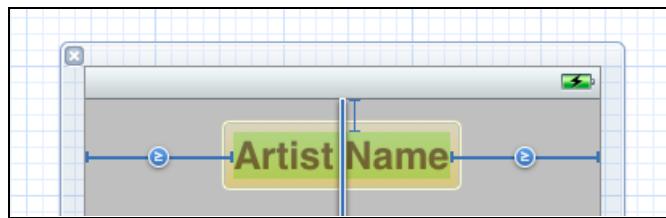
In this case, just saying that the space on the left and the right sides of the label can be anything, but not smaller than 20 points, is not enough to determine the size of the label.

What if you do Size to Fit Content, would that solve this? After all, a label has an intrinsic content size – it knows how big it should be from the text that has been set on it. Try it out. Select the Artist Name label and tap **Cmd =**. Interface Builder will do something like this:



That's closer, but not quite there. There is still an extra Horizontal Space on the left. Auto Layout still does not have enough information to position this label with just the Greater Than or Equals constraints.

The solution is to center the label again. Select the label and choose **Align\Horizontal Center in Container**:



Now Auto Layout finally has enough information:

- It knows how big the label is, through the intrinsic content size.
- It knows where to put the label: always horizontally centered.

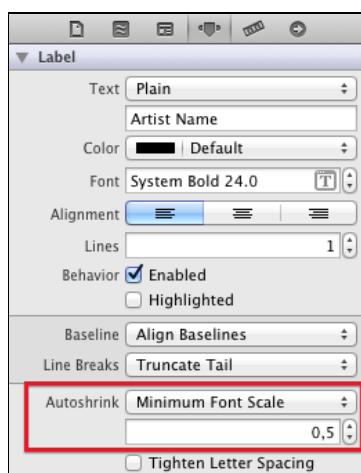
- It knows that the label cannot grow beyond the standard margins; that is what the \geq constraints are for.

Do you see how the constraints express exactly what I have just described?

Run the app to verify that the artist name is now always centered.

Of course, you can do a bit better still. If the text is too long it gets truncated. You can fix that with the standard label attributes (Line Breaks and Autoshrink). That didn't change with Auto Layout, except that iOS 6 now adds a Minimum Font Scale property.

Select the Artist Name label and set its Autoshrink property to "Minimum Font Scale":



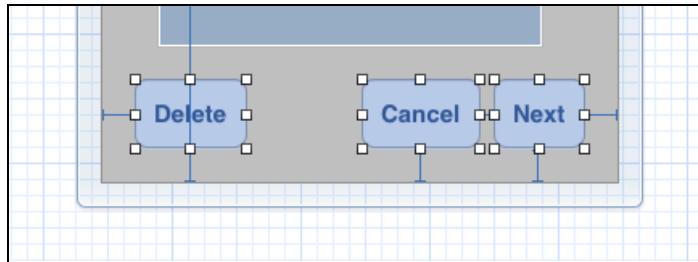
Now run the app and the long text "Louis Jordan & His Tympany Five" should shrink to fit the maximum label size. Because the label uses its intrinsic content size (there is no fixed width constraint on it), and it is centered with a constraint, it will always grow or shrink to be the correct size. Pretty cool.



Push the buttons

Let's now focus on the bottom of the screen, where the buttons live. I just plunked them down rather haphazardly, but it would look nicer if Delete was all the way over to the left and Cancel and Next were all the way to the right.

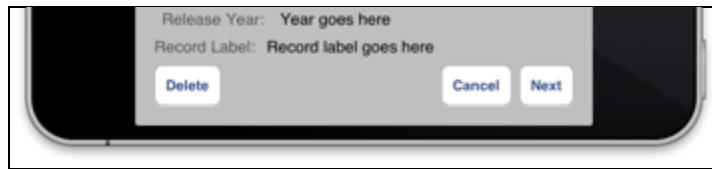
Use **Size to Fit Content** to give the buttons their ideal size, and attach the Cancel button to the left of Next:



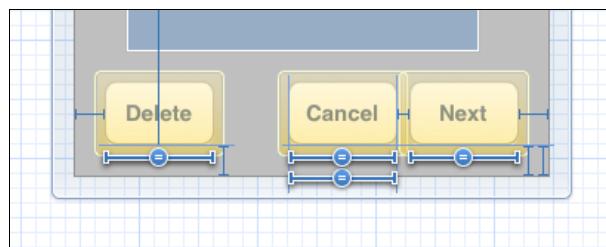
It might be a bit hard to see, but there is a standard-sized Horizontal Space between Cancel and Next, because I want to express the intention that the Cancel button always sits attached to the left of the Next button.

As you can also see in the screenshot, when I moved the buttons around, the Delete button got center-aligned with some other control in the screen (one of the labels). You may get similar things in your own nib when you do this, depending on where you dropped all the labels. It's just Interface Builder trying to be helpful, but as you will find out shortly, that often turns into a case of Interface Builder getting in the way. For now, simply ignore this additional constraint.

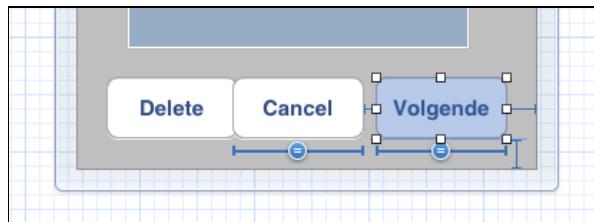
Run the app and flip to landscape. The buttons do what you'd expect:



It's not too bad, but something upsets the visual designer inside me. The Next button is a bit smaller than the other two, but I think it would look nicer if these three buttons are always the same size. Select the three buttons and do **Pin\Widths Equally**:



Problem solved? Well, at this point it's smart to think ahead a bit. You probably want to translate your app to other languages (you should!). In some languages the text on the buttons might be bigger than in English. For example, replace the text on the Next label with "Volgende", which is the Dutch translation:



Note: There are two ways to change a title of a button – you can type the title in directly by double-clicking the button, or you can change the button's Title attribute in the Attributes inspector. The latter method will not resize the button automatically. So if you want the button to resize automatically, you have to type the button title in directly by double-clicking the button.

Whoops, now the Delete and Cancel buttons overlap! The Equal Widths constraint here works to your disadvantage. It would be good to add a limitation to the button layout:

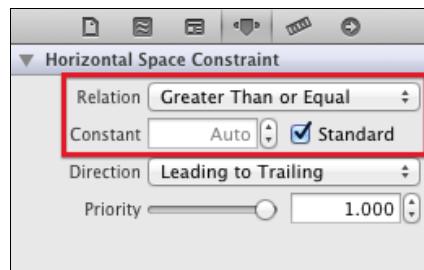
"Give them all equal widths... except when there isn't any room for that."

Undo to put the "Next" text back so you have some room to work with.

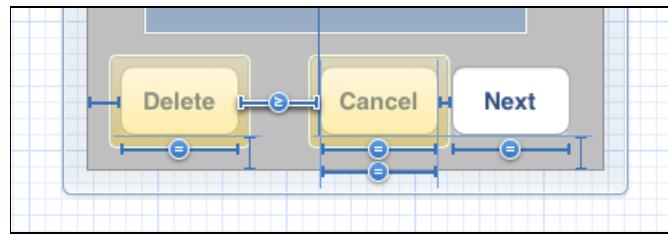
Note: If this messes up the widths of the buttons, then select the button that has both Equal Widths constraints and press **Cmd =** to restore them to their optimal sizes. In my screenshots that is the Cancel button, but in your project that could also be Delete or Next; Interface Builder seems to pick it at random.

Add a Horizontal Space between the Delete and Cancel buttons. You should be able to do this on your own by now, but in case you forgot how, select both buttons and choose **Pin\Horizontal Spacing**.

With the new Horizontal Space constraint selected, go into the Attributes inspector and change Relation to Greater Than or Equal. Check the Standard box:



The buttons now look like:

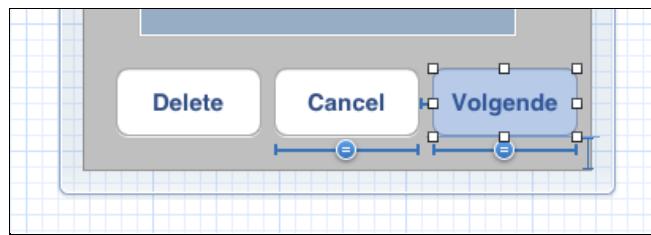


This means that there will always be at least 8 points of space (the standard margin between controls) between the Delete and Cancel buttons, but if there is room for more space then that's OK too.

Note: The left side of the Cancel button in the above screenshot now has a Leading Alignment with some other view (the blue line extending upwards). Again, this is Interface Builder trying to be helpful.

It just so happens that the left edge of the Cancel button aligns with the left edge of one of your labels, and Interface Builder assumes you always want these two to be aligned. You don't, but this will be fixed later when you rearrange the labels. If you get something similar, then just ignore it for now.

Let's try renaming the Next button again. Replace its text with "Volgende". Sure enough, the Delete and Cancel buttons no longer overlap, but the Next button gets pushed into the right margin:



There simply isn't enough room to satisfy all these constraints. Something has to give and Interface Builder has removed the Trailing Space that used to be between the Next button's right edge and the border of the screen.

It gets worse if you also rename the Delete button to "Verwijderen":



Obviously, this isn't working quite right yet. But recall that you wanted to express the intention, "give all the buttons equal width, except when there isn't room."

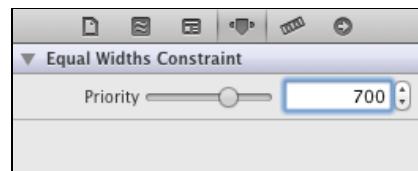
Well, here you have a case where there isn't enough room. Fortunately, Auto Layout has a solution for this: *priorities*.



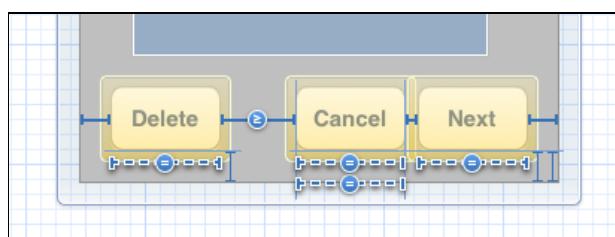
You can tell Auto Layout which constraints should be satisfied first, and which ones are nice to have, but optional. Each constraint has a priority associated with it. The priority is a number between 1 and 1,000. A priority of 1,000 means that this constraint should always be satisfied, but anything lower than 1,000 makes it optional. There are several predefined priority levels that have special meaning, but we'll get to that later.

First, undo the most recent changes so that the Delete and Next buttons are in English again. (Don't go too far back! Make sure there is still a \geq constraint between Delete and Cancel and that all three buttons still have the Equal Widths constraint set on them.)

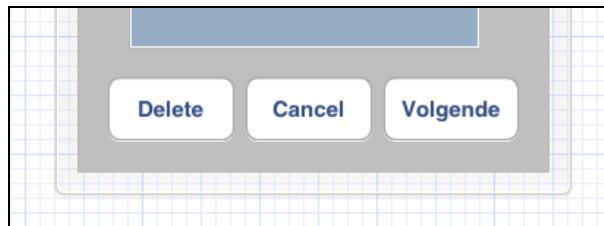
Now select the Equal Width constraint between Delete and Cancel and set its priority to 700. You do this in the Attributes inspector for the constraint. Do the same for the Equal Width constraint between Cancel and Next:



When a constraint has a priority lower than 1,000 – in other words, when it is optional – it shows up as a dotted line:

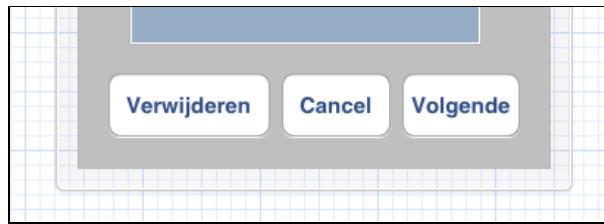


Now rename Next to "Volgende" again. Unlike before, the button doesn't get pushed into the margin:



You can verify with the Size inspector that the Delete and Cancel buttons now have the same width (86 points), but the Volgende button is slightly wider. It needed to be wider in order to fit its text, but Delete and Cancel still had room to shrink. Even though Auto Layout was not able to satisfy the Equal Widths constraint between Cancel and Volgende – which is OK because it's optional – it did still have enough room to satisfy the Equal Widths constraint between Delete and Cancel.

Rename Delete to "Verwijderen" again. The Cancel button shrinks even further, so all buttons have different sizes and none of the Equal Widths constraints are active anymore. But thanks to the other constraints, the buttons never overlap:



Using priorities and inequalities, you can create these sorts of relationships between the buttons to make them resizable without overlapping. This is essential when translating your apps to different languages.

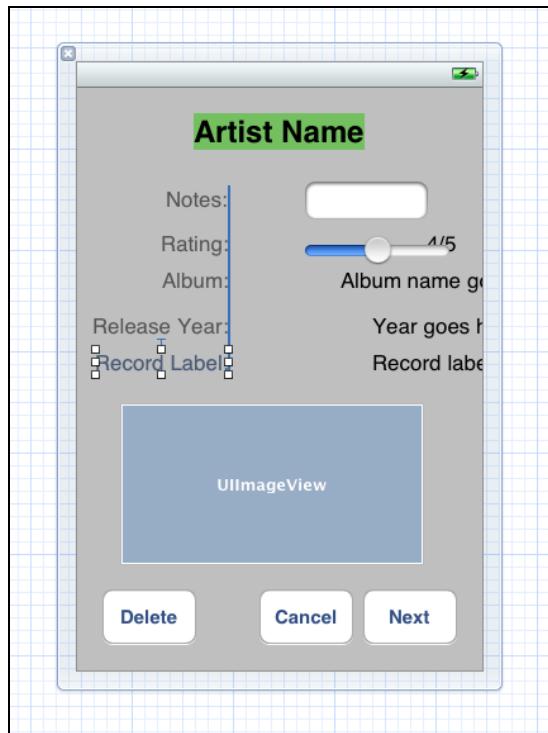
We'll get into this in detail in the next chapter, but using Auto Layout, you can provide a single nib or storyboard file for all languages, while separate strings files provide the translations. So it is essential that your buttons and labels can grow and shrink to accommodate different titles.

Undo until you have the Next text back, unless you speak Dutch. ☺

Label me beautiful

You have fixed up the Artist Name label at the top and the buttons at the bottom, but everything in between is still a bit of a mess. First let's concentrate on the labels on the left.

Just to make sure they don't get in the way, move the text field, the slider and the other labels over to the right. Then select the gray labels and choose **Align\Right Edges**. It might look something like this:



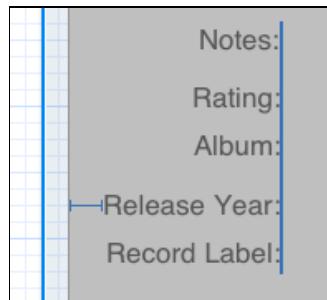
Note: If you don't see the labels lining up correctly, it could be that you have some other automatic constraints on the labels that are messing things up for you. The best way to sort things is to move each of the labels manually so that they don't line up against each other, and to also set "Size to Fit" on each label individually. This usually identifies any other constraints and will help you set things up so that you can get the layout similar to what's shown above.

The five labels on the left have Trailing Alignment constraints between them. Remember that "trailing" means as much as "right", except in right-to-left languages such as Arabic or Hebrew, where it gets flipped to mean "left".

Also note that a constraint is always between two views at most, so if you right-align five labels, there are also five Trailing Alignment constraints.

In the above screenshot, I don't really like the way some of the labels sit inside the left margin, such as Release Year and Record Label. The problem is that if you decide to move the Release Year label by itself, Interface Builder will break the Trailing Alignment constraints that you just added. It's better to select all five labels and move them as a group. Interface Builder will keep the constraints between these labels intact, although it will obviously modify any constraints with their superview.

Hold down **Cmd** and select the five labels. Then use the arrow keys to move them as a group until the Release Year label snaps with the left margin. If you were to look at just how these labels are now anchored, it would look like this:



There is a Horizontal Spacing between the Release Year label and the edge of the screen that determines the X-position of that label. The X-positions of the other four labels are derived from the Trailing Alignment that they all share. (I left out the constraints for the vertical positions from this picture; we'll get into those shortly.)

These constraints work fine for an English version of the app, because here the text "Release Year:" is the longest label of the five. But what if the app is translated to another language where this assumption no longer holds true? Let's try changing the text of this label at runtime to see what effects this will have on the layout.

Add a new outlet property to the class extension in **ViewController.m** (@interface section):

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

Insert the following code before the last line (`index++`) of `nextButtonTapped:`:

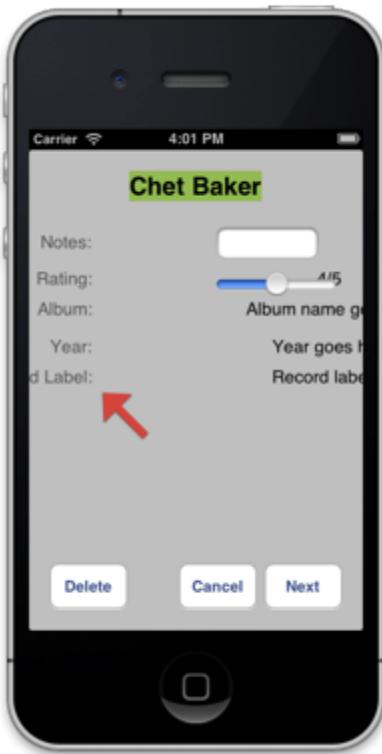
```
static NSArray *texts;
if (texts == nil)
{
    texts = @[@"Year:", @"Very Long Label Text:", @"Release
Year:"];
}

self.releaseYearLabel.text = texts[index % 3];
```

This makes it so every time you tap Next, the text will change on the Release Year label.

Back in **ViewController.xib**, connect the Release Year label to the `releaseYearLabel` outlet.

Run the app and tap the Next button a few times. Because the positions of the other labels are relative to each other, and only Release Year is anchored to the side of the screen, the other labels fall off the screen when the text becomes shorter:



If this sort of situation should appear in your own app, then it's a good idea to add some more constraints to prevent this from happening.

There are two solutions, one simple and one complicated.

The simple solution

You might want to make a copy of the project folder or an Xcode snapshot at this point, so you can easily go back later to try the second solution.

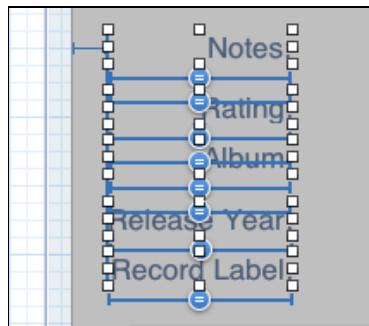
Note: To make an Xcode snapshot, choose **File\Create Snapshot...** and follow the instructions.

Select the five labels. Choose **Align\Left Edges**. Now the labels are no longer right-aligned.

Select the five labels again. Choose **Pin\Widths Equally**.

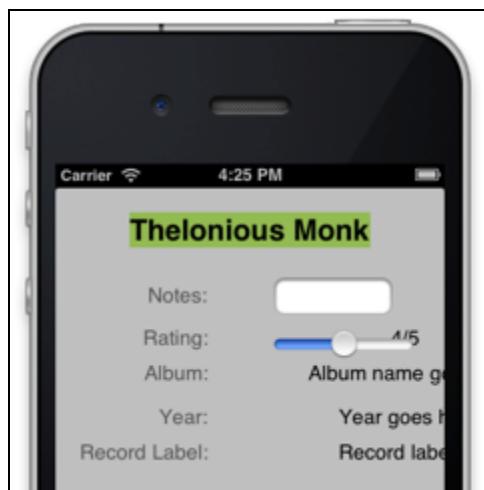
Select the labels again. In the Attributes inspector, set the Alignment property to "right aligned".

This works, but it does put a lot of constraints between the labels, which makes it harder to see what is going on:

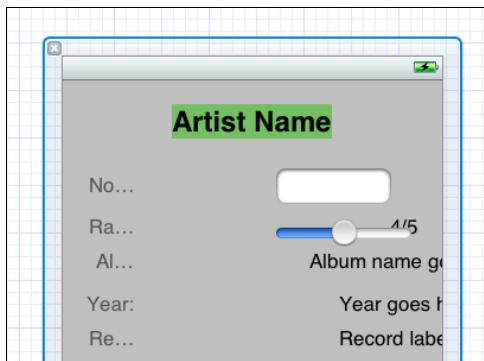


It doesn't really matter now which label has the Horizontal Space to the edge of the screen (in my screenshot it is the Notes label). All labels have the same size anyway, and their left edges are all aligned.

Run the app. Now when the text in the Release Year label is small, all the other labels still fit in the screen:

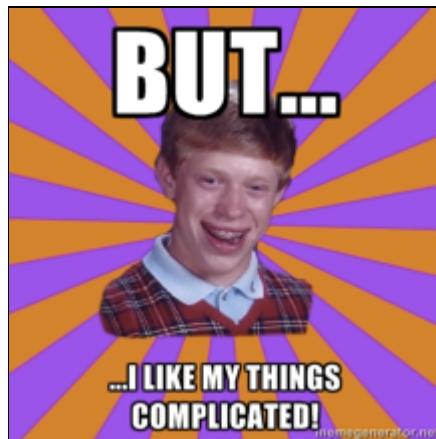


By the way, if you go back to the nib and give the Release Year label a shorter text, you might see this (or, depending on the other Auto Layout constraints set up, you might not. But remember that here too, you need to double-click the label and modify the label directly in order for it to auto-size):



That is different from what happens if you set the label text during runtime. Interface Builder recognizes that the other four labels should have the same width as this one, sees that this one has become smaller, and resizes the other ones to the same, smaller, width. That is obviously not what you wanted here.

When this happens, you can easily fix it by finding the largest label (Record Label at the bottom) and doing **Size to Fit Content**. Don't resize by hand because that will mess up your carefully-constructed constraints. Always use **Size to Fit Content** (or **Cmd =**) if you can.



The complicated solution

Even though the simple solution should suffice in most cases, I want to point out that constraints also let you solve this problem in a different way. Besides, why do something the easy way when you can also make it difficult for yourself? ☺

Restore the project to how it was before the simple solution. If you made an Xcode snapshot, you can simply restore it by choosing **File\Restore Snapshot...**. You can also press Undo a couple of times.

The five labels are now right-aligned again, and there is a Leading Space that keeps the Release Year label away from the margin.

Add a new **Leading Space to Superview** on the "Record Label" label. As always, do this from the **Pin** menu. In the Attributes inspector for the new constraint, choose Relation: Greater Than or Equal, and check the Standard box.



Run the app and tap the Next button a few times. It works OK if the text in the Release Year label is longer than any of the others, but not so good when it is

shorter. Still, it is better than before, because the other labels do not get pushed out of the screen anymore:



Ideally, the "Year" label would be pushed to the right so that it right-aligns with the others. However, this doesn't happen because it has a Leading Space with a fixed size keeping it in position. So there are two constraints fighting for attention here – the Trailing Alignment and the Leading Space – and the space wins.

Maybe you're thinking, "What if I turn this Leading Space into a Greater Than or Equal constraint? That way this space can grow when necessary." If you did think that, then you're getting the hang of this!

Select the Leading Space on the left side of the Release Year label and change its Relation to Greater Than or Equal. Run the app. Hmm, still no go. Take a closer look at the constraints for that label:



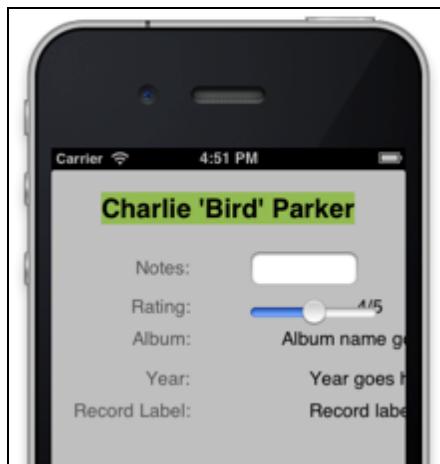
Wait a minute... where did that second Leading Space come from? There was only one and you turned that into a Greater Than or Equal. Well, when you did that, Interface Builder automatically inserted another one with a fixed width.

By now you should know that when Interface Builder does this, it is because the constraints you've given are not sufficient by themselves. Just a \geq constraint is not enough. There always needs to be some sort of Equals constraint as well.

Fortunately, you can take advantage of the fact that Auto Layout works with priorities. Select that second Leading Space (the one Interface Builder inserted), and set its priority to 200 in the Attributes inspector. Because this constraint is now optional, it shows up as a dotted line:



Run the app and tap Next. Now the smaller "Year:" label is properly right-aligned with the rest of them:



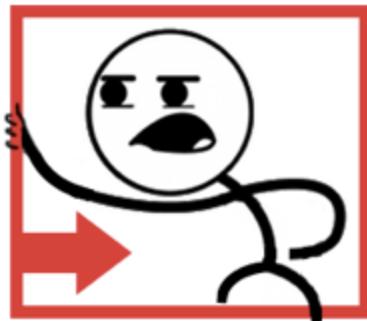
Cool. But why did you have to set the priority to 200 – what is so special about this number? To be honest, it doesn't really matter what you set the priority to, as long as its value is lower than the *Content Hugging priority* of the label.

Say what?

Each view that uses its intrinsic content size has two additional priorities associated with it – the Content Hugging priority and the Content Compression Resistance priority – that determine at which point the view will allow itself to grow or shrink.

Remember that views such as labels and buttons know what their ideal size is, and they will always try to be that size. But that is not always possible in the larger scheme of things. That label or button does not know how important its content is in relationship to the other views around it. You can tweak the control's resizing behavior using the Content Hugging and Content Compression Resistance properties.

The important one here is Content Hugging. This determines how much the control resists becoming bigger – you can also think of it as "expansion resistance." Imagine a little guy sitting inside the label who is pulling hard to keep the label from growing.



To allow the label to grow in this case, the priority of the fixed Leading Space has to be lower than the Content Hugging priority, which by default has the value 250 (although you can change this for each view in the Size inspector).

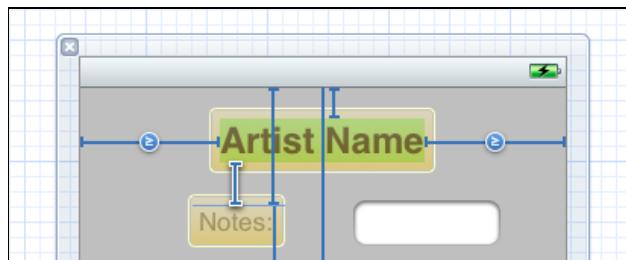
To see the difference, change the priority of the Leading Space to 300, to make it higher than the label's Content Hugging priority. Run the app again and tap Next. Now the Leading Space weighs heavier in Auto Layout's calculations than the Content Hugging, and the label will hold onto its size (the little guy in the label wins). But if the priority is lower than 250, then the label is free to expand.

Does this blow your mind? It wouldn't surprise me. Just play with it until it makes sense. Keep in mind that Interface Builder likes to sneak in all kinds of extra constraints "to keep the peace," so if something should work but it doesn't, double-check to make sure there are no constraints that shouldn't be there. And if there are, tapping **Size to Fit Content** will often get rid of them.

Stack 'em up

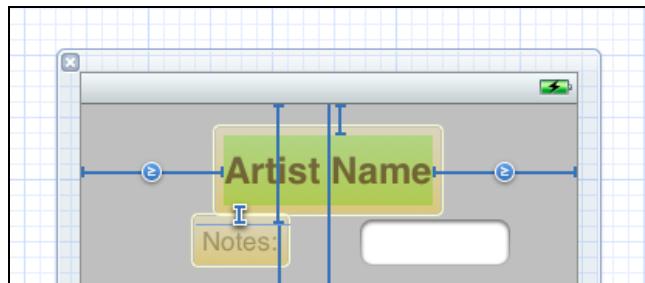
The labels look all right horizontally, but in the vertical direction it's still a bit uneven. Your intent is that the labels be stacked on top of each other, with the top label hanging off the page title (the Artist Name label).

Select the Notes and Artist Name labels. Choose **Pin\Vertical Spacing**. This adds a new spacing constraint between the two labels:



The way my labels are arranged, it's a space of 28 points. That's too much, so select the new constraint and set its Constant to 12.

Hmm, that didn't quite do what you may have expected:



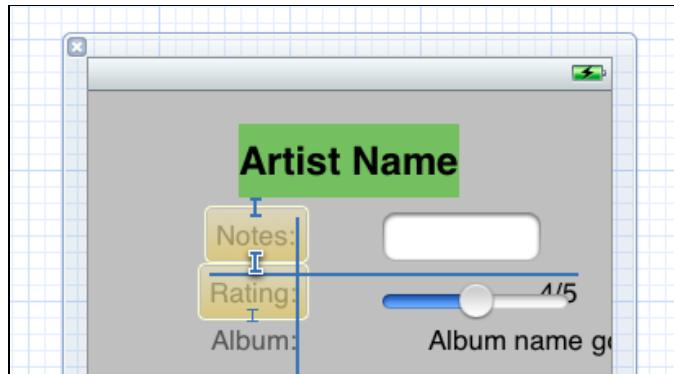
It certainly made the Vertical Space constraint smaller, but did so by stretching the Artist Name label. Why does this happen? If you look closely at the Notes label, you'll see that there is also a Vertical Space that goes all the way up to the top edge of the screen. (Alternatively, it may have aligned the notes label with the text field next to the label, which then has a Vertical Space locking it at the top of the screen). In order to satisfy this constraint, the only thing Interface Builder could do when you resized the other vertical space was stretch the Artist Name label.

Choose Undo to restore the Artist Name label to its proper size. Then delete that other Vertical Space that keeps the Notes label pinned to the top of the screen. You can delete it without problems because the other constraint is enough to determine the vertical position of the Notes label.

Now you can safely change the Vertical Space between the two labels to 12 points again.

Also add a Vertical Space between the Notes and Rating labels. Make it 14 points high. Depending on exactly where you dropped the labels when you dragged them onto the nib, there may already be a vertical space between these two labels. If that is the case, then simply change its Constant value in the Attributes inspector.

When I did this last step, the following thing happened for me:



The Artist Name label stretched again! The Rating label, instead of going up to stick to the Notes label, dragged everything down instead. (Of course, this might not happen to you, depending on how all your other constraints are set up.)



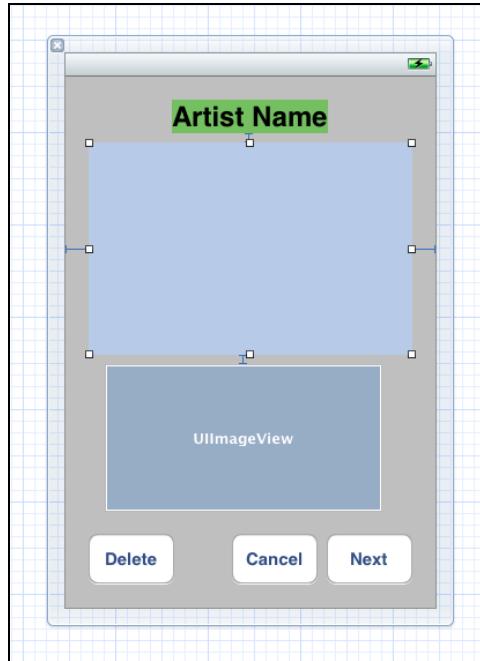
A big clue as to why this happened is the fat blue bar between the Rating label and the "4/5" label at the right. These two labels are top-aligned. I didn't do that myself, but Interface Builder thought it was a good idea. Because this particular constraint exists, any changes you make to other constraints will also take this one into account. Because this constraint could not be broken, the Artist Name and Notes labels adjusted instead.

In this particular case, I was able to remove that Top Alignment constraint and try again, but as you can tell, Interface Builder has a tendency to get in the way, especially if there are many views inside your layout. It will often attempt to make relationships between views that you don't want to be related at all.

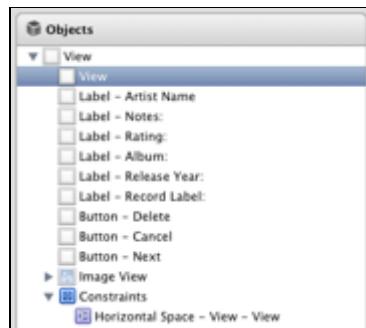
Here's a trick you can use to avoid such problems if your layouts get moderately complex: use subviews to organize your screens into logical sections. You will now make such a container subview to hold the labels.

First, remove the text field, the slider, and all the labels on the right. You need some room to work with, and these things are getting in the way.

Drag a new view object from the Object Library onto the nib and position it between the Artist Name label and the image view:



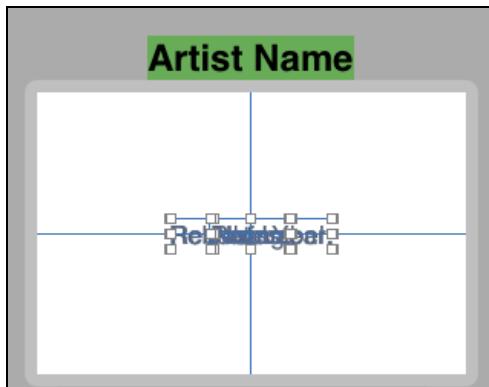
Currently it obscures the labels, so in the Document Outline drag this view all the way to the top:



Now select the Notes, Rating, Album, Release Year, and Record Label labels in the Document Outline and drag them onto the container subview:



Unfortunately, this makes a jumble of things – it places all the labels in the center of the container subview – so you'll have to remake the constraints on those labels:



First, give the container view a specific height. If you don't do this, Interface Builder may spontaneously decide to resize it when you change the constraints on the labels, and you want to avoid that. Select the container view and choose **Pin\Height**.

Drag the labels roughly into the same positions as before:



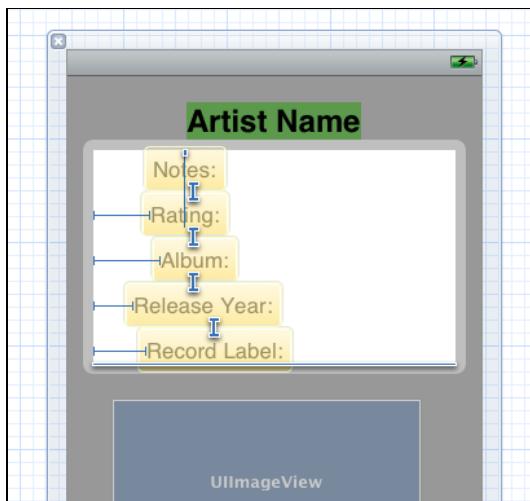
Select the Notes label. If it already has a Vertical Space going to the edge of the container view, then set its Constant to 4. If it doesn't, then from the **Pin** menu choose **Top Space to Superview** and change the Constant of this new constraint to 4.

Select both the Rating label and the Notes label. Choose **Pin\Vertical Spacing**. If there already was a Vertical Space from Rating up to the container view's top edge, then delete it. Set the Vertical Space between the two buttons to 14 points.

Do the same thing for the Album, Release Year and Record Label labels. Each time, make a Vertical Space between the label and the label above it. Delete any other Vertical Space that exists. Then change the distance to 14.

When you're done, resize the container view so that it snaps against the bottom of the Record Label label. This will add a new Vertical Space constraint at the bottom with size 0, but that's OK.

The vertical constraints should now look something like this:

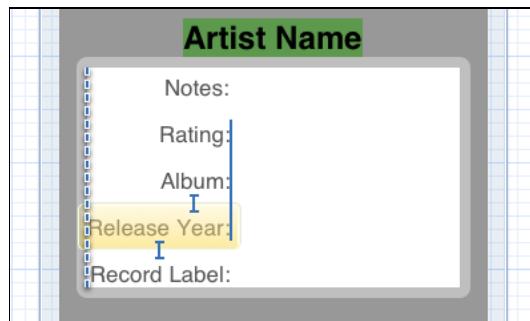


Next you'll repeat the steps from the "complicated solution" section to horizontally align the labels.

Select all five labels. Choose **Align\Right Edges**.

Again select all five labels. Use the arrow keys to move the entire group until the Release Year label snaps against the left edge of the container view. This inserts a new Horizontal Space constraint with size 0.

Set the priority of that constraint to 200. This makes it a dotted line:

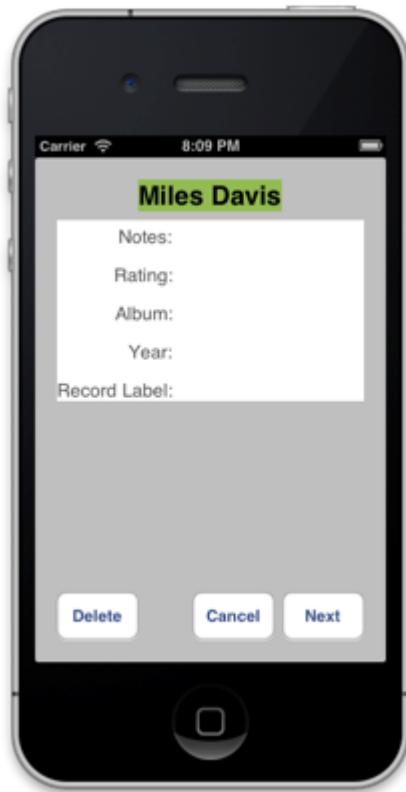


Because this horizontal space is 0, the constraint actually shows up as a vertical line across the entire height of the container subview.

Select the Release Year label and choose **Pin\Leading Space to Superview**. In the Attribute inspector, change Relation to Greater Than or Equal. Leave Constant set to 0.

Select the Record Label button and choose **Pin\Leading Space to Superview**. In the Attribute inspector, change Relation to Greater Than or Equal. The Constant is not 0 because this label is slightly shorter, so set it to 0 by hand.

Run the app and tap the Next button. It should properly align the labels on the left, even if the text in the Release Year label is short or long:



OK, great. Get ready to add the rest of the labels back in.

Note: The advantage of putting the labels inside their own container view is that Interface Builder will only draw the alignment guides to the other views in that same container view. That makes it impossible to inadvertently align one of the labels with one of the bottom buttons, for example.

It is still possible to make constraints between views that have different superviews, but you will have to do that in code, not in Interface Builder. More about that in the next chapter.

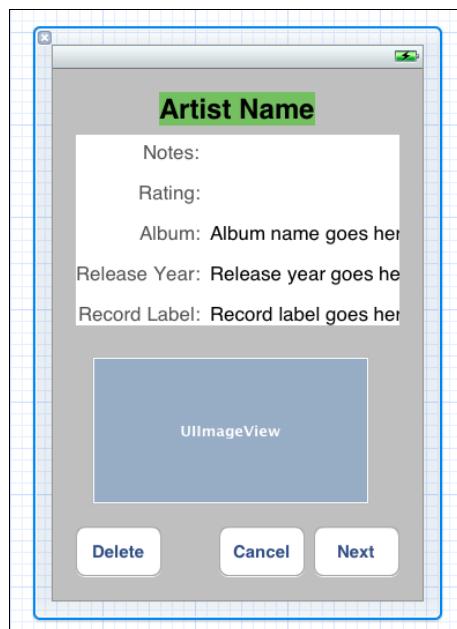
Right-hand labels

Drag a new Label into the container subview and snap it in place next to Album:



Change the text in the label to "Album name goes here".

Repeat this for the labels for Release Year and Record Label. You should end up with something like this:



Try putting some real content into the Album value label, so you can test how this works at runtime. It's one thing to look at the design in Interface Builder, but it is entirely another thing when you start putting real data into your labels.

Add an outlet property to the class extension in **ViewController.m**:

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

Add the following code before the last line in `nextButtonTapped`:

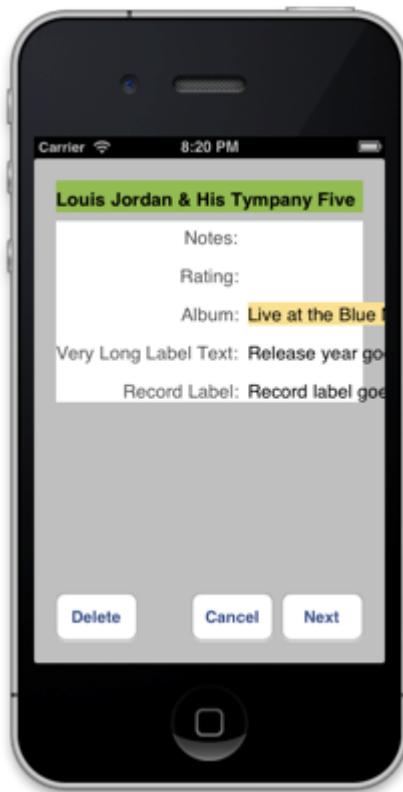
```
static NSArray *albums;
```

```
if (albums == nil)
{
    albums = @[ @"The Complete Riverside Recordings",
                @"Live at the Blue Note" ];
}

self.albumValueLabel.text = albums[index % 2];
```

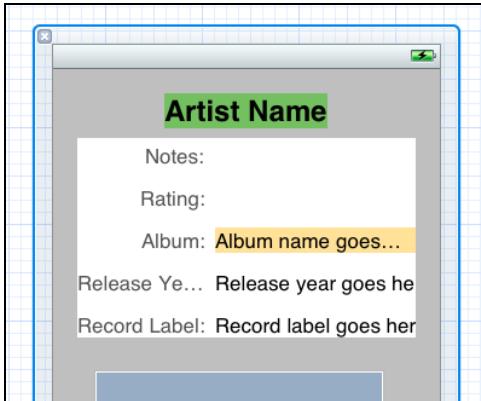
Back in **ViewController.xib**, connect the Album value label to the `albumValueLabel` outlet. Also give this label a non-white background color, so you can see exactly how big it is.

Run the app and tap the Next button a few times. Hmm, that is strange. In Interface Builder the labels are too long, but there they don't overflow beyond the margins. When you run the app, they do:



Obviously that won't do. If the previous 90 pages of this chapter made any impression at all, you know what to do already: add a Trailing Space between the label and the container view's right edge. To save some time, you'll just do it for the Album here. The process is exactly the same for the other two labels.

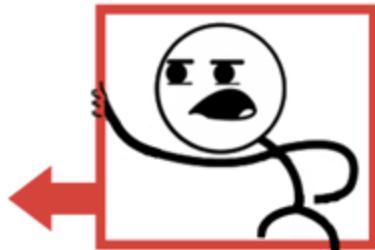
Select the Album value label and choose **Pin\Trailing Space to Superview**. This adds a Horizontal Space constraint of size -12, because the label in Interface Builder is too big. Change it to 0. Hmm, that messes up the layout on the left:



Because there isn't enough room, Interface Builder pushes the Release Year label to the side. That is not what you want. The labels on the left should never be truncated. Run the app and you will see the same thing, but worse. Auto Layout needs to find room for the text that is too long, and it tries to squeeze some extra space out of the labels on the left.

A little while ago I mentioned two properties that are important for labels, buttons, and other views that have an intrinsic content size: the Content Hugging priority and the Content Compression Resistance priority. The Hugging priority determines how strongly the label wants to prevent becoming larger. This is a little guy inside the label that is pulling very hard to keep the label the same size.

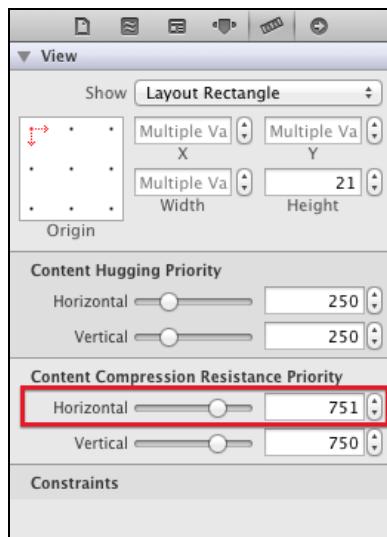
This time, however, you're going to use the Compression Resistance. This property determines how strongly the label doesn't want to become any smaller. Imagine another little guy sitting inside the label, pushing very hard against the edges to resist any force pushing in on the label.



In our case, the labels on the left and the labels on the right have the same Compression Resistance priority, the default value of 750. When Auto Layout needs more room, it will try to compress both labels. Because both labels resist with the same amount of force, Auto Layout will also shrink them by about the same amount. However, if you tell the labels on the left to resist harder, then Auto Layout won't make them any smaller.

Undo until the point where the new Trailing Space is -12 points again.

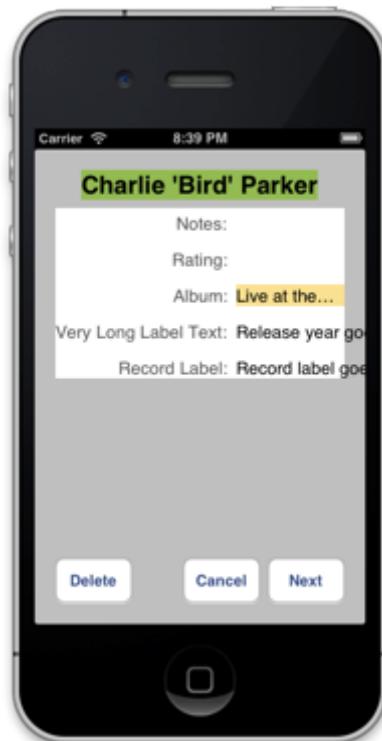
Select all five labels on the left. In the Size inspector, set Content Compression Resistance Priority, Horizontal to 751:



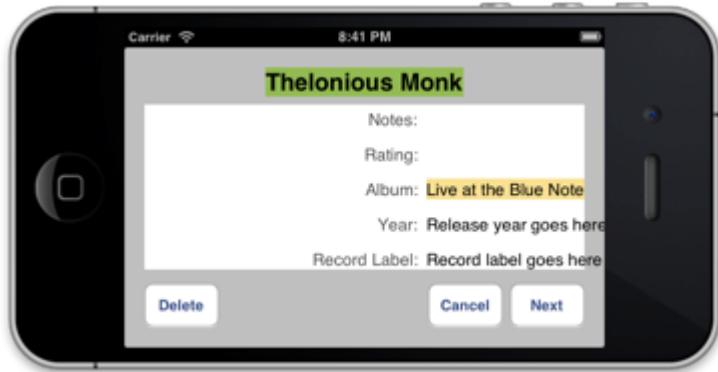
It doesn't matter what the exact value is, as long as it is higher than the priority value of the labels on the right.

Now select the Horizontal Space to the right of the Album value label and change the Constant back from -12 to 0. The labels on the left don't give in this time; they keep their sizes.

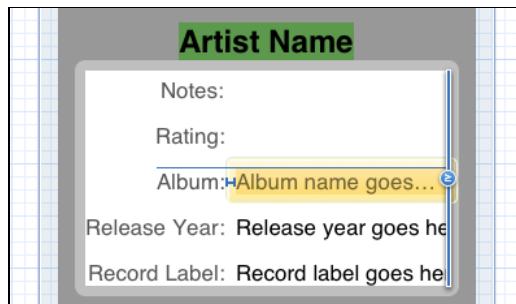
Run the app and tap Next to toggle through the different label sizes.



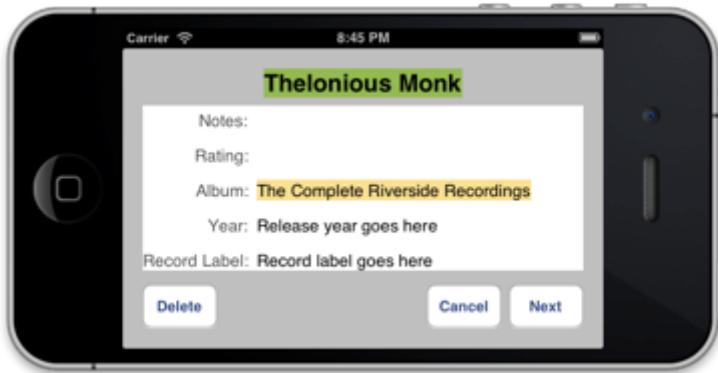
The Album value label now properly truncates while the labels on the left are rock-solid. There is one small problem in landscape, though:



Because the left-hand labels are attached to the right-hand labels, everything now shifts to the right to accommodate the album name. To fix this, make the Horizontal Space that anchors the label to the right edge of the container view a \geq constraint, rather than a fixed value:



Now when you run the app, the labels do exactly what you want:



It takes a bit of work to make this happen, but you end up with a layout that is very flexible. The names of the labels on the left can change, and everything in the layout shifts accordingly. All of this without writing a single line of code!

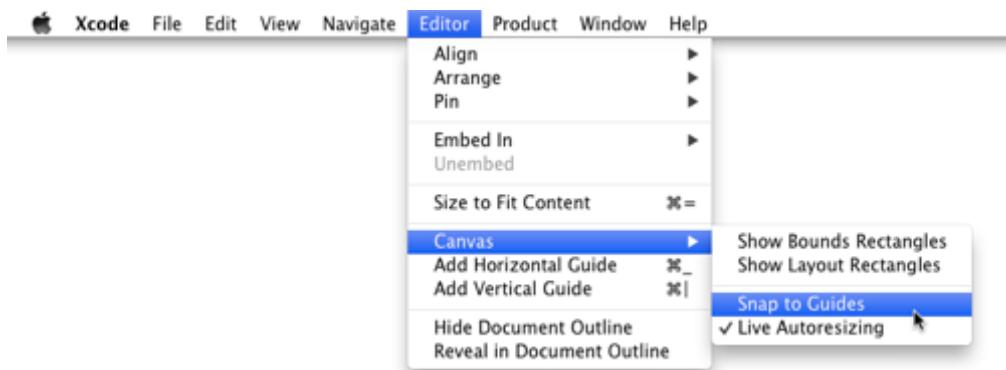
You just need to get a sense for which constraints to use where, and how to combine them with the different priorities available.

The other stuff

The screen design also had a slider and a text field.

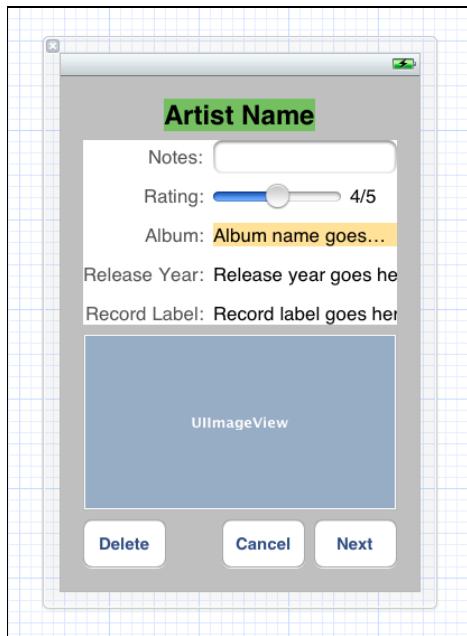
Drag the slider back onto the container subview. Because it is starting to get crowded again, Interface Builder may start to add all kinds of unwanted constraints, aligning it with this and that. You already avoided some of that mess by putting the labels into a container subview, so that they were insulated from the buttons and everything else, but you cannot escape it entirely.

But there is a trick you can use that will stop Interface Builder from interfering in most cases. From the **Editor** menu, turn off **Canvas\Snap to Guides**:

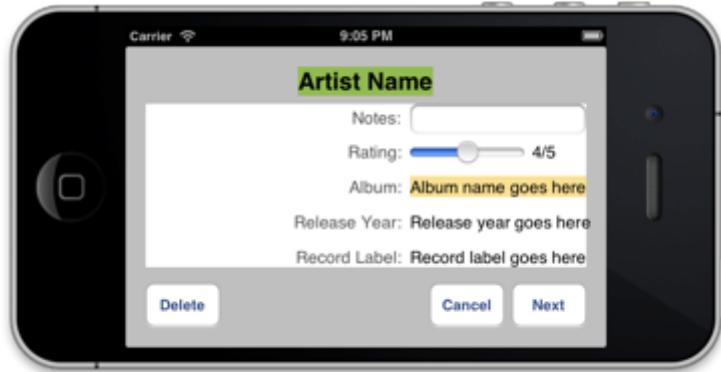


Now when you drag new items onto the canvas, Interface Builder will no longer show any guides, and as a result will not snap the items to any existing controls. But if you happen to drop it in the right (or wrong) place, then Interface Builder may still choose to align it with a control that you did not intend. After all, Interface Builder will always try to find the best set of constraints that describe a valid layout.

Finish the layout of the screen by adding the text field, and by aligning the image view properly with the views that surround it:



Run the app and flip to landscape. What the...?! Everything is aligned with the right edge of the screen again (at least for me – I hope you are more fortunate):



It worked fine until you added the text field, so what does the text field do to cause this? Take a look at the constraints and you will see that the text field is anchored to the right edge of the container view using a Trailing Space of size 0. That is what pulls everything towards the right edge.

Note: If you don't have the same results as above, examine the text field's constraints. If there are pinned leading and trailing spaces and there is no pinned width, you should be set. Otherwise, modify the constraints to get it to work as above.

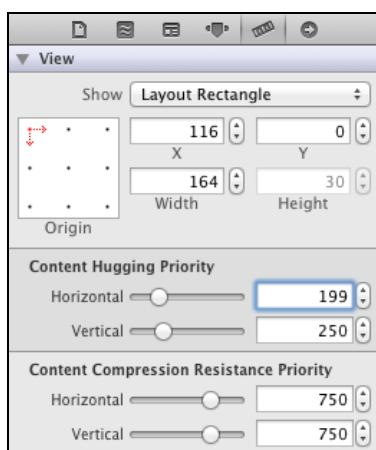
You may think, "Ah ha! I will simply make this Trailing Space a Greater Than or Equals, just like we did with the Album label." Go ahead, try it out.

It was a good idea, but unfortunately, it doesn't work. The text field simply doesn't want to enlarge to fill up the remaining space to the left.

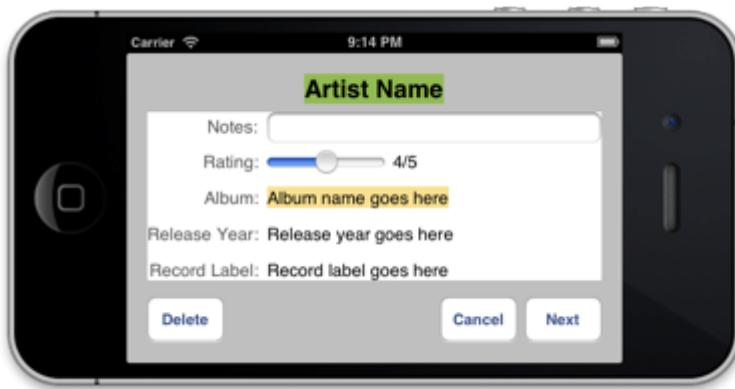
This should ring a bell. Controls have a certain resistance to wanting to expand – the Content Hugging priority. Currently that is set to the default value of 250 on the text field. But there is also a Horizontal Space way over on the left, attached to the Release Year label, that has a priority of 200 (from a while back, remember?).

Because the Content Hugging priority of the text field is higher than 200, the text field's resistance to expansion takes precedence, and Auto Layout decides to move everything over to the right.

The solution is simple: set the horizontal Content Hugging priority of the text field lower than 200, for example to 199:



Now it will work. Run the app and flip to landscape to see:

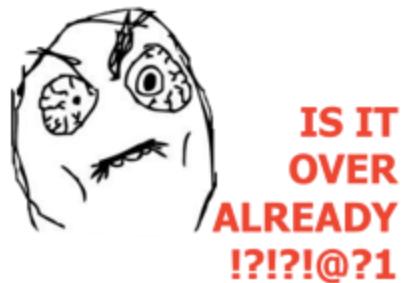


As you have seen, designing with Auto Layout constraints takes a bit more effort than just dumping controls onto your Interface Builder canvas, but it gives you a lot more flexibility in return. Unfortunately, Interface Builder doesn't always understand what you're trying to do – and with a lot of controls on the screen it is easy to mess up your existing constraints.

Note: Well done, you have made it to another huge milestone in this chapter! You now have hands-on experience creating a practical and flexible detail screen layout, and have learned a lot about working with tricky Auto Layout situations.

There's still more aspects of Auto Layout to cover, but this is also a great natural breaking point in this chapter if want to take a break and return back later. I'll still be here! ☺

But just in case you can't get enough of this, let's do one more for the road, this time with storyboards!



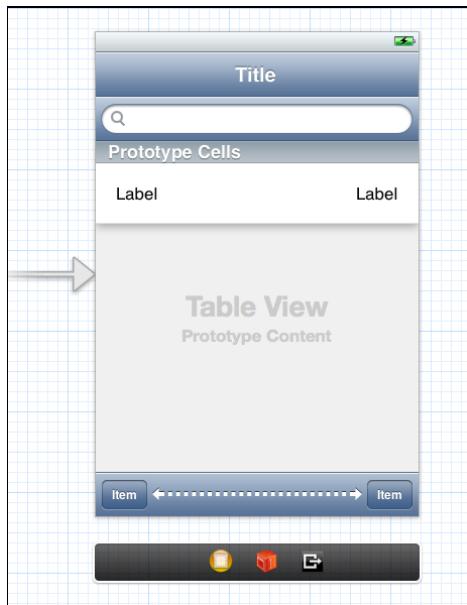
Storyboard example

Auto Layout works just as well with storyboards as it does with nibs. In this section, you'll use Auto Layout in a storyboard, and in the process will learn some new and subtle aspects of Auto Layout such as layout priorities, ambiguous layout, and more.

Note: This chapter only demonstrates how to make layouts for iPhone with Auto Layout, but of course it works for iPad too. It's just bigger. ☺

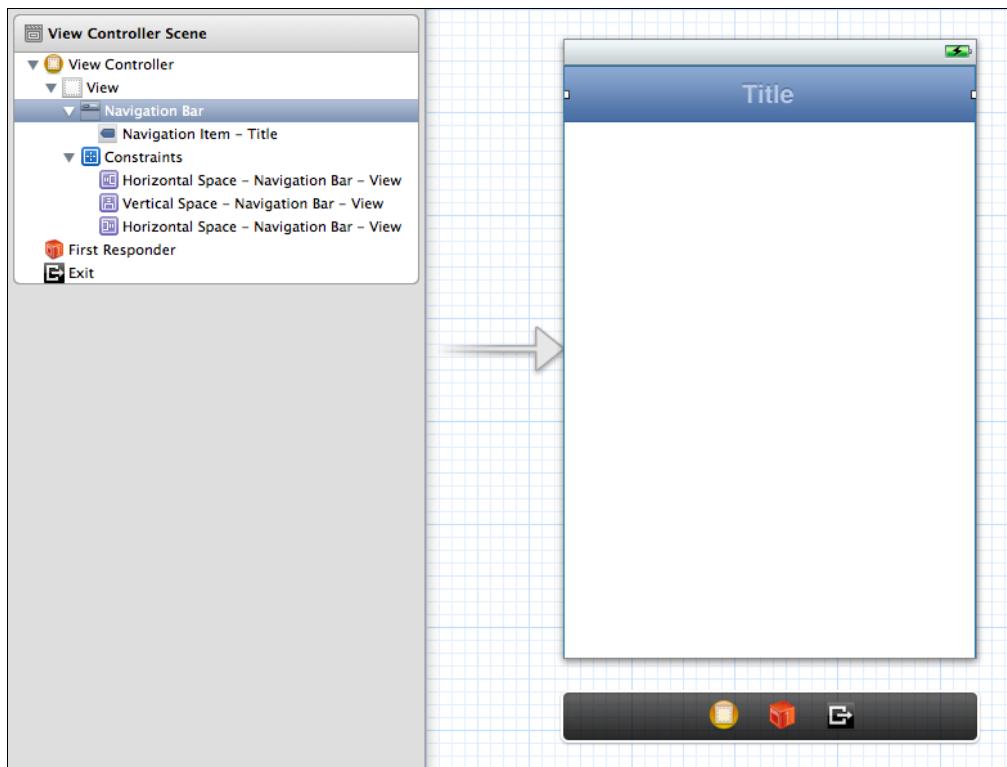
Create a new project from the Single View Application template. This time, enable the Use Storyboards setting. Call it "StoryboardLayout".

You will make the following, fairly standard, app layout:



Open **MainStoryboard.storyboard**. From the Object Library, drag a navigation bar onto the view. As you're dragging, notice that there are now no guides for the margins, because the navigation bar should always line up to the edges without a margin.

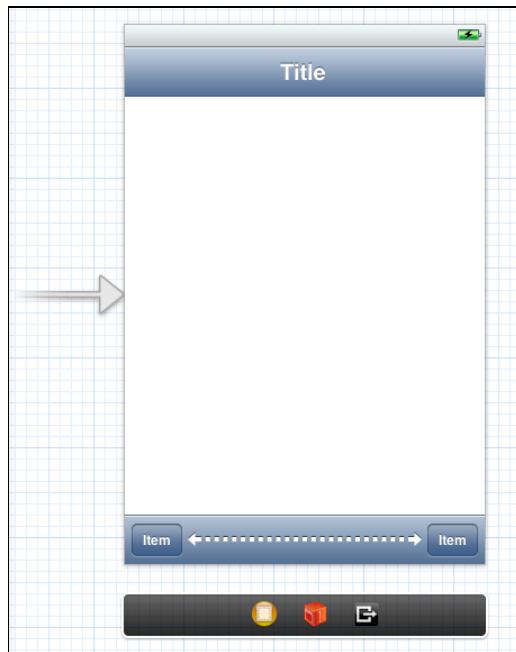
After you drop the Navigation Bar onto the canvas, there are three constraints: two Horizontal Spaces and one Vertical Space. They all have size 0 because the navigation bar snaps right to the edges.



Drag a new toolbar component to the bottom of the scene. The same thing happens there; toolbars are always supposed to sit at the bottom.

Drag a new bar button item onto the toolbar. Note that the bar button items in the toolbar are totally independent of Auto Layout. The toolbar does its own layout, using fixed space and flexible space items.

Drop a flexible space between the two bar buttons:



I just wanted to point this out. Even though the main view does its layout using Auto Layout, it doesn't mean that individual controls cannot override it to do their own layout. This also works for any custom controls that you might write.

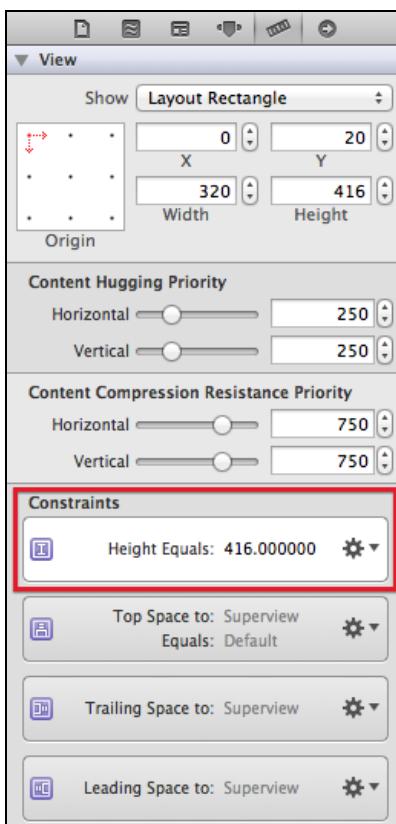
Drag a search bar onto the view and place it directly below the navigation bar. Because you snapped these two components together, there is now a 0-sized Vertical Space between them that keeps them glued together:



Note that these three items – the navigation bar, the search bar and the toolbar – all have an intrinsic height (44 points), but not an intrinsic width. That's why they have two Horizontal Space constraints each, but only need one vertically.

To fill up the rest of the space, drag a new table view onto the canvas. When I tried that, Interface Builder gave me some grief – it wouldn't resize the table view to fit nicely into the available space – so I had to drop it partially on top of the other controls. You may also find that it is hard to resize the table because its resize handles fall outside the editable area.

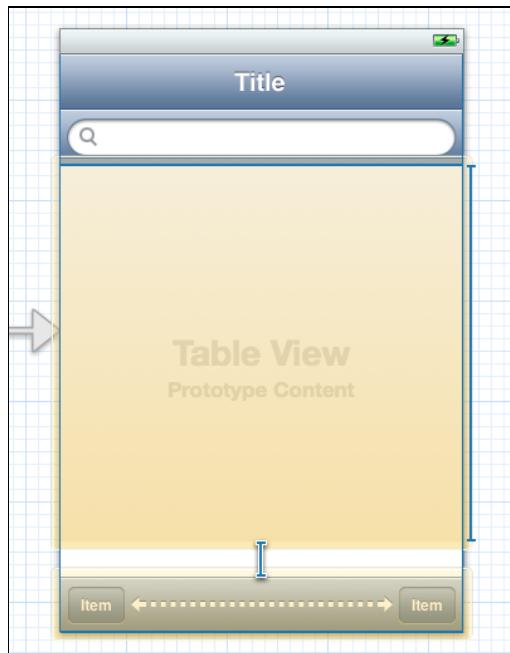
If that happens, go into the Size inspector for the table view. Do not edit the Height field directly, but select the Height Equals constraint instead:



The height of a new table view is fixed to 416 points, which is too large for your screen. Click the arrow next to the constraint's gear icon, choose **Select & Edit** from the popup menu, and change the Constant to 300. That is a bit smaller than it needs to be, but gives you some room to work with.

If necessary, drag the table view upwards or downwards until it snaps with the bottom of the search bar. There is still some space at the bottom between the table view and the toolbar. You could drag the bottom resize handle of the table view until it snaps with the toolbar, but you can also create a Vertical Space constraint manually. Let's do the latter.

Select both the table view and the toolbar. Choose **Pin\Vertical Spacing**. This adds a T-bar with 28 points of spacing between the two:

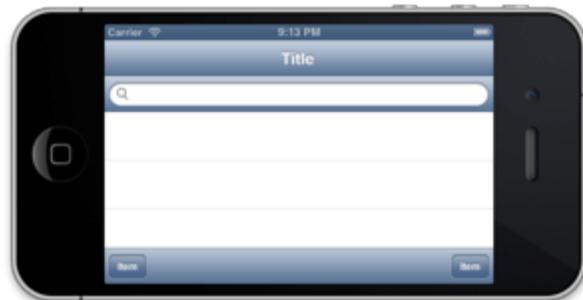


If you set this space to 0 now, then the toolbar will resize towards the table rather than the table towards the toolbar, or it will pull down the navigation and search bars. Try it out.

Undo to restore the height of that Vertical Space. Before you change it, you first have to delete the fixed Height constraint from the table view. This is now possible because the table view is anchored between the search bar and the toolbar with Vertical Space constraints, which is enough to determine the height of the table view.

After you delete the Height constraint, select the Vertical Space and set it to 0 again.

Run the app to make sure it all works. Also flip to landscape to see if everything rotates as it should.



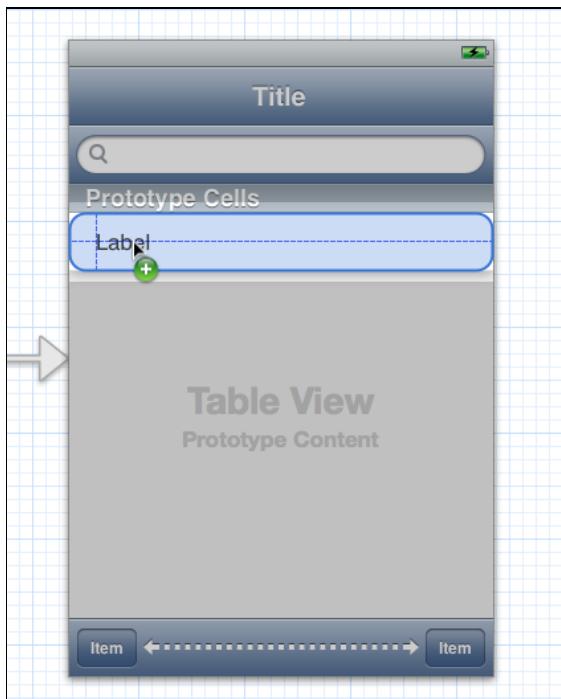
This was a pretty simple layout, but getting the table view to fit snugly between the search bar and toolbar was a bit tricky. That's because Auto Layout no longer works with absolute positions and sizes, but always derives these things from the constraints you set.

You sometimes have to add a new constraint (the Vertical Space between table and toolbar) before you can remove an old one (the table's fixed height). But you have to remove the old constraint before you can change the Constant value of the new constraint.

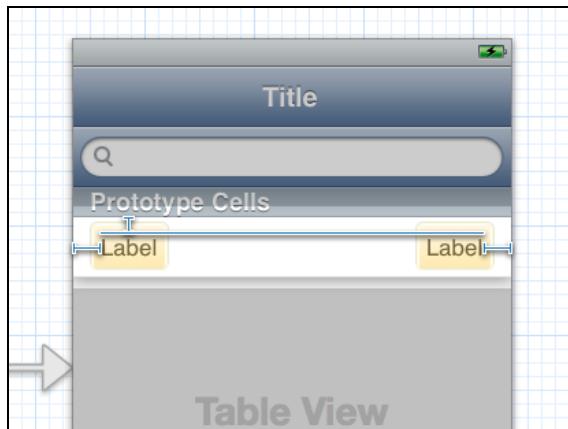


Unlike the toolbar, which uses its own layout system to place the bar button items, the cells in the table view do use Auto Layout. Add a prototype cell. Do this by first selecting the table view, and then in its Attributes inspector, set the Prototype Cells attribute to 1.

Drag a label onto the cell. Guides will appear when you drag, and when you drop the label, new constraints will lock it into place:

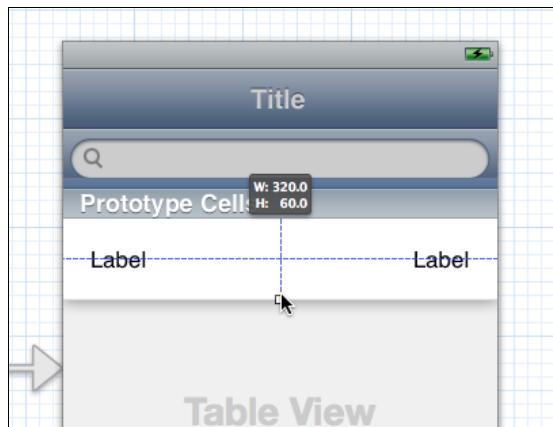


Drag another label onto the cell and place it on the right. You will likely end up with constraints that look like this, depending on exactly where you placed the labels:



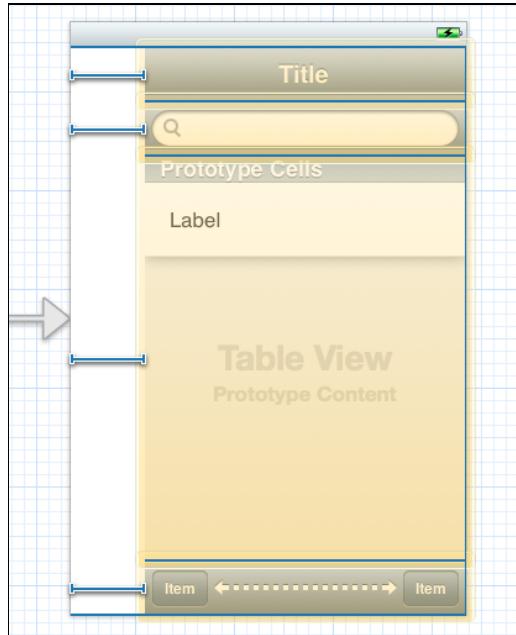
The left-hand label is pinned to the left and top edges of the cell; the right-hand label is pinned to the right edge and is top-aligned with the left-hand cell.

For fun, select the cell and drag the handle at the bottom up and down to resize the entire cell. Depending on how the labels are aligned within the cell, they should move up and down with it. In the screenshot above, the labels are stuck to the top so they won't move. Select the left-hand label and choose **Align\Vertical Center in Container**; then try resizing the cell again.



Mixing it up

The navigation bar, search bar, table view and toolbar are all glued to the scene's left edge using Horizontal (or Leading) Space constraints. Select these four constraints and change their Constant value from 0 to 60. This will make a bit of room on the left:

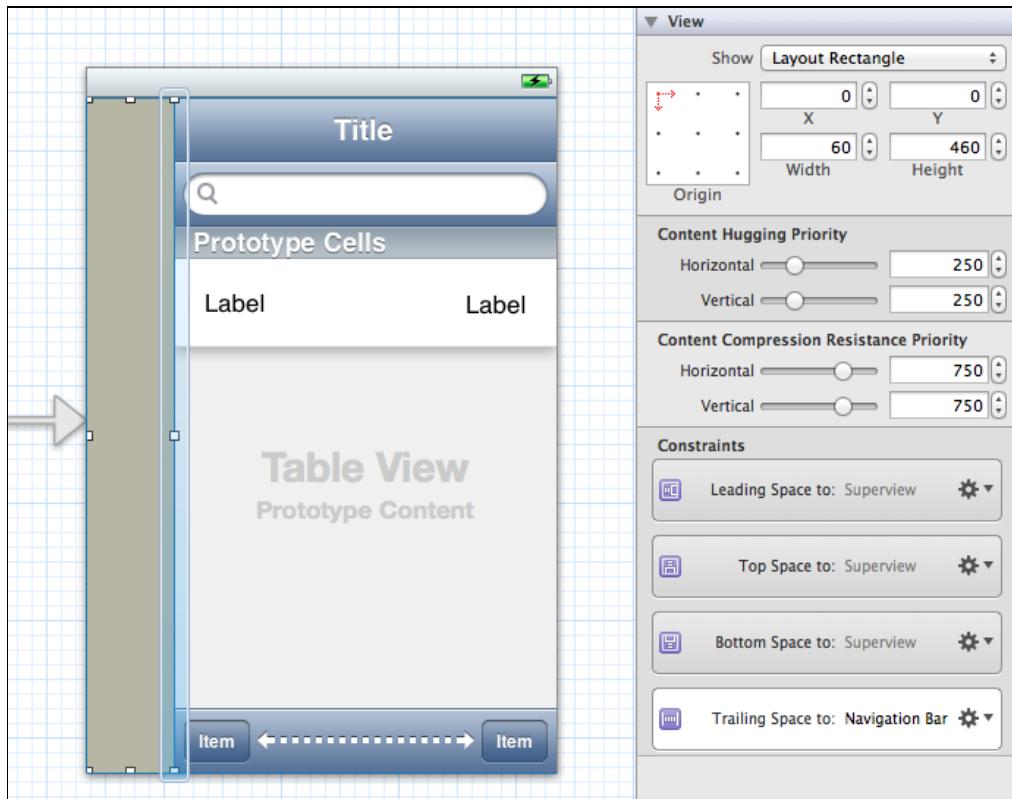


If this makes the right-most label in the table view disappear for you, as it did for me, then manually move it back. If the label is too far off the screen to grab with the mouse pointer, then go into the Size inspector and directly change its X-coordinate. Then drag it back into position.

Drop a new view object into the empty space. You may have to resize the new view manually because Interface Builder isn't smart enough to automatically fit it into the free space. The position of the new view will be (0, 0) and its size 60 x 460. You're going to pretend that this strip holds a sidebar menu with a number of buttons.

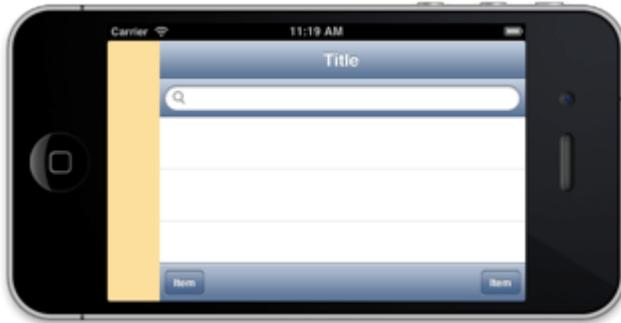
Change the background color of this sidebar view so that it is no longer white.

If you examine the constraints that Interface Builder set on the sidebar, you'll see that it now has a Trailing Space to the navigation bar with space 0 (or possibly the search bar, table view or toolbar – the editor seems to pick one of these components at random to glue the sidebar to):



This is good. You want the sidebar to stick to these other components.

Run the app and rotate to landscape. OK, it looks as intended. The sidebar keeps the same width, while the rest of the controls resize to take advantage of the extra room that landscape affords:



But what if you want it to be the other way around? In other words, make the sidebar stretch out while the table view stays the same size?

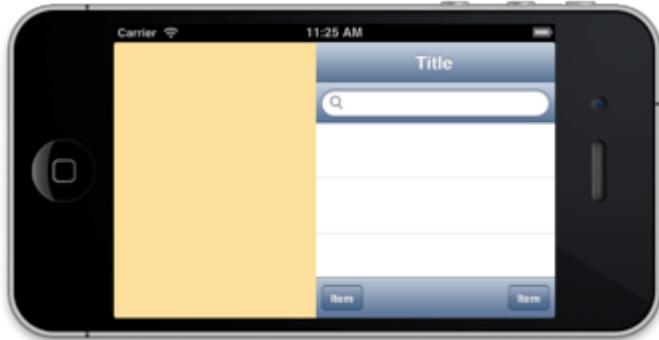
You have already learned all about the tools you need to pull this off. First give the navigation bar a fixed width with **Pin\Width**.

Select the navigation bar, search bar, table view and toolbar. Choose **Pin\Widths Equally**. You could also set a fixed width on all of these instead, but what you're expressing here is: "These components are all just as wide as the navigation bar."

(Yet another approach would have been to left-align them all with the navigation bar.)

You can now remove the 60-point Horizontal Space constraints on the left of these components, because those are still keeping them glued to the left edge of the screen.

Run the app and flip to landscape again. Now it should look like this:



Because the navigation bar has a fixed width, this also gives the table view and the others on the right the same width. The right edge of the sidebar is glued to the left edge of the navigation bar, so the sidebar gets stretched to take up all the room.

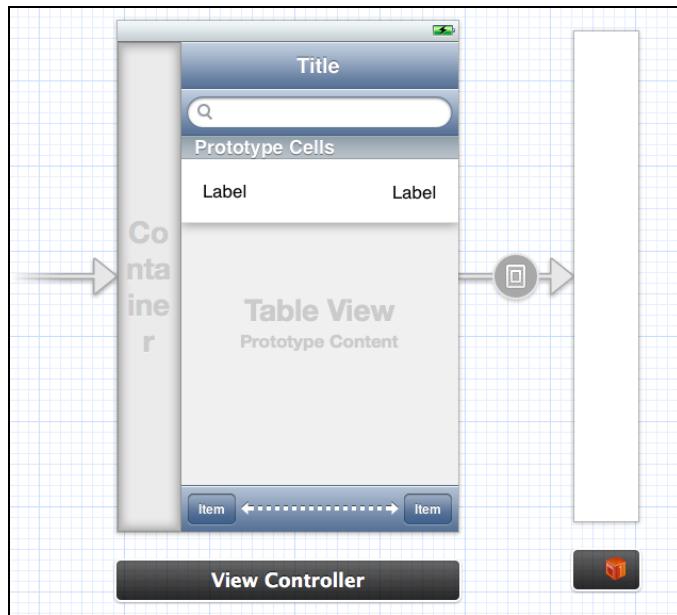
Embed it in a container view

iOS 6 added a cool new feature to Storyboards: the ability to embed the view of one view controller inside of another. You do this using container view objects.

Note: You can read more about container views in Chapter 21, "What's New with Storyboards."

First, delete the yellow view from the scene. This shouldn't have an impact on any of the other components; they still leave a 60-point gap on the left side of the scene. (Verify this by running the app.)

Find the container view object in the Object Library and drag it into the empty spot. In the Size inspector, set X and Y to 0, Width to 60 and Height to 460.



The interesting thing about container view is that the contents of this view now come from another view controller that is attached using an "embed segue".

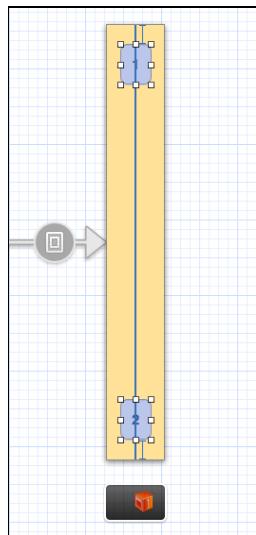
Change the background color of the embedded view controller's view to yellow. Run the app. It should work as before, even when you flip to landscape, except that the sidebar (and everything that you will add to it) now comes from a different view controller.

Note: If the sidebar doesn't grow when you rotate the app to landscape, then you need to make a Horizontal Space between the container view and the navigation bar, in order to glue them together. This happened automatically when I resized the container view to fit the empty space, but it is possible that you got a different set of automatic constraints.

Drag a new round rect button onto the embedded view controller. Note that you do not drag controls onto the container view's area, but onto the view controller that is attached to it. Put the button at the top and label it "1".

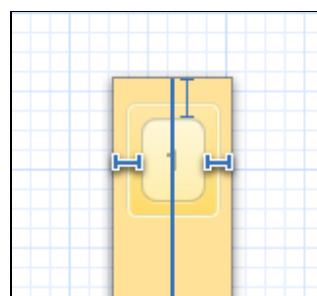
Drop another button at the bottom and label it "2".

Make sure both buttons are "sized to fit", and center them horizontally (rather than dragging them into position, remember to use **Align\Horizontal Center in Container**), so that it all fits into the view (there is not a lot of room to work with here).



Run the app. You should now see the two buttons in the sidebar.

For fun, also give the top button a **Leading Space to Superview** and a **Trailing Space to Superview**:



Run the app again. You should now get an “unable to simultaneously satisfy constraints” error message in the Xcode output pane. This happens because the Leading and Trailing Space constraints don’t mix very well with the Center X Alignment constraint.

If you look at the size of the two spacing constraints, you’ll see that the one on the left is 14 points and the one on the right is 13 points. The button cannot be centered if the left and right margins aren’t equal, even if Auto Layout will end up rounding them off to 14 and 13 points anyway. Auto Layout will handle any rounding for you behind the scenes, but you shouldn’t get in its way.

I’ve pointed this out so you know what is going on. Now that you do, get rid of the Leading and Trailing Spaces.

More fun with inequalities

What if you do want the table view to grow wider when switching to landscape, but only a little bit, so that the container view also gets room to grow? Currently there

is no way to express that – things either have a fixed width or they can grow/shrink, but you cannot tell them by how much.

That's where "inequalities" come into play again. You have already seen these a few times before, constraints whose Relation attribute is not Equal, but Greater Than or Equal (or Less Than or Equal, but these are much less common). Inequalities are constraints that you use for setting maximums and minimums.

The sidebar is now 60 points wide and the table view is 260. After switching to landscape, the table stays 260 but the sidebar becomes 220. Instead, you now want the sidebar to be 160 points (i.e. to grow by 100) and the table view to be 320 points (to grow by 60).

You can pull this off by making some of these constraints \geq instead of $=$.



The navigation bar has a fixed Width constraint that currently restricts it to always be 260 points wide. Change that constraint into Greater Than or Equal. That tells Auto Layout the navigation bar can grow beyond 260 points if possible. Because the search bar, table view and toolbar have Equal Widths constraints set on them, they will all grow an equal amount.

This change by itself is not enough. In fact, as soon as you turn the Width constraint into \geq , Interface Builder adds a new fixed Width constraint to the navigation bar in its place. Apparently, just having a \geq constraint is not enough to determine the size and position of the navigation bar. (If this didn't happen for you, then check the sidebar. It may now have a sparkling new 60 points Width constraint.)

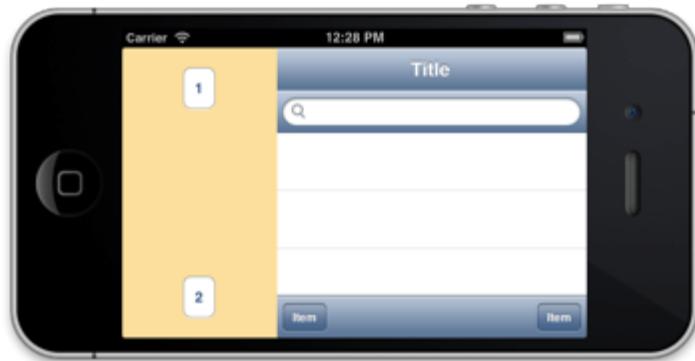
Add a new Width constraint on the sidebar (the container view). Set it to ≥ 60 . Now you can remove the fixed Width constraint from the navigation bar because it is no longer needed.

Run the app and flip to landscape. It still doesn't do what you want. The table view is still only 260 points wide, instead of 320. (Or, the sidebar will remain at 60 instead of resizing at all. This might happen if your Trailing Space on the container view is attached to something like the search bar or the toolbar instead of the navigation bar.)

Add a new Width constraint to the navigation bar. Change it to Less Than or Equal with a Constant value of 320. You haven't used this one before. Just as Greater Than or Equal means "not smaller than", you can think of Less Than or Equal as "not larger than." You use it to set a maximum.

The navigation bar now has two Width constraints on it, one ≥ 260 and one ≤ 320 . The sidebar just has ≥ 60 .

Run the app and flip to landscape, and now it should look like this:



Pretty nifty!

Try this: change the Width constraint on the sidebar to Equals 60. Run the app and flip to landscape. You will now get the dreaded "Unable to simultaneously satisfy constraints" error. Because the right edge of the sidebar is glued to the left of the navigation bar (possibly indirectly, if it's attached to something else like the search bar or toolbar), this means the navigation bar must stretch to $480 - 60 = 420$ points, but its maximum is 320 points.

Better change it back quickly!

Tip: You don't always have to run the app to see what the layout will look like in landscape. You can also rotate the view controller to landscape in the storyboard editor, by changing the Orientation attribute under Simulated Metrics in the Attributes inspector.

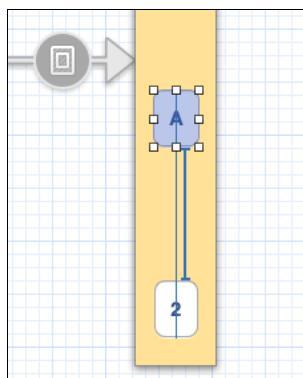
The embedded view controller from the container view will also rotate, because it always takes on the size of its container view. Interface Builder reevaluates the constraints and presents a new layout based on how it should look when rotated to landscape.

You can even resize the container view object to get a live preview of how the buttons in the embedded view controller react. But remember to be careful; Interface Builder may mess up your carefully laid out constraints when you do so.

Open to multiple interpretations: ambiguous layouts

Let's practice a little more with these inequalities, because they are important for making dynamic user interfaces.

Add a new button to the sidebar. Give it the title "A" and place it about 100 points above button "2". Put a Vertical Space between the two buttons, size 100:



This middle button is only attached to button 2. Run the app and rotate to landscape. Hmm, the visual designer in me says the button gets a bit too close to button 1 here. Visually I always want it to be in close proximity to button 2, not button 1. In portrait that works fine, but not in landscape:

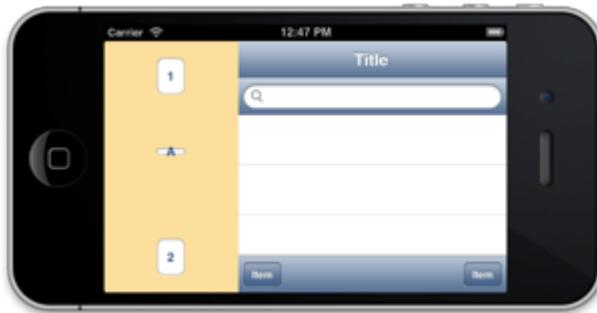


In this example, you want the distance between button A and button 2 to be 100 points in portrait, but in landscape it would be nicer if the three buttons were equally spaced.

Let's do some math. In landscape, the screen is 300 points high (not counting the 20 points for the status bar). The top and bottom margins are both 20 points. The three buttons are 44 points high. That leaves $300 - 20 \times 2 - 44 \times 3 = 128$ points. That means button A has to be 64 points away from both buttons 1 and 2.

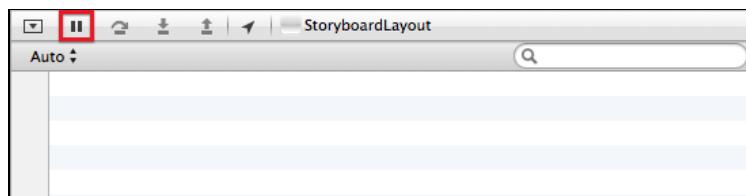
Obviously, you cannot place a 64-point fixed Vertical Spacing between the buttons, as that will give a "conflicting constraints" error in portrait mode. But you can use an inequality here. Select buttons 1 and A and add a new Vertical Spacing. Change it to ≥ 64 points. This should push button A down in landscape mode but keep it in place in portrait.

Simple enough, right? Run the app to see if it works. Whoa, that looks weird, button A gets completely squashed! (It's also possible that button 1 gets squashed instead.)



Whenever you see something like this, there is a good chance that you have an *ambiguous* layout. That means Auto Layout has multiple ways of solving the constraints, but probably none that do what you want. This is why button 1 gets squashed sometimes instead of button A. The layout is ambiguous, after all – there is more than one solution, but none of them are any good. Which one you get to see is a matter of chance...

Press the pause button in Xcode to stop the app in the debugger:



There should now be an (lldb) prompt in the output pane. Type the following at that prompt and press Enter:

```
po [[UIWindow keyWindow] _autolayoutTrace]
```

This shows a whole bunch of stuff, including:

```
| | *<UIView:0x717d680> - AMBIGUOUS LAYOUT
| | | *<UIView:0x718d490> - AMBIGUOUS LAYOUT
| | | | *<UIRoundedRectButton:0x718d4f0> - AMBIGUOUS LAYOUT
| | | | | <UIGroupTableViewCellBackground:0x718dc0>
| | | | | <UIImageView:0x718e400>
| | | | | <UIButtonLabel:0x718e4f0>
| | | | *<UIRoundedRectButton:0x718e640> - AMBIGUOUS LAYOUT
| | | | | <UIGroupTableViewCellBackground:0x718e740>
| | | | | <UIImageView:0x718e810>
| | | | | <UIButtonLabel:0x718e870>
| | | | *<UIRoundedRectButton:0x718f270> - AMBIGUOUS LAYOUT
| | | | | <UIGroupTableViewCellBackground:0x718f370>
| | | | | <UIImageView:0x718f670>
```

```
| | | | <UIButtonLabel:0x718f750>
```

What you see here is a debug dump of all the views in your layout. Notice where it says AMBIGUOUS LAYOUT. That is Auto Layout letting you know that some of your constraints need fixing.

It's a bit annoying to pause the app all the time just to find out where your layout is ambiguous, but there's a trick you can use that tells you every time you rotate the app.

Add the following code at the top of **ViewController.m**:

```
@interface UIWindow (AutoLayoutDebug)
+ (UIWindow *)keyWindow;
- (NSString *)_autolayoutTrace;
@end
```

Put this method somewhere in the `@implementation` section:

```
- (void)didRotateFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
{
    NSLog(@"%@", [[UIWindow keyWindow] _autolayoutTrace]);
}
```

This essentially does the same thing as the command that you typed into the debugger, except that this code writes to the log every time you rotate the device. The category on `UIWindow` is necessary because these methods aren't public, and without these declarations the compiler will not build the app.

Important: You shouldn't use the above code in production, only for debugging.

Try it out. Run the app. Rotate to landscape and it shows you the debug output with AMBIGUOUS LAYOUT markers next to the views that have problems. Rotate back to portrait and the AMBIGUOUS LAYOUT markers should be gone.

Note: If you are paying attention to the log then you should have seen that the navigation bar, search bar, table view and toolbar also say AMBIGUOUS LAYOUT after rotating to landscape. There doesn't seem to be anything wrong with them, so why do they say that?

It turns out that the sidebar needs a Width constraint that is Less Than or Equal 160, so that Auto Layout knows the sidebar doesn't want to grow bigger

than 160 points. Add this constraint now and run the app again. This time the ambiguous layout warnings should be gone for those views.

But, if your Trailing Space for the container view was with something other than the navigation bar, you might also see the sidebar staying at 60 pixels in landscape. These are the vagaries of Auto Layout. ☺ To fix, simply add another Trailing Space constraint between the container view and the navigation bar and delete the other Trailing Space constraint.

When you're making fairly complex layouts such as these, it's always a good idea to add this code snippet to your view controllers just to double-check that your layouts are not ambiguous. Interface Builder will work hard to prevent you from making invalid layouts, but it just cannot catch everything.

Here is an additional debugging tool that you can use to gain an understanding of any ambiguities your layout may have.

First, create a new `UIViewController` subclass for the sidebar. Make a new file using the Objective-C class template. Name it "SidebarViewController", subclass of `UIViewController`.

Add the following method to **SidebarViewController.m**:

```
- (IBAction)buttonTapped:(UIButton *)sender
{
    [sender exerciseAmbiguityInLayout];
}
```

Return to **MainStoryboard.storyboard** and in the Identity inspector set the Class of the sidebar view controller (the yellow one) to "SidebarViewController".

Note: Remember, if you tap on the yellow view, the view is selected instead of the view controller. So tap on the black bar below the yellow view to select view controller. Or, use the Document Outline. ☺

Now connect all the buttons to the `buttonTapped:` action method.

Run the app again. In portrait mode, tapping the buttons will have no effect; there is no ambiguity in the layout here, so nothing happens. Switch to landscape. Now tapping the buttons will make Auto Layout cycle through the different possible scenarios it has calculated. This allows you to see where the ambiguities are.

Auto Layout can solve this particular layout in three different ways, but only by crunching one of the buttons:



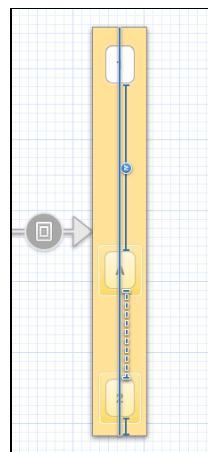
Setting the Vertical Spacing between the top and middle buttons to ≥ 64 is not enough.

What if you changed the spacing between button A and button 2 to \geq as well? Currently it is a fixed space of 100 points. Try it out: select that constraint and make it Greater Than or Equals 64.

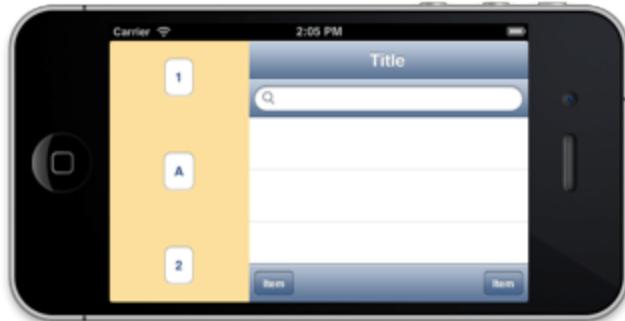
As soon as you make it \geq , Interface Builder adds in a new Vertical Space constraint to anchor button A to the bottom of the view. That is obviously not going to work!

Here is the key thing: **Inequalities by themselves are usually not enough.** If you only have \geq or \leq constraints, then Auto Layout does not have enough information to calculate the positions of these buttons. To avoid such ambiguity, Interface Builder keeps adding constraints, often using explicit widths and heights.

To fix this, you need to use a combination of inequality and equality constraints with different priorities. Undo to restore the Vertical Space constraint between the A and 2 buttons to an Equals constraint of 100 points. Set its priority to 700, making the constraint a dotted line:



Now run the app. There should be no more ambiguous layout and the A button should sit exactly between the buttons at the top and bottom:



If you only have two inequalities that are both ≥ 64 , then Auto Layout has no idea how to resolve this in portrait mode. There is no problem in landscape because the only possible solution there is to make both Vertical Spaces 64 points big. But in portrait there is a lot more room, so by how much should Auto Layout increase these Vertical Space constraints? There is an almost infinite number of possible answers here – the A button could sit anywhere between buttons 1 and 2.

To turn this into a valid layout, Auto Layout needs the lower priority equality to fall back on. It cannot satisfy the ≥ 64 constraint, but now there is a lower priority Vertical Space that says, “Well, in that case, put button A 100 points above button 2.”

Priorities are useful for “...if possible” situations. Here you say, “Put button A 100 points above button 2, if possible. If not, then center it between the other two buttons.”

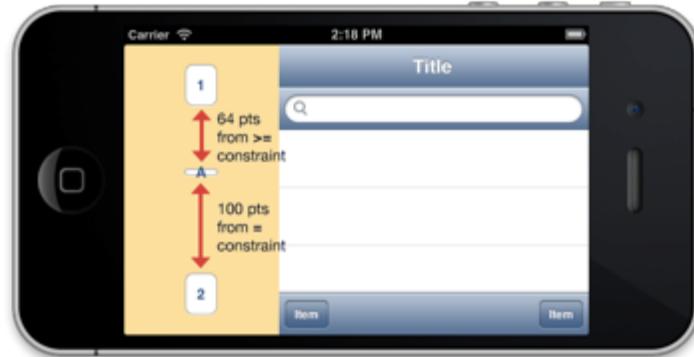
Note: There isn't a “center this view between these two other views” constraint, which is why you had to calculate the ideal distance – 64 points – by hand. If you change these constants to something lower, then button A will sit 100 points from the bottom again. If you make the constants too high, then the layout will become ambiguous again and the controls will look messed up. Try it out to see for yourself.

The priority value you set is important. Auto Layout assigns meaning to these priority values. In this particular case, the priority must be lower than the center button's Content Compression Resistance property.

Recall that the Compression Resistance prevents you from making the button smaller than its ideal content size. This resistance has a priority like constraints do. The default value for the Compression Resistance priority is 750, which is why you set the priority of the Vertical Spacing to 700 (even 749 would be fine; all that matters is that one is lower than the other, but not by how much).

To see the effect this has, lower the vertical Content Compression Resistance priority of button A to 699 so that it is lower than the priority of the fixed spacing. (As always, you can do this in the Size inspector.)

Run the app, and notice that Auto Layout now shrinks the button to make room, rather than ignoring the fixed Vertical Spacing constraint:



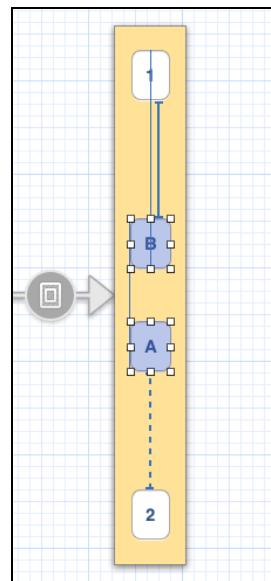
This is obviously not what you want, so restore the Content Compression Resistance priority of the button to its default value of 750.

Even more buttons

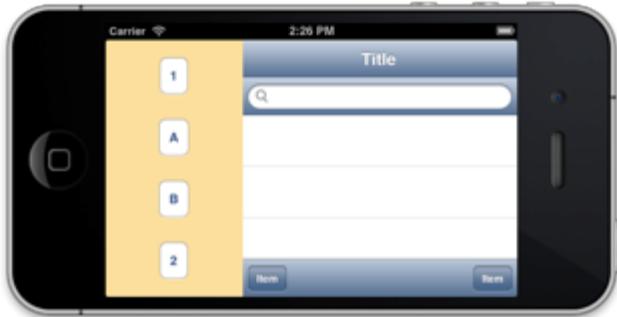
Drag another round rect button onto the sidebar's view controller. Name it "B" and place it roughly 100 points below button 1.

Remove the Vertical Space between 1 and A. Add a new Vertical Space between 1 and B, set to 100 points distance. Remove any other vertical constraints that B may have.

It should look like this:



Run the app and flip to landscape. Oddly enough, the B and A buttons appear to have swapped positions. B should be above, not below A:

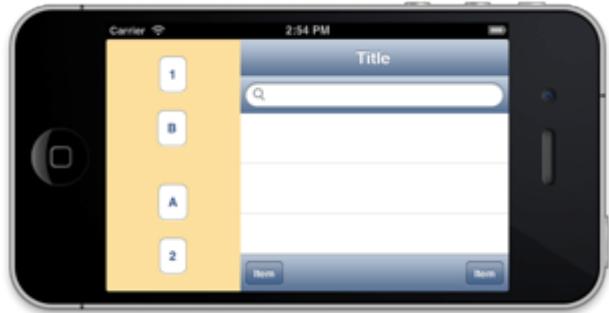


Of course, this is merely an illusion, but you don't want that to happen regardless. Put a Vertical Spacing between buttons A and B (this will be 44 points high).

Run the app again. Now it looks completely wonky!

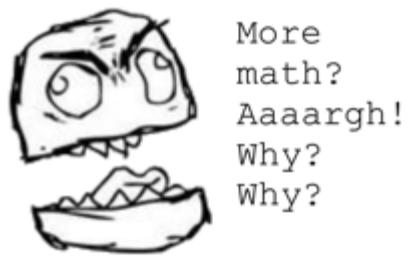
The fix is fairly straightforward but it requires some math again. Remember that with just three buttons, the distance between button 1 and button A was 64 points in landscape? This time you have to subtract 44 from that, which is the distance between B and A, leaving 20 points. Change the Vertical Spacing between 1 and B to ≥ 20 points.

This works and there is no ambiguity in the layout:



You will probably never make a layout like this in a real app, but it's fun to play with so you can see how all these constraints and priorities work together.

Something that you might want to do in a real app is put these four buttons at equal distances from each other, in both portrait and landscape. Obviously, in portrait those distances will be larger than in landscape. Let's make that final change and then call it quits for this chapter.



More arithmetic: four buttons times 44 points is 176 points. In portrait the screen is 460 points high, minus the 20 points margin at the top and at the bottom. That leaves $460 - 20 \times 2 - 176 = 244$ points that you can put between the buttons.

There are four buttons, so three vertical spaces, which makes it $244 / 3 = 81$ points for each space, with a small remainder because it's not an even division. For landscape, the math is similar: $(300 - 20 \times 2 - 176) / 3 = 28$ points for each vertical space.

Change the Vertical Space between buttons B and A to Equal 81. Change the space between A and 2 to Equal 82 (the remainder gets put inside this one) and restore the priority to 1,000. Finally, change the Vertical Space between 1 and B to Equal 81.

It's best to do it in this order, changing the \geq constraint last, so that Interface Builder won't mess up any of the button sizes. Should Interface Builder mess up your buttons anyway, use **Size to Fit Content** (or **Cmd =**) to restore them to their optimal sizes.

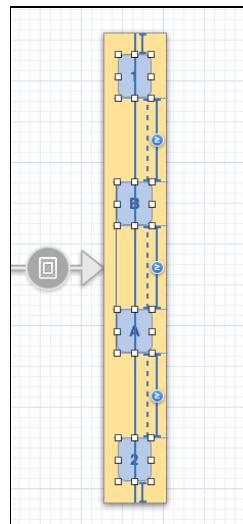
Obviously, these constraints will only work in portrait mode. Auto Layout cannot fit everything into the smaller landscape height.

The solution is similar to before: make all the vertical space constraints optional. Give them a priority that is less than 750, the Compression Resistance priority for the buttons. You can **Cmd-select** the three Vertical Space constraints from the Document Outline and set this in one fell swoop.

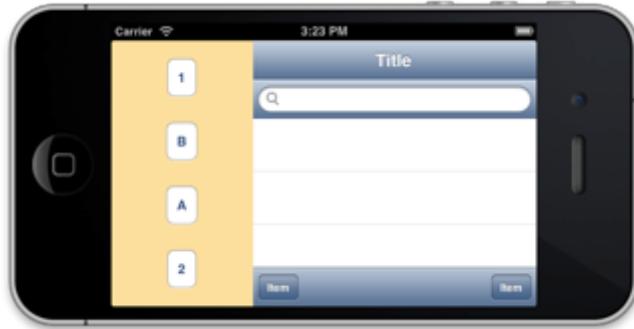
Just changing this doesn't fix anything, because it makes no sense to lower the priority unless you also make something that has a higher priority. (However, because the priority is now lower than the compression resistance, the buttons no longer collapse when you flip to landscape.)

To make this work in landscape, you have to add a new Vertical Spacing between each of the buttons, and make it Greater Than or Equal with a Constant value of 28, which is the distance we calculated earlier.

The final design looks like this:



Run the app and... ta-da! The buttons are vertically equidistant:



Try this: set the \geq Vertical Spacing constants to size 0 instead of 28. What do you think happens now?

You get another ambiguous layout. After all, Auto Layout only knows that buttons B and A must be at least 0 points away from each other. It cannot magically guess that your intention was to give all the buttons the same distance. That is why you need to help it out a bit by specifying the exact distance for landscape, but inside an \geq constraint so that it doesn't cause any problems in portrait. Phew!

If you can't get enough of this stuff, try adding two additional buttons so that there will be six total. You'll have to face a new problem, because $6 \times 44 = 264$, plus 40 points for the top and bottom margins = 304, and there are only 300 points vertically. That means you will have to make the margins flexible as well, so they become smaller than 20 points in landscape. Have fun! ☺

Where to go from here?

If you followed this chapter all the way through, then you have spent a lot of time messing about – and sometimes fighting! – with constraints in Interface Builder. I

hope these examples gave you insight into how constraints work and how you can manipulate them to give you the layouts you need.

Most of the time, your layouts will be simple, and you'll just line up your buttons and labels and be done with it. But sometimes you may need to dig into your bag of tricks and apply some inequalities and priorities, especially if your views need to be resizable.

There is much more to Auto Layout! In the next chapter, you will learn:

- How to give your custom views and controls their own intrinsic content sizes, so you can avoid setting fixed Width and Height constraints on them.
- Tips for migrating from your old springs and struts-based projects to Auto Layout.
- Everything you need to know about internationalization and localization. Auto Layout makes localization so much easier that this feature alone makes it worth switching.
- How to use Auto Layout programmatically. Interface Builder is pretty handy when it comes to making layouts, but there are certain types of constraints it cannot create – such as a constraint that keeps the aspect-ratio of a view intact – and sometimes it is downright frustrating to use, as you have no doubt experienced in the course of this chapter. Fortunately, it is also possible to make constraints in code, and we will take an in-depth look at that.

If reading this chapter made you sick to your stomach, then remember that Auto Layout is an optional feature. If you don't want to use it, then simply don't use it. Disable the Auto Layout checkbox in the File inspector for your nib or storyboard and you no longer have to worry about it!

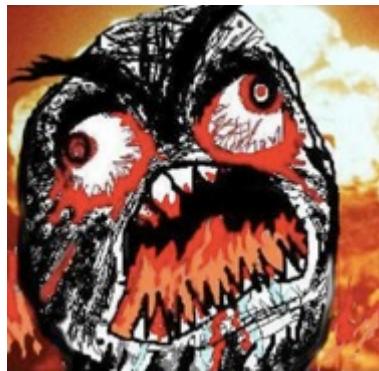
Chapter 4: Intermediate Auto Layout

By Matthijs Hollemans

In the previous chapter, Beginning Auto Layout, you learned how to use iOS 6's new Auto Layout system using Interface Builder. With Auto Layout, you no longer have to hardcode the positions and sizes of your views. Instead you add constraints, using the Pin and Align menus to keep them in place.

That works quite well, but if your layouts reach a certain level of complexity, Interface Builder can become, uhhh, *frustrating* to use – and that's putting it mildly! You may have experienced this already if you tried to move views around and suddenly some of your constraints were gone.

This is the sort of thing that can quickly throw one into nuclear rage mode.



Interface Builder tries to prevent you from making invalid layouts, and to do so it always puts the “minimum best set” of constraints in place. That is very considerate of it, but the obvious downside is that, when you move things around or resize them, chances are high that Interface Builder will mess up your carefully laid-out constraints.

What's more, Interface Builder cannot do everything. It only provides a limited selection of constraints. For example, there is no way to create a constraint that keeps the *aspect ratio* of a view – how wide versus how high the view is – the same at all times.

Interface Builder also doesn't have any way to change your screen layouts at runtime, like when you add new subviews to a view for example.

Fortunately, you can also work with Auto Layout constraints programmatically, and that is what you'll learn to do in this chapter. Knowing how to manipulate `NSLayoutConstraint` objects programmatically unlocks the full power of Auto Layout.

For mere mortals it takes many years of dedicated training to become masters of this dark art, but all *you* have to do is make your way through the next hundred pages and come out alive! ☺



Don't worry, it won't be that extreme – you can survive this! Because Auto Layout is such a huge – and oftentimes confusing – topic, this chapter provides many different examples and exercises for you to dig your teeth into. That's why it is of epic proportions. The trip will be worth it, but make sure you take regular breaks and stretch your legs once in a while.

Ready? Let's continue our journey!

The magic formula

Recall that a constraint simply represents a relationship between two views.

Interface Builder has different types of constraints: Width, Height, Horizontal Space, Vertical Space, Center Alignment, Left and Right Alignment, Widths Equally, and so on. It may sound like a lot, but these are all variations on the same theme.

In fact, Auto Layout only knows one type of constraint. All these different relationships between views work the same under the hood, using a simple mathematical formula:

$$A = B * m + c$$

Here, A and B are attributes of the two views, such as "this button's left side" or "that label's top", m is a multiplier and c is the constant.

If the formula is too abstract for you, think of it like this:

$$\text{view1-attribute} = \text{view2-attribute} * \text{multiplier} + \text{constant}$$

In the last chapter, you saw the Constant property in the Attributes inspector – it was the setting you could modify to set the distances between the different views. The multiplier for Interface Builder constraints is always 1, and there is no setting for it in the Attributes inspector.

Here's a typical example of a constraint expressed by this formula:

okButton.left = cancelButton.right*1 + 12

This means the OK button's left edge is 12 points to the right of the Cancel button's right edge. This is an *equality* constraint, because the operator is `=`, but you can also use `>=` and `<=` for Greater Than or Equal and Less Than or Equal constraints, respectively. And of course, constraints can have different priorities, but you don't put that into the formula.

In code, you will use the `NSLayoutConstraint` class to represent your constraints. Unbelievable as it may sound, this is the only public new class that was added for Auto Layout. Everything else happens in `UIView` and `UIViewController`.

Let's play with this magic formula in a new app.

Constraints from code

Create a new project using the Single View Application template and name it "CodeConstraints." Disable storyboards, but enable Automatic Reference Counting. This creates a standard app project with a `ViewController.xib`, but you won't be using the nib either – everything will be done in code this time.

Begin by adding a simple button to the view. In `ViewController.m`, add a new instance variable for the button:

```
@implementation ViewController
{
    UIButton *button1;
}
```

The `viewDidLoad` method is a good place to create this button and to set up your constraints:

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

    button1 = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [button1 setTitle:@"Button 1" forState:UIControlStateNormal];
    [button1 sizeToFit];
```

```
[self.view addSubview:button1];  
}
```

If you haven't created views from code before, then rest assured that the process is fairly simple. You first `alloc` and `init` the view, or use a convenience constructor such as `buttonWithType:`, and then you configure it. Here you set the button's title and call `sizeToFit` to give the button its ideal size. Finally, you add the button to the screen by calling `addSubview:`. Easy-peasy.

Run the app. The new button appears in the top-left corner:



Remember how in the previous chapter, in order to use Auto Layout in a nib or storyboard, the "Use Autolayout" setting had to be checked in the File inspector?

If you want to use Auto Layout from code, you have to do the opposite and disable the old springs-and-struts mechanism.

Add the following line before the call to `addSubview:`:

```
button1.translatesAutoresizingMaskIntoConstraints = NO;
```

The `translatesAutoresizingMaskIntoConstraints` property exists for backwards compatibility, and it allows you to combine views that use springs-and-struts with Auto Layout. When the property is `YES`, UIKit automatically gives the view constraints based on its old autosizing mask. You don't want that to happen when you're making your own constraints, so it's best to set `translatesAutoresizingMaskIntoConstraints` to `NO`.

In addition to that, the Auto Layout system does not kick in until you set at least one constraint on the view. Add this to the bottom of `viewDidLoad`:

```
NSLayoutConstraint *constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeRight
    relatedBy:NSLayoutRelationEqual
    toItem:self.view
    attribute:NSLayoutAttributeRight
    multiplier:1.0f
    constant:-20.0f];

[self.view addConstraint:constraint];
```

That's quite a mouthful, but it's less scary than it looks. Recall that a constraint represents a simple formula: $A = B*m + c$. In this case, the formula is:

**button1:NSLayoutAttributeRight =
self.view:NSLayoutAttributeRight*1.0f - 20.0f**

In other words, the position of the button's right edge is equal to the position of its superview's right edge, minus 20. Or in simpler terms, the button is glued to the window's right border with a 20-points margin between them.

Run the app to see if it works. It does!



Of course, `:NSLayoutAttributeRight` is not a real property on `UIView` or `UIButton`. The formula is only supposed to describe your intent, and Auto Layout will figure out for itself which properties to set.

This one constraint only describes the horizontal position of the button. To make the button sit at the bottom of the screen, you need to add another constraint, so add this right after the other constraint:

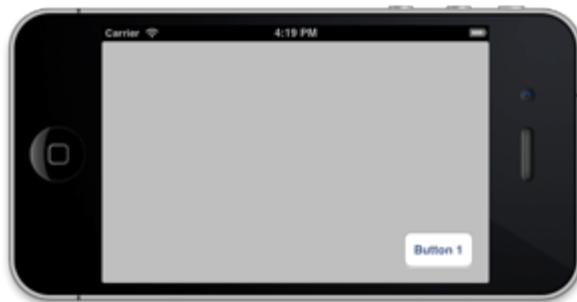
```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeBottom
    relatedBy:NSLayoutRelationEqual
    toItem:self.view
    attribute:NSLayoutAttributeBottom
    multiplier:1.0f
```

```
constant:-20.0f];  
  
[self.view addConstraint:constraint];
```

It's almost exactly the same as the first one, except that this time you're using the `NSLayoutAttributeBottom` attribute instead of `NSLayoutAttributeRight`. The formula is now:

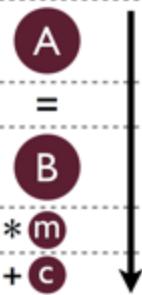
button1.bottom = self.view.bottom*1.0f - 20.0f

And here's the result. The button stays in the bottom-right corner, even after you flip to landscape:



The code statement to create a new constraint may look a bit intimidating, but all it does is express the familiar formula, if you read it from top to bottom:

```
[NSLayoutConstraint  
constraintWithItem:  
attribute:  
relatedBy:  
toItem:  
attribute:  
multiplier:  
constant:]
```



These are the different view attributes that you can use to make constraints:

- `NSLayoutAttributeLeft`
- `NSLayoutAttributeRight`
- `NSLayoutAttributeTop`
- `NSLayoutAttributeBottom`
- `NSLayoutAttributeLeading`
- `NSLayoutAttributeTrailing`
- `NSLayoutAttributeWidth`

- `NSLayoutAttributeHeight`
- `NSLayoutAttributeCenterX`
- `NSLayoutAttributeCenterY`
- `NSLayoutAttributeBaseline`

Most of these should sound obvious and you'll use them all in the rest of this chapter.

Notice that you have used `NSLayoutAttributeRight` to align the button with the right border of the screen, but there is also an `NSLayoutAttributeTrailing` attribute.

You might recall the difference between these two from the previous chapter: they mean the same thing for languages such as English and other European languages, but for right-to-left languages such as Hebrew and Arabic, leading becomes "right" and trailing becomes "left." So if you want this button to stick to the window's left side for Hebrew and Arabic, it is better to change the first constraint to:

```
NSLayoutConstraint *constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeTrailing
    relatedBy:NSLayoutRelationEqual
    toItem:self.view
    attribute:NSLayoutAttributeTrailing
    multiplier:1.0f
    constant:-20.0f];

[self.view addConstraint:constraint];
```

Make that change and test it out. Assuming that you're running this on a device or the Simulator with a left-to-right language such as English, the results should be no different from before. In the section on internationalization and localization later in this chapter, you will see the effect the leading and trailing attributes have on a right-to-left language.

Note: To see what happens when you forget to disable the old springs-and-struts mechanism, set the `translatesAutoresizingMaskIntoConstraints` property to `YES` instead of `NO`, or simply remove the entire line.

Run the app and – POW! – there's a nasty-looking error message waiting for you in the Xcode output pane:

"Unable to simultaneously satisfy constraints."

Auto Layout cannot calculate a valid layout for this screen because one or more constraints are conflicting. At least the error message tells you which constraints are causing the trouble:

```
<NSAutoresizingMaskLayoutConstraint:0x7437070 h=---- v=----  
H:[UIWindow:0x7143490(320)]>  
  
<NSAutoresizingMaskLayoutConstraint:0x7435790 h=-&- v=-&-  
UIView:0x714b230.width == UIWindow:0x7143490.width>  
  
<NSAutoresizingMaskLayoutConstraint:0x7433f60 h=-& v=-&  
H:[UIRoundedRectButton:0x714ac50(98)]>  
  
<NSAutoresizingMaskLayoutConstraint:0x7433e70 h=-& v=-&  
UIRoundedRectButton:0x714ac50.midX == + 49>  
  
<NSLayoutConstraint:0x7180e20 UIRoundedRectButton:0x714ac50.trailing ==  
UIView:0x714b230.trailing - 20>
```

What I want to point out is that whenever you get this error and you see a bunch of constraints with something like “h=-& v=-&” in their description, that means these are automatically-generated constraints based on the view’s autosizing mask. And they’ve gotten snarled up with your other constraints.

You can also see this in the name of the constraint: `NSAutoresizingMaskLayoutConstraint`. If this happens, you know you probably forgot to set `translatesAutoresizingMaskIntoConstraints` to `NO`.

Don’t forget to put it back the way it was to make the app work again!

What do you think you need to do to place the button in the bottom center of the screen? Try and code it yourself – you can always check back here if you get stuck!

I’m waiting. . .

Were you able to figure it out? If so, you’re well on your way to becoming an auto layout master already! ☺

As you may have figured out, the vertical constraint doesn’t need to change – the button stays glued to the screen bottom – but the horizontal one does. It’s pretty simple; just change the first constraint to:

```
NSLayoutConstraint *constraint = [NSLayoutConstraint  
constraintWithItem:button1  
attribute:NSLayoutAttributeCenterX  
relatedBy:NSLayoutRelationEqual  
toItem:self.view  
attribute:NSLayoutAttributeCenterX  
multiplier:1.0f  
constant:0.0f];
```

The formula is now:

button1.centerX = self.view.centerX*1.0f + 0.0f

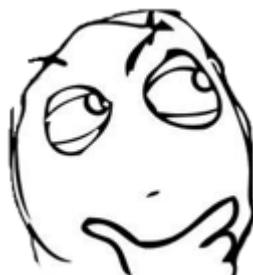
Or simplified, because the multiplier is 1 and the constant is 0:

button1.centerX = self.view.centerX

Run the app and try it out.



Believe it or not, but this one formula, $A = B*m + c$, really is enough to describe all the constraints you have seen in Interface Builder... and a lot more!



Fixing the width

What if you want to set a view to a fixed width or height, instead of auto-sizing it like you're currently doing for the button? To do this, you can use the `NSLayoutAttributeWidth` or `NSLayoutAttributeHeight` attributes. Try it out by adding this new constraint below the others:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeWidth
    relatedBy:NSLayoutRelationEqual
    toItem:nil
    attribute:NSLayoutAttributeNotAnAttribute
    multiplier:1.0f
    constant:200.0f];

[button1 addConstraint:constraint];
```

Up until this point, the constraints you've added have been between two different views – the button and its superview. But this time, the width and height constraint only applies to a single view – the button itself. So what should you set for A and B in the formula in this case?

As you can see above, you can express this situation by setting the `toItem:` parameter to `nil` and the corresponding `attribute:` to `NSLayoutAttributeNotAnAttribute`.

The formula is now:

button1.width = nil * 1.0f + 200.0f

Which reduces to simply:

button1.width = 200.0f

It looks a little weird in the code when you say "the item is `nil` and the attribute is not an attribute," but that was done so the Auto Layout API would just have one method that could handle all possible situations, rather than having many different methods to cope with every possible situation.

Also notice that this time you're adding the constraint to the button itself, not `self.view`. It would still work if you did `[self.view addConstraint:]`, but the general rule is that you add constraints to the superview, unless the constraint applies just to the view itself.

Run the app. The button is now a lot wider indeed:



Let's play a bit more with these constraints. What if you want to have a button that is 200 points wide in portrait but 300 in landscape?

Replace the fixed width constraint with these two:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeWidth
    relatedBy:NSLayoutRelationGreaterThanOrEqual
    toItem:nil
    attribute:NSLayoutAttributeNotAnAttribute]
```

```
multiplier:1.0f  
constant:200.0f];  
  
[button1 addConstraint:constraint];  
  
constraint = [NSLayoutConstraint  
constraintWithItem:button1  
attribute:NSLayoutAttributeWidth  
relatedBy:NSLayoutRelationLessThanOrEqual  
toItem:nil  
attribute:NSLayoutAttributeNotAnAttribute  
multiplier:1.0f  
constant:300.0f];  
  
[button1 addConstraint:constraint];
```

This adds two new constraints to the button:

button1.width >= 200.0f

AND

button1.width <= 300.0f

The relationship operator in the formula is no longer = (equals) but Greater Than or Equal in the first constraint and Less Than or Equal in the second. You change this by giving the `relatedBy:` parameter one of these values:

- `NSLayoutRelationEqual`
- `NSLayoutRelationGreaterThanOrEqual`
- `NSLayoutRelationLessThanOrEqual`

Run the app and... there's no change. The button is always 200 points wide, even in landscape. Apparently, having these two constraints is not enough. In fact, if you think about it, both constraints are satisfied perfectly when the width is exactly 200 points, so Auto Layout sees no reason to make the button wider when it doesn't need to.

One way to solve this problem is to put leading and trailing spaces between the button and the edges of the screen. In portrait mode, the screen is 320 points wide, so that leaves $(320 - 200)/2 = 60$ points on either side. Add the following constraints:

```
constraint = [NSLayoutConstraint  
constraintWithItem:button1  
attribute:NSLayoutAttributeLeading  
relatedBy:NSLayoutRelationEqual  
toItem:self.view
```

```
attribute:NSLayoutAttributeLeading  
multiplier:1.0f  
constant:60.0f];  
  
[self.view addConstraint:constraint];  
  
constraint = [NSLayoutConstraint  
constraintWithItem:button1  
attribute:NSLayoutAttributeTrailing  
relatedBy:NSLayoutRelationEqual  
toItem:self.view  
attribute:NSLayoutAttributeTrailing  
multiplier:1.0f  
constant:-60.0f];  
  
[self.view addConstraint:constraint];
```

Can you find out for yourself what the formulas are for these two constraints?

Here's what you should have found:

button1.leading = self.view.leadingAnchor * 1.0f + 60.0f

AND

button1.trailing = self.view.trailingAnchor * 1.0f - 60.0f

Remember that leading means "left" and trailing means "right" in a left-to-right language such as English, so this puts the button's left edge 60 points away from the screen's left border, and also leaves 60 points between the button's right edge and the screen's right border.

This setup works fine in portrait, but not in landscape. Try it out. Auto Layout gives an "Unable to simultaneously satisfy constraints" error when you flip to landscape. That makes sense, because if the button should be 300 points wide in landscape, then the margins should be $(480 - 300) / 2 = 90$ points, not 60.

Change the `relatedBy:` parameter on the leading space constraint to `NSLayoutRelationGreaterThanOrEqual` to make it a minimum of 60 points rather than an exact requirement.

Do the same for the trailing space constraint, but here make it `NSLayoutRelationLessThanOrEqual`. That's a little sneaky, but because the constant is negative (-60.0f), you want to allow it to become -90.0f, which obviously is smaller, not greater.

Note: If you were to place this constraint in Interface Builder, you'd use a greater-than 90 instead of less-than -90, because Interface Builder doesn't work with negative values. Don't let that confuse you!

Run the app, and – drat! – it still doesn't work. The button stays 200 points wide even in landscape. Again, Auto Layout has no incentive to increase the width of the button because a width of 200 still satisfies all requirements.

Here's the winning fix: change the "width <= 300" constraint to the following:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeWidth
    relatedBy:NSLayoutRelationEqual
    toItem:nil
    attribute:NSLayoutAttributeNotAnAttribute
    multiplier:1.0f
    constant:300.0f];

constraint.priority = 999;

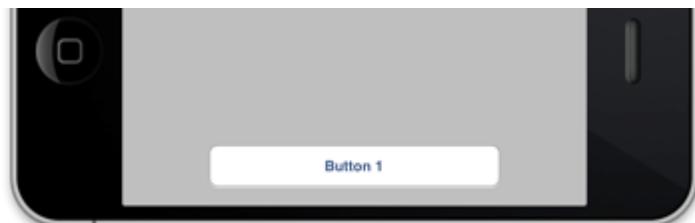
[button1 addConstraint:constraint];
```

The relationship is now "equal," so Auto Layout tries to force the button to be 300 points wide, but you have also made the constraint optional by lowering the priority to be less than 1,000 (the level at which it becomes required).

Auto Layout will try to apply this constraint but when it doesn't work – because there is not enough room for a 300-point wide button in portrait – it simply ignores this constraint. After all, you have made it optional.

The exact priority level doesn't matter, as long as it is lower than 1,000. Often when you set priorities, all that matters is what the priority values are relative to one another.

Run the app and check it out: the button grows to 300 points in landscape:



Note: The priority level of the constraint does matter to some extent. If you set it to 250 or lower, then the button's Content Hugging priority takes over.

Recall from the previous chapter that Content Hugging wants to keep the button at its optimal size, the intrinsic content size. You can think of this as an additional width constraint on the button that tries to keep the button from growing larger:

button1.width <= optimal size

This built-in constraint has priority 250, although you can change that with `UIView's setContentHuggingPriority:forAxis:` method. So if you set another width constraint on the button with a priority lower than 250, it has no effect because Content Hugging is considered more important.

Views also have a Content Compression Resistance priority that keeps the view from getting smaller. Think of this as a width constraint that says:

button1.width >= optimal size

The default priority for Content Compression Resistance is 750, but you can change it using the `setContentCompressionResistancePriority:forAxis:` method.

Can you simplify this?

You may wonder why you still need the "center X" constraint. After all, the constraints for the margins are equal on both sides of the button, so they should already cause the button to be centered, right?

Comment out the line that adds the Center X constraint (the very first one) to the view:

```
//[self.view addConstraint:constraint];
```

And run the app again. Guess what? After you rotate to landscape, the button is no longer centered. In fact, the layout is now *ambiguous*. You can see this by adding the following code above the `@implementation` section:

```
@interface UIWindow (AutoLayoutDebug)
+ (UIWindow *)keyWindow;
- (NSString *)_autolayoutTrace;
@end
```

And adding these two methods somewhere inside the `@implementation` body:

```

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSLog(@"%@", [[UIWindow keyWindow] _autolayoutTrace]);
}

- (void)didRotateFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
{
    [super didRotateFromInterfaceOrientation:
        fromInterfaceOrientation];

    NSLog(@"%@", [[UIWindow keyWindow] _autolayoutTrace]);
}

```

This calls the private `_autolayoutTrace` API to dump a debug trace of the view hierarchy to Xcode's output pane when the view becomes visible and whenever the device is rotated afterwards. It's handy to put this code snippet in your view controller when you're building and testing your layouts, but be sure to take it out for the final release build of your app.

Run the app and rotate to landscape. The output pane now says something like this:

```

*<UIWindow:0x8a56ff0>
|  *<UIView:0x7516d90>
|  |  *<UIRoundedRectButton:0x8a57e50> - AMBIGUOUS LAYOUT
|  |  |  <UIGroupTableViewCellBackground:0x8a58810>
|  |  |  <UIImageView:0x7517070>
|  |  |  <UIButtonLabel:0x7517e00>

```

There you have it: the button has an ambiguous layout. That explains why it is no longer horizontally centered.

There is more you can do to debug situations like these. Add the following line to `viewDidLoad`:

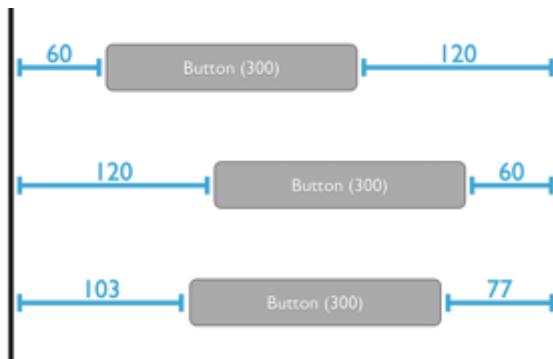
```

[button1 addTarget:button1
           action:@selector(exerciseAmbiguityInLayout)
forControlEvents:UIControlEventTouchUpInside];

```

An ambiguous layout means that Auto Layout has multiple ways to solve your constraints, but none of them is any good. Tapping on the button will now make Auto Layout cycle through the various options, so you can see which constraints are actually making this happen.

Run the app, flip to landscape, and tap the button. The button should now jump back and forth as Auto Layout tries out to find a layout that fits. It's obvious that something is wrong with the leading and trailing space constraints.



Remember that inequalities (the \geq and \leq constraints) by themselves are never enough; there must always be an $=$ constraint in there somewhere as well. Without the equality, Auto Layout doesn't know how to distribute the extra 60 points of space that it has. It can put them all on the right, all on the left, or a few points here and a few points there. There are an infinite number of solutions to these equations.

That is what the Center X constraint is for. It anchors the button in the middle, and now there is always exactly one solution. No more ambiguity. Enable the center constraint again, and the problem is gone.

Note: Instead of the Center X constraint, you could also have added two new constraints: a leading space between the superview and the button's left edge for ≤ 90 points, and a trailing space for the button's right edge that is ≥ -90 points. With those additional constraints in place, there is only one possible solution in landscape, and that is to make them both 90 points wide. There is often more than one way to skin a cat with Auto Layout!

The one constraint that you can remove safely is the "width ≥ 200 " constraint. It doesn't add anything that the others don't already take care of. The ≥ 60 and ≤ -60 spaces already force the button to be 200 points wide in portrait. Remove that constraint and verify that the app still works.

Tip: The fewer constraints you have, the better. With too many constraints, it becomes hard to figure out what's going on and to make changes. On the other hand, with too few constraints your layout can become ambiguous. As you can tell, building layouts with Auto Layout is a careful balancing act!

The possibilities are endless

So far the constraints you have made in code are no different than the ones you can make in Interface Builder. But the $A = B*m + c$ formula allows for many more interesting constructs. For example, what if you want the button to always be square?



It's hip to be square!

Interface Builder has no **Pin\Make Square** menu option, but you can easily express this relationship in code as:

button1.width = button1.height

Or equivalently:

button1.height = button1.width

To see how this works, add the following constraint to `viewDidLoad`:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeHeight
    relatedBy:NSLayoutRelationEqual
    toItem:button1
    attribute:NSLayoutAttributeWidth
    multiplier:1.0f
    constant:0.0f];

[button1 addConstraint:constraint];
```

This time the constraint expresses a relationship between the button and itself. Both the `constraintWithItem:` and `toItem:` parameters refer to `button1`.

Note that you could also have written it as follows, with the width and height attributes swapped:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeWidth
    relatedBy:NSLayoutRelationEqual
    toItem:button1
    attribute:NSLayoutAttributeHeight
    multiplier:1.0f
    constant:0.0f];
```

That makes no difference whatsoever to Auto Layout; it always picks the largest of the two dimensions, and makes the other one just as wide or tall.

Run the app to verify that it works.



In landscape orientation, the button is now taller than the screen. How to fix that? You can try adding a new constraint that says the button's height can be no more than 260 points. Add the following code:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeHeight
    relatedBy:NSLayoutRelationLessThanOrEqual
    toItem:nil
    attribute:NSLayoutAttributeNotAnAttribute
    multiplier:1.0f
    constant:260.0f];
```

```
[button1 addConstraint:constraint];
```

This is a constraint that just works on one attribute of the button, so `toItem:` is nil.

The approach works, except that if you look closer, it also reduces the width of the button to 260 points. That makes sense, because you do have a constraint on the button that says: **width = height**.

So what happened to the constraint that says, "I want the button to be 300 points wide" in landscape? Well, if you remember, that constraint has a lower priority (999 instead of 1,000), so "width equals height" takes precedence.

What if you do want the button to be 300 points in landscape, but square when possible? Easily done.

Whenever you think of a constraint as "...if possible," you need to use the priority system. Set the priority of the "width = height" constraint to 998 so that it is of less importance than the "width = 300" constraint (anything lower than 999 and higher than 250 will do):

```
constraint.priority = 998;
```

Now it works. The button is square in portrait, and as tall as it can be in landscape:



In a real app, you probably wouldn't put such constraints on a button, but for something such as a `UIImageView` or a container `UIView` it can be quite useful.

Go forth and multiply

Until now you have used the value 1.0 for the multiplier, but let's say you want the height of a view to be half its width. In that case you set the `multiplier:` parameter of the `NSLayoutConstraint` to 0.5f.

Try it out on the "width = height" constraint:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button1
    attribute:NSLayoutAttributeHeight
    relatedBy:NSLayoutRelationEqual
    toItem:button1
    attribute:NSLayoutAttributeWidth
    multiplier:0.5f
    constant:0];
```

```
    toItem:button1
    attribute:NSLayoutAttributeWidth
    multiplier:0.5f                         // change this
    constant:0.0f];

constraint.priority = 998;
```

The formula is now:

button1.height = button1.width * 0.5f

There you go. You can use this construct to keep the *aspect ratio* of a view intact, even if one of the dimensions of the view changes, which comes in handy for image views.

Suppose you have a photo-viewing app. In portrait the photo occupies the entire screen (including the status bar area). The aspect ratio here is 320:480, or 1:1.5 – that is, the height is always 1.5 times the width of the image.



If you set up the following constraints, then after rotating to landscape the photo will be centered horizontally but it keeps the same aspect ratio:

```
photoView.centerX = self.view.centerX
photoView.centerY = self.view.centerY
photoView.height = self.view.height
photoView.height = 1.5f * photoView.width
```

The last formula is the important one. It uses a multiplier of 1.5 to set the aspect ratio. If you don't do this, the dimensions of the photo will be distorted after rotation.

It looks like this in landscape. The height of the photoView is now 320 points and the width is 1.5 times as small, about 214 points:



And now I have another challenge for you - try to make this layout in a new project! You should be able to do this for yourself, using the things you have learned in the previous sections.

Note: If you get stuck, the resources for this chapter contain an example project, "AspectRatio", that shows you how to make this layout.

More buttons!

You have seen constraints between a view and its superview, and constraints that just apply to the view itself. But you haven't created constraints between two views at the same level of the view hierarchy programmatically yet. So let's try that out by adding another button into the mix (still back in the CodeConstraints project).

First add a new instance variable:

```
@implementation ViewController
{
    UIButton *button1;
    UIButton *button2;
}
```

And add the button creation code to `viewDidLoad`:

```
button2 = [UIButton buttonWithType:UIButtonTypeRoundedRect];
button2.translatesAutoresizingMaskIntoConstraints = NO;
[button2 setTitle:@"Button 2" forState:UIControlStateNormal];

[self.view addSubview:button2];
```

Run the app now, and you'll see that Button 2 appears half off the screen. That makes sense, because you haven't set any constraints on it yet. The `NSLog` output also points out this button has AMBIGUOUS LAYOUT.

In Interface Builder, when you drag one view close to another, it “snaps” into place and from then on, the two views are attached to each other using a Horizontal Space or Vertical Space constraint. If you do it from code, you have to describe this using the $A = B*m + c$ formula. For example:

button2.bottom = button1.top - 8.0f

Button 2 gets placed on top of Button 1, with a standard margin of 8 points between them.

Expressed as an `NSLayoutConstraint`, that looks like this:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button2
    attribute:NSLayoutAttributeBottom
    relatedBy:NSLayoutRelationEqual
    toItem:button1
    attribute:NSLayoutAttributeTop
    multiplier:1.0f
    constant:-8.0f];

[self.view addConstraint:constraint];
```

Add this code to the app and run it. That's better – at least vertically. In the horizontal dimension, the new button still has no constraints.



Note: You added the constraint between `button1` and `button2` to `self.view`, their common ancestor in the view hierarchy.

Here are the rules for where to add constraints:

- * **Between a view and its superview:** Add to the superview.
- * **Between two views with the same superview:** Add to the superview.
- * **On just the view:** Add to the view itself.

It is even possible to add constraints between views that do not have the same superview, i.e., that sit in different parts of the view hierarchy. In that case, add the constraint to the nearest ancestor view that they both have in common.

Where shall you place the second button horizontally? In Interface Builder, you had the Align menu with options such as Left Edges, Right Edges, Horizontal Centers, and so on. In code, you will have to come up with formulas that describe the same thing.

Let's try Align Left Edges first:

button2.left = button1.left

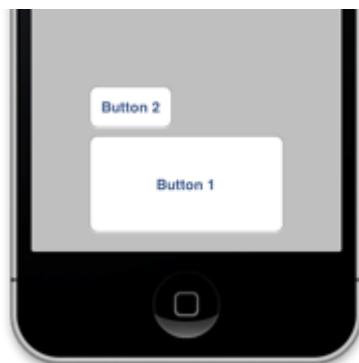
That should be pretty simple to express:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button2
    attribute:NSLayoutAttributeLeading
    relatedBy:NSLayoutRelationEqual
    toItem:button1
    attribute:NSLayoutAttributeLeading
    multiplier:1.0f
    constant:0.0f];

[self.view addConstraint:constraint];
```

Because you should always be building your interfaces with internationalization on your mind, you're using the "leading" attribute here instead of "left."

Run it and try it out.



The layout is no longer ambiguous because there are now enough constraints to determine the positions and sizes of both buttons.

Aligning the right edges is just as easy. Replace `NSLayoutAttributeLeading` with `NSLayoutAttributeTrailing` in the previous constraint, and run again.

Center alignment is not any harder than that: just use `NSLayoutAttributeCenterX`.

Of course, you don't need to give both attributes the same value. Try making the first attribute `NSLayoutAttributeCenterX` and the second `NSLayoutAttributeTrailing`.

Now you're saying: "The center of `button2` is at the same position as the right edge of `button1`":



You can even change the constant to add an offset to the alignment. Set the `constant:` parameter of the constraint to `-200.0f`. This makes it appear as if the center of Button 2 is above the left edge of Button 1, but that's just an illusion. Rotate to landscape to see what is really going on.

For more fun, change the multiplier to something like `0.75f` and set the constant back to `0.0f`. Now the center of Button 2 is 75% away from the button's right edge, whatever that means. It doesn't make much sense to do that, but it's fun to play with different values for these parameters. As you can see, this simple formula, $A = B*m + c$, is quite flexible and powerful.

Here's another thing you cannot do in Interface Builder: making the size of one view depend on the size of another. Add a new constraint:

```
constraint = [NSLayoutConstraint
    constraintWithItem:button2
    attribute:NSLayoutAttributeHeight
    relatedBy:NSLayoutRelationEqual
    toItem:button1
    attribute:NSLayoutAttributeHeight
    multiplier:0.5f
    constant:-10.0f];

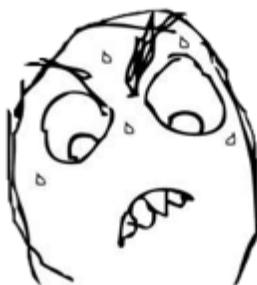
[self.view addConstraint:constraint];
```

Button 2 is now half as high as Button 1, minus an additional 10 points. The possibilities for making ugly user interfaces are endless!

Notice how nowhere in this source file are you saying: “The buttons are positioned at these coordinates,” or “their frames are this big.” All you’re doing is describing how the views are related to each other and to the super view. Again, that is what design-by-intent is. You just describe *what* you want and let Auto Layout figure out *how* to deliver it.

What about the call to `[button1 sizeToFit]`, doesn’t that set the frame size of the button? Actually, that call is no longer necessary with Auto Layout. The button already has an intrinsic content size and if you don’t put any constraints on the button that force it to have an explicit width or height, Auto Layout will automatically use this intrinsic content size.

Remove the call to `sizeToFit` and the app should still work as before.



I NEED A BREAK...

You’ve accomplished a lot so far – you now know how to create constraints programatically and should have a good understanding of the simple formula that makes this all work.

You’re about to move onto some new material, so this is a good time to have a cup of coffee, feed the fish or walk the dog. Toilet breaks are also permitted. ☺ If everything so far made sense, then you’re off to a good start already, but most of the journey is still ahead of you!

The Visual Format Language

You should have a pretty good feel for creating constraints from code by now, but you have to admit, it’s also a lot of typing! For every single constraint you need to do at least:

```
NSLayoutConstraint *constraint = [NSLayoutConstraint
    constraintWithItem:view1
    attribute:NSLayoutAttributeTrailing
    relatedBy:NSLayoutRelationEqual
    toItem:view2
    attribute:NSLayoutAttributeLeading]
```

```
multiplier:1.0f  
constant:20.0f];  
  
[self.view addConstraint:constraint];
```

That adds up to a lot of code really quickly.

It's also easy to make mistakes: is it **trailing = leading + 20**, or **leading = trailing - 20**, or what? You'll notice those mistakes quickly enough when you run the app, but it can still be a bit of a hassle to work them out.

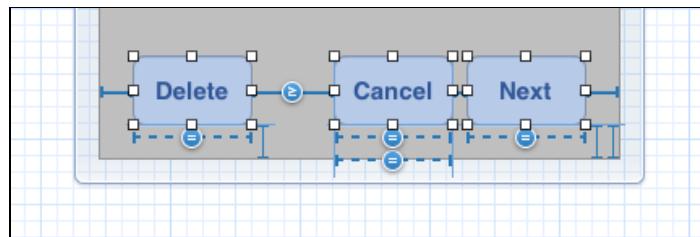
Expressing constraints directly using the formula is very powerful, but with great power also comes great responsibility.

Fortunately, the designers of Auto Layout came up with a cool shortcut for specifying constraints. They call it the Visual Format Language, but really that's just a fancy way of saying drawing using... ASCII art!

```
/@  
\\ \  
__> \  
(____) \---  
(____)_  
(____)  
(____)/---
```

Remember the Artist Details app from last chapter? Don't worry, you won't have to go through all of that again. However, I do want to explain how to make the row of buttons from that app using constraints and the Visual Format Language. It will be easier than you think!

To jog your memory, the buttons at the bottom were created using the following constraints:



I'm counting nine different constraints to keep those buttons in place. Without the Visual Format Language, you'd also have to create nine `NSLayoutConstraint` objects. That's simply too much work for a lazy programmer!

So let's learn an easier way to do it. ☺

Create a new project using the Single View Application template and name it "VisualArtistDetails." Disable storyboards, but enable Automatic Reference Counting.

Before you go any further, first add the debugging code to **ViewController.m**:

```
@interface UIWindow (AutoLayoutDebug)
+ (UIWindow *)keyWindow;
- (NSString *)_autolayoutTrace;
@end
```

And:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSLog(@"%@", [[UIWindow keyWindow] _autolayoutTrace]);
}

- (void)didRotateFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
{
    [super didRotateFromInterfaceOrientation:
        fromInterfaceOrientation];

    NSLog(@"%@", [[UIWindow keyWindow] _autolayoutTrace]);
}
```

This is just a preventative measure. I like to have this tracing code in there so that it's easy to spot ambiguous layouts. Interface Builder works hard to prevent you from making mistakes, but when you're building up your constraints programmatically, you can do all sorts of things wrong.

Sometimes your layout may even appear to work just fine, but then you do an `_autolayoutTrace` and it actually says AMBIGUOUS LAYOUT. You want to catch such errors before you ship your app to customers. It's better to be on the safe side, especially when you're just starting out with Auto Layout.

Add instance variables for the three buttons:

```
@implementation ViewController
{
    UIButton *deleteButton;
    UIButton *cancelButton;
    UIButton *nextButton;
}
```

And for now just create the Delete button:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    deleteButton = [UIButton buttonWithType:
                    UIButtonTypeRoundedRect];
    deleteButton.translatesAutoresizingMaskIntoConstraints = NO;
    [deleteButton setTitle:@"Delete"
                  forState:UIControlStateNormal];

    [self.view addSubview:deleteButton];
}

```

Run the app, and the Delete button is nowhere to be found. The Xcode output pane says something like this:

```

<UIWindow:0x718e150>
|  <UIView:0x8875960>
|  |  <UIRoundedRectButton:0x88754c0>
|  |  |  <UITableViewCellBackground:0x88749b0>
|  |  |  <UIImageView:0x88a8180>
|  |  |  <UIButtonLabel:0x88a9490>

```

So the button is definitely there. You just can't see it. Let's add some constraints so that Auto Layout kicks in. Remember, Auto Layout is not activated unless you add at least one constraint.

Add the following code to viewDidLoad:

```

NSDictionary *viewsDictionary =
    NSDictionaryOfVariableBindings(deleteButton);

NSArray *constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:@"|-[_deleteButton]"
    options:0
    metrics:nil
    views:viewsDictionary];

[self.view addConstraints:constraints];

```

This is the new way of creating constraints, with the ASCII-art style Visual Format Language.

First note that this method returns an array of constraints instead of a single constraint like before. This is because it may take more than one constraint in order to resolve the condition that you write with the Visual Format Language.

Also, instead of passing a ton of parameters to the method, you define the constraint with the following visual format string:

```
@ " |-[deleteButton]"
```

If you leave out the quotes for the string, it becomes a piece of primitive ASCII art:

```
|-[deleteButton]
```

With a little imagination, you can see that the | bar represents the left edge of the superview, [deleteButton] is the actual button, and - is a horizontal space between the two. The horizontal space is 20 points, the standard size as determined by the HIG (Apple's Human Interfaces Guidelines).

Build and run the app, and you'll see the following:



The debug output still says the layout is ambiguous. True, because you have only specified constraints for the horizontal dimension, not for the vertical, which is why the button is half off-screen.

Add these new lines of code:

```
constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:@ "V: |-[deleteButton]"
    options:0
    metrics:nil
    views:viewsDictionary];

[self.view addConstraints:constraints];
```

The format string is now:

```
V: |-[deleteButton]
```

It is identical to the previous string, except it has V: in front. That means this format string is for the vertical direction. Now the | bar represents the top edge of the superview.

Note: If you don't put a v: in front, then the format string is assumed to be for a horizontal constraint. You can also specify H: for a horizontal constraint, like H: |-[deleteButton], but that's not strictly necessary.

Run the app, and now the button is lodged in the upper-left corner of the view.

To see the actual constraints that were generated, you can simply `NSLog()` the arrays:

```
- (void)viewDidLoad
{
    . . .

    NSArray *constraints = . . .;

    NSLog(@"Horizontal constraints: %@", constraints);

    [self.view addConstraints:constraints];

    constraints = . . .;

    NSLog(@"Vertical constraints: %@", constraints);

    [self.view addConstraints:constraints];
}
```

Run the app and look at the output pane:

```
Horizontal constraints: (
    "<NSLayoutConstraint:0x756cea0 H:|-(NSSpace(20))-"
    [UIRoundedRectButton:0x7568e70]    (Names: '|':UIView:0x76629c0 )>"
)

Vertical constraints: (
    "<NSLayoutConstraint:0xff58ae0 V:|-(NSSpace(20))-"
    [UIRoundedRectButton:0x7568e70]    (Names: '|':UIView:0x76629c0 )>"
)
```

Both arrays currently contain just a single constraint, but that will change when you learn to use more complex format strings.

When you dump a constraint, it contains a representation of that constraint in the Visual Format Language. The first one says:

```
H:|-(NSSpace(20))-[UIRoundedRectButton:0x7568e70]
```

And the second one:

```
v: |-(NSSpace(20))-[UIRoundedRectButton:0x7568e70]
```

The description is a bit more specific than the format string you used. It points out the size of the horizontal space (20 points), and the actual pointer to the button. That can be helpful with debugging. The full debug message also shows which `UIView` the `|` bars represent.

Tip: As you progress through this chapter, keep an eye on the way that UIKit logs the constraints. Knowing how to read these format strings is an invaluable debugging tool.

Let's move the button to the bottom-right corner. Replace the format string for the horizontal constraint with:

```
[deleteButton]-|
```

Now the `|` bar represents the window's right edge. Do the same thing for the vertical constraint:

```
v:[deleteButton]-|
```

Run the app and now the button sits in the opposite corner of the screen:



What about centering the button so that it sits in the bottom center? You may be tempted to do this:

```
|-[deleteButton]-|
```

Try it out. That will stretch the button to be as wide as the screen, minus the margins at the edges. The button is certainly centered, but it's not quite what I had in mind.



What if you remove the dashes for the horizontal spaces and do:

```
| [deleteButton] |
```

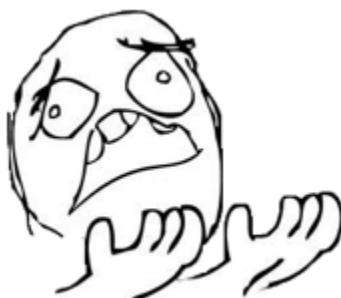
That's still no good. Even though you didn't put any dashes into the format string, this still inserts horizontal space constraints between the edges of the superview and the button, but of size 0. The button will now fill up the entire width of the screen.



What about just:

```
[deleteButton]
```

No edges, no spaces. Sounds reasonable, right? Unfortunately, it doesn't work. This will not give the button a horizontal constraint at all, because there is nothing to connect it to (you can see this in the `NSLog` output).



I have to admit that I'm leading you on a wild goose chase here. I'm sorry to say that there isn't a way to express that you want a view to be centered in its

superview using the Visual Format Language. You will need to use the long form of the `NSLayoutConstraint` constructor for that. The visual language can save you a lot of time, but it's not a complete replacement for the magic formula.

Adding the Cancel and Next buttons

Let's return to the thing that we set out to do: replicating the row of buttons from the Artist Details app. Return the horizontal format string to:

```
|-[deleteButton]
```

This is so that the button now sits in the bottom-left corner. That's where it is supposed to be, so you're done with the Delete button for the moment.

Add the code to create the Cancel and Next buttons to the top of `viewDidLoad`:

```
cancelButton = [UIButton buttonWithType:  
                           UIButtonTypeRoundedRect];  
cancelButton.translatesAutoresizingMaskIntoConstraints = NO;  
[cancelButton setTitle:@"Cancel"  
               forState:UIControlStateNormal];  
  
[self.view addSubview:cancelButton];  
  
nextButton = [UIButton buttonWithType:  
                           UIButtonTypeRoundedRect];  
nextButton.translatesAutoresizingMaskIntoConstraints = NO;  
[nextButton setTitle:@"Next" forState:UIControlStateNormal];  
  
[self.view addSubview:nextButton];
```

Notice that you have to set `translatesAutoresizingMaskIntoConstraints` to `NO` on every single button that you create. Don't forget it!

In the original design, the Next button is attached to the right side of the screen, and the Cancel button is snapped to the Next button using a standard space of 8 points. You can express this as follows (add this to the bottom of `viewDidLoad`):

```
constraints = [NSLayoutConstraint  
    constraintsWithVisualFormat:  
        @"[cancelButton]-[nextButton]-|"  
    options:0  
    metrics:nil  
    views:viewsDictionary];  
  
[self.view addConstraints:constraints];
```

The visual format string is:

```
[cancelButton]-[nextButton]-|
```

This is where the Visual Format Language becomes interesting: you can describe the relationships between different views with a single command. This will create two constraints: a 20-point space between the Next button and the screen border, and an 8-point space between the Cancel button and the Next button.

Of course, you also need to specify what happens in the vertical direction. Both buttons sit at the standard margin from the bottom:

```
constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:@"V:[nextButton]-| "
    options:0
    metrics:nil
    views:viewsDictionary];

[self.view addConstraints:constraints];

constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:@"V:[cancelButton]-| "
    options:0
    metrics:nil
    views:viewsDictionary];

[self.view addConstraints:constraints];
```

In this case, you can't combine the two buttons in a single format string because that would stack them on top of each other, which is not the effect you're looking for here.

Run the app to try it out. Yikes! A big fat crash is your reward:

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
reason: 'Unable to parse constraint format:
cancelButton is not a key in the views dictionary.
[cancelButton]-[nextButton]-|
^'
```

What's going on here? When you put something like `[cancelButton]` in the visual format string, Auto Layout doesn't automatically know what you mean by "cancelButton." This so happens to be the same name as your instance variable, but Auto Layout will not make any assumptions about that. Instead, you have to tell it what object the name "cancelButton" refers to.

So why did this work earlier with the Delete button? How did Auto Layout know which `UIButton` object you meant with `[deleteButton]`?

Here's the trick. One of the lines near the top of `viewDidLoad` says this:

```
NSDictionary *viewsDictionary =
    NSDictionaryOfVariableBindings(deleteButton);
```

When you created the constraints, you passed this dictionary object into the `constraintsWithVisualFormat` method:

```
NSArray *constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:@"|[deleteButton]|"
    options:0
    metrics:nil
    views:viewsDictionary]; // it goes here
```

Auto Layout looks into that dictionary for a key named "deleteButton" and expects that to refer to the correct `UIButton` object. `NSDictionaryOfVariableBindings()` is a macro that creates a new `NSDictionary` that maps the pointer from the `deleteButton` variable to the name "deleteButton."

Any view that you want to refer to by name in the format string, you have to add to this dictionary. So change the line to:

```
NSDictionary *viewsDictionary =
    NSDictionaryOfVariableBindings(
        deleteButton, cancelButton, nextButton);
```

You could also have written:

```
NSDictionary *viewsDictionary = @{
    @"deleteButton": deleteButton,
    @"cancelButton": cancelButton,
    @"nextButton": nextButton
};
```

But that gets tiring fast. Using the `NSDictionaryOfVariableBindings()` macro is a lot easier, and you can simply reuse the same dictionary for setting all the constraints.

Run the app again and it should work. You now have three buttons in a row:



It can always be simpler still

The code is already simpler than when you used the long form of `NSLayoutConstraint`'s constructor to make all the constraints individually, but you can still do better by unleashing the full power of the Visual Format Language.

Notice that `cancelButton` and `nextButton` both have their own vertical constraint? That is not really necessary if you declare that Cancel and Next are always aligned by their tops (or bottoms, it doesn't really matter). Then you just need to connect one of them to the bottom of the screen, not both.

You don't need to write much extra code for this. Instead, replace the horizontal constraint with:

```
constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:
        @"[cancelButton]-[nextButton]-|"
    options:NSLayoutFormatAlignAllTop
    metrics:nil
    views:viewsDictionary];
```

Previously, the `options:` parameter was 0 but now it is `NSLayoutFormatAlignAllTop`. That means you want all the views that are specified in this format string to be top-aligned. Now remove the `v:[cancelButton]-|` statement, so that the `nextButton` is the only one connected to the bottom of the screen.

Run the app to verify that it still works.

There is more that can be done. The Delete button has its own format string, but really it sits on the same line as the other two. So you should be able to put all three buttons into one format string. Try this one:

```
|-[deleteButton]-[cancelButton]-[nextButton]-|
```

You can remove all other format strings, until you're left with just these two:

```
NSDictionary *viewsDictionary =
    NSDictionaryOfVariableBindings(
        deleteButton, cancelButton, nextButton);

NSArray *constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:
        @"|-[deleteButton]-[cancelButton]-[nextButton]-|"
    options:NSLayoutFormatAlignAllTop
    metrics:nil
    views:viewsDictionary];
```

```
NSLog(@"Horizontal constraints %@", constraints);

[self.view addConstraints:constraints];

constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:@"V:[nextButton]-| "
    options:0
    metrics:nil
    views:viewsDictionary];

NSLog(@"Vertical constraints %@", constraints);

[self.view addConstraints:constraints];
```

That's right, for this particular layout you just need to specify two format strings, one for the horizontal direction and one for the vertical direction. This is enough to give constraints to all the buttons.

Run the app. Of course, it doesn't quite work as it should – all the buttons are now AMBIGUOUS according to the `NSLog` output – but it's a start.



When you try this, it may not be the Delete button that is wider than the others – it could be Cancel or Next. That's the whole point: because there is more than one possible solution – none of which is really good enough – Auto Layout will randomly pick one.

The reason this layout is ambiguous is that the buttons want to be just as big as their ideal sizes – no more, no less – but the constraints between them won't let this happen. As a result, at least one of the buttons has to grow, but Auto Layout does not know which one to pick.

To see the ambiguity in action, add the following lines to `viewDidLoad`:

```
[deleteButton addTarget:deleteButton
    action:@selector(exerciseAmbiguityInLayout)
    forControlEvents:UIControlEventTouchUpInside];
```

```
[cancelButton addTarget:cancelButton  
    action:@selector(exerciseAmbiguityInLayout)  
    forControlEvents:UIControlEventTouchUpInside];  
  
[nextButton addTarget:nextButton  
    action:@selector(exerciseAmbiguityInLayout)  
    forControlEvents:UIControlEventTouchUpInside];
```

Tapping the buttons will now make Auto Layout cycle through the possible layouts it has calculated. Note that the `exerciseAmbiguityInLayout` method only works on the views that are actually ambiguous, so for example if you were to send this message to `self.view` instead, it has no effect.

Run the app again and tap the buttons. They take turns being the biggest:



Because the layout is ambiguous, Auto Layout cannot decide which button should be the largest. All options are equally valid.

Note: What if that is exactly the sort of layout you want, where one of the buttons takes up all the space while the other two stay at their ideal size? Is there no way to get rid of this ambiguity, so that you can say: "I always want the Delete button to grow to fill up the remaining space"? Of course there is!

Remember Content Hugging and Compression Resistance? All three buttons currently want to stay at their ideal size, but if you play with these priorities

you can make one button's Content Hugging less important than the others.
Try this:

```
[deleteButton setContentHuggingPriority:249  
    forAxis:UILayoutConstraintAxisHorizontal];
```

This tells Auto Layout that the Delete button doesn't care as much about keeping its size as the others, because its Content Hugging priority is lower – 249 instead of the default 250. The layout is no longer ambiguous, because by lowering this priority you have given Auto Layout permission to stretch the Delete button.

In the Interface Builder design from last chapter, you put a Greater Than or Equals space between the Delete and Cancel buttons, in order to keep the Cancel button from overlapping the Delete button if the text in the Next button became larger. This horizontal space made sure there was always at least 8 points of space between the two buttons. That's great and we want to keep it that way.

However, in the current format string, the space between Delete and Cancel is a *fixed* space of 8 points, not a flexible one, and that is what causes the problem. Fortunately, the Visual Format Language allows you to specify inequalities too.

Change the format string to:

```
|-[deleteButton]-(>=8)-[cancelButton]-[nextButton]-|
```

The new bit is `-(>=8)-`. This is called the *predicate* and it tells Auto Layout that the space between Delete and Cancel should be at least 8 points big. In other words, it creates a Greater Than or Equal constraint with its constant set to 8.0f.

Run the app and now it should work. The three buttons are in their proper positions again:



The difference from before is that now it only took a single visual format string to make this happen, instead of two separate ones.

Now that you've got a small taste of predicates, let's see what else you can do with them!

LEARN ALL ABOUT
PREDICATES...



...AND OTHER FANCY
WORDS

Playing with predicates

Predicates are pretty cool because they let you customize the sizes of the constraints between the views. For example, change the format string to:

```
| -5-[deleteButton]-(>=8)-[cancelButton]-30-[nextButton]-|
```

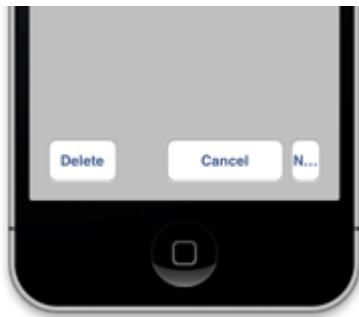
The margin between the screen border and the Delete button is now reduced to 5 points, and the space between the Cancel and Next buttons is 30 points instead of the default 8.



You can also give views a fixed width by adding predicates inside the square brackets:

```
| -[deleteButton]-(>=8)-[cancelButton(120)]-[nextButton(30)]-|
```

This adds a **width = 120** constraint on the Cancel button and a **width = 30** constraint on the Next button.



Of course, this works in the vertical direction as well. Change the other format string to:

```
V:[nextButton(100)]-50-
```

Obviously this only changes the Next button, but since the other two are aligned to its top, they shift up as well.



You usually want to align controls that contain text by their *baseline*, rather than by their top or bottom. The baseline is the imaginary line that the text sits on – that's usually the important bit that should look properly aligned.

In the statement that makes the horizontal constraints, change the `NSLayoutFormatAlignAllTop` setting to `NSLayoutFormatAlignAllBaseline` and run the app again.



Now the buttons are aligned by the vertical position of their text labels. For a button, this is very similar to `NSLayoutFormatAlignAllCenterY` because the label is

always vertically centered, but if you want to align a button with a segmented control, for example, then `NSLayoutFormatAlignAllBaseline` is what you want to use.

OK, enough fooling around. Put the format strings back to:

```
|-[deleteButton]-(>=8)-[cancelButton]-[nextButton]-|
```

And:

```
v:[nextButton]-|
```

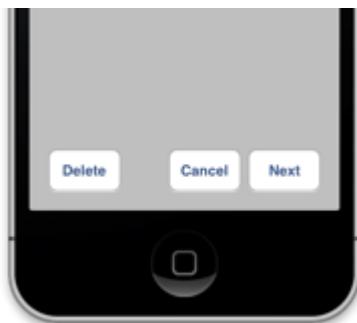
The one thing you haven't replicated yet from the original Interface Builder design is to give all buttons the same width. In Interface Builder this was easy with the **Pin\Widths Equally** menu option. Well, with the Visual Format Language it's not much harder:

```
|-[deleteButton(==nextButton)]-(>=8)-  
 [cancelButton(==nextButton)]-[nextButton]-|
```

Do note, the above goes on a single line in the source code.

And that's it. Rather than hardcode the width like you did before with `(120)`, you use the `(==nextButton)` predicate to tell Auto Layout that both the `deleteButton` and `cancelButton` should have the same width as the `nextButton`.

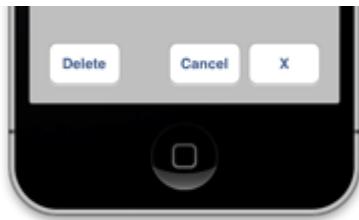
Run the app, and all the buttons should now have equal widths.



Just because you said `[deleteButton(==nextButton)]`, that doesn't mean that the Delete button will become very small if the Next button title is very short. Add this to the end of `viewDidLoad`:

```
[nextButton setTitle:@"X" forState:UIControlStateNormal];
```

And run the app. The buttons are still big enough to fit all the text:



The notation `[deleteButton(==nextButton)]` simply puts an Equal Widths constraint between the two. It means both these buttons will have the same size, or in other words:

deleteButton.width = nextButton.width

BUT ALSO

nextButton.width = deleteButton.width

You could also have written `[nextButton(==deleteButton)]` and it would have worked exactly the same. Auto Layout figures out which button is largest and gives the others the same size.

So what happens when the Next button gets a very long title? Pretend the user has switched to the Dutch localization and change the title of the Next button to "Volgende":

```
[nextButton setTitle:@"Volgende"  
forState:UIControlStateNormal];
```

The app still does what it should do:



Also translate the Delete button:

```
[deleteButton setTitle:@"Verwijderen"  
forState:UIControlStateNormal];
```

Hmm, that's not ideal. The text doesn't quite fit into the button anymore:



The way you solved this in Interface Builder was to give the Equal Widths constraints a lower priority. By lowering the priority you say, "I'd like these buttons to have the same width, if possible." If it's no longer possible, Auto Layout is free to ignore that constraint.

Setting priorities is also possible with the Visual Format Language:

```
|-[deleteButton(==nextButton@700)]-(>=8)-  
    [cancelButton(==nextButton@700)]-[nextButton]-|
```

Again, all of the above goes on one line in the source code.

By adding @700 after the predicate, you change its priority. Run the app and now the Equal Widths constraint is broken in portrait because there isn't enough room for it, but it's still honored in landscape.



And that's all you need to do with the Visual Format Language to layout these buttons. Just two statements to set up the constraints: one for horizontal, one for vertical. Once you understand how the format works, it's quicker than doing it in Interface Builder. And hey, it's definitely easier than reading regular expressions! ☺

If you look at the `NSLog` output, you'll see the nine constraints for these three buttons:

```
Horizontal constraints (
    "<NSLayoutConstraint:0x7177530 H:|-(NSSpace(20))-[UIRoundedRectButton:0x7170f50]   (Names:
' | ':UIView:0x7171440 )>",
```

```
"<NSLayoutConstraint:0x71772b0 UIRoundedRectButton:0x7170f50.width ==  
UIRoundedRectButton:0x71755a0.width priority:700>"
```

```
"<NSLayoutConstraint:0x71776f0 H:[UIRoundedRectButton:0x7170f50]-(>=8)-[UIRoundedRectButton:0x7174eb0]>",

"<NSLayoutConstraint:0x71777c0 UIRoundedRectButton:0x7174eb0.width == UIRoundedRectButton:0x71755a0.width priority:700>",

"<NSLayoutConstraint:0x7177870 H:[UIRoundedRectButton:0x7174eb0]-(NSSpace(8))- [UIRoundedRectButton:0x71755a0]>",

"<NSLayoutConstraint:0x71778f0 H:[UIRoundedRectButton:0x71755a0]-(NSSpace(20))- | (Names:
' | ':UIView:0x7171440 )>",

"<NSLayoutConstraint:0x7177830 UIRoundedRectButton:0x7170f50.baseline == UIRoundedRectButton:0x7174eb0.baseline>",

"<NSLayoutConstraint:0x71778b0 UIRoundedRectButton:0x7174eb0.baseline == UIRoundedRectButton:0x71755a0.baseline>"

)

Vertical constraints (
    "<NSLayoutConstraint:0x717a0a0 V:[UIRoundedRectButton:0x71755a0]-(NSSpace(20))- | (Names:
' | ':UIView:0x7171440 )>"
)
```

Nice, nine constraints made with a single API call!

You should be able to decipher this now. There are four horizontal space constraints:

- 20 points space between the screen's left edge and the Delete button, `NSSpace(20)`.
- At least 8 points space between the Delete and Cancel buttons, `(>=8)`.
- Exactly 8 points space between the Cancel and Next buttons, `NSSpace(8)`.
- Exactly 20 points space between the Next button and the screen's right edge, another `NSSpace(20)`.

There are also two Equal Widths constraints with priority 700, and two constraints that align the buttons at their baselines.

Finally, there is a single vertical constraint that forces a space of 20 points between the Next button and the bottom of the screen.

Enter the metrics

You may have wondered what the `metrics:` parameter is for in the method `constraintsWithVisualFormat:options:metrics:views:.` So far you have always passed `nil` for this parameter, but you can also pass it an `NSDictionary` object.

The purpose of the `metrics:` parameter is similar to the `views` dictionary, except that the `NSDictionary` contains values for the constants that appear in the visual format string. This allows you to make your layouts a bit more flexible, so you don't have to hardcode any numbers the way you did in the previous section.

For example, if you want to change the spacing of the vertical distance between the buttons and the bottom of the window, you already know that you can change:

V:[nextButton]-|

Into:

V:[nextButton]-50-|

But you can also do this:

V:[nextButton]-distance-|

Here, "distance" is a key in the metrics dictionary. In code, this looks like:

```
constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:@"V:[nextButton]-distance- |"
    options:0
    metrics:@{ @"distance": @50 }
    views:viewsDictionary];
```

The `NSDictionary` itself is specified by:

`@{ @"distance": @50 }`

The key is the name of the constant as it appears in the format string, `@"distance"`, and the value is an `NSNumber` object. Of course you can have more than one key-value pair in the dictionary. You probably won't use this a lot, but if you have dynamic designs where the lengths of the spaces can change, then this is a useful feature.

This concludes the introduction to the Visual Format Language. I'm sure you'll agree with me that using the visual format strings to set these constraints is certainly a lot less work than setting them up individually!

Note: The Visual Format Language is pretty cool, but it does have its limitations. For example, you cannot do `[deleteButton(==nextButton*2)]` to make one button twice as big as the other. Of course, it is perfectly simple to

do this with the $A = B*m + c$ formula. So, sometimes you have no choice but to create your constraints the long way.

Time for another pit stop! In fact, I think I have to go myself. ☺ You're at the halfway point so I hope you're enjoying the ride. Refuel, clear your head, and we'll continue when you're ready.

Dynamic layouts

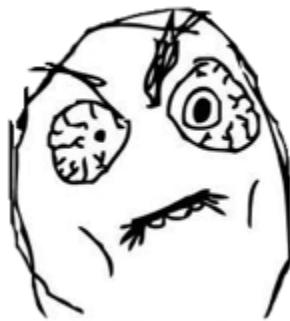
The examples you've seen so far were all pretty static. The only change to the UI happened when you rotated from portrait to landscape, or when the text on the buttons or labels changed. Many apps need a bit more flexibility than that – you may need to show or hide controls, or create new views on-the-fly, often with animations.

To build such dynamic screens with Auto Layout, you will need to destroy and create constraints while the app is running. That's not something you can do from Interface Builder; you will have to get down and dirty with `NSLayoutConstraint`. This section shows you how to do that.

Create a new project based on the Single View Application template and name it "DynamicLayout." No storyboards, but enable ARC.

You will now add three buttons to the screen.

MORE BUTTONS?!?!



I know, I know, you're probably getting fed up with buttons, but they're a useful tool for studying Auto Layout. They have a label that can change size, which gives the button an intrinsic content size (and therefore Content Hugging and Content Compression Resistance priorities). Best of all, you can tap buttons to make them do things.

Open **ViewController.m** and add new instance variables:

```
@implementation ViewController
```

```
{  
    UIButton *leftButton;  
    UIButton *centerButton;  
    UIButton *rightButton;  
}
```

As usual, create these buttons in viewDidLoad:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    leftButton = [UIButton buttonWithType:  
                           UIButtonTypeRoundedRect];  
    leftButton.translatesAutoresizingMaskIntoConstraints = NO;  
    [leftButton setTitle:@"Left" forState:UIControlStateNormal];  
    [self.view addSubview:leftButton];  
  
    centerButton = [UIButton buttonWithType:  
                           UIButtonTypeRoundedRect];  
    centerButton.translatesAutoresizingMaskIntoConstraints = NO;  
    [centerButton setTitle:@"Center"  
                  forState:UIControlStateNormal];  
    [self.view addSubview:centerButton];  
  
    rightButton = [UIButton buttonWithType:  
                           UIButtonTypeRoundedRect];  
    rightButton.translatesAutoresizingMaskIntoConstraints = NO;  
    [rightButton setTitle:@"Right"  
                  forState:UIControlStateNormal];  
    [self.view addSubview:rightButton];  
}
```

Your goal is to arrange these buttons as follows:



The Center button has two constraints that center it on the screen, horizontally and vertically. The Left button is attached to the Center button on one side, and the Right button is attached to it on the other side.

First add the constraints to position the Center button. You cannot do this with the Visual Format Language, so do it old school style:

```
NSLayoutConstraint *constraint = [NSLayoutConstraint
    constraintWithItem:centerButton
    attribute:NSLayoutAttributeCenterX
    relatedBy:NSLayoutRelationEqual
    toItem:self.view
    attribute:NSLayoutAttributeCenterX
    multiplier:1.0f
    constant:0.0f];

[self.view addConstraint:constraint];

constraint = [NSLayoutConstraint
    constraintWithItem:centerButton
    attribute:NSLayoutAttributeCenterY
    relatedBy:NSLayoutRelationEqual
    toItem:self.view
    attribute:NSLayoutAttributeCenterY
    multiplier:1.0f
    constant:0.0f];
```

```
[self.view addConstraint:constraint];
```

Now that the Center button is in place, add the constraints that connect the Left and Right buttons to it:

```
NSDictionary *viewsDictionary =
    NSDictionaryOfVariableBindings(
        leftButton, centerButton, rightButton);

NSArray *constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:
        @"[leftButton]-[centerButton]-[rightButton]"
    options:NSLayoutFormatAlignAllBaseline
    metrics:nil
    views:viewsDictionary];

[self.view addConstraints:constraints];
```

Notice that the format string is:

```
[leftButton]-[centerButton]-[rightButton]
```

There are no | bars in here for the superview's borders, but that's OK. This will work because `centerButton` already has the Center X and Y constraints that anchor it into place.

Finally, also add the following for debugging purposes:

```
@interface UIWindow (AutoLayoutDebug)
+ (UIWindow *)keyWindow;
- (NSString *)_autolayoutTrace;
@end
```

And:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSLog(@"%@", [[UIWindow keyWindow] _autolayoutTrace]);
}

- (void)didRotateFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
{
    [super didRotateFromInterfaceOrientation:
        fromInterfaceOrientation];
```

```
    NSLog(@"%@", [[UIWindow keyWindow] _autolayoutTrace]);
}
```

None of this should be too surprising; it's all stuff you have seen before.

Run the app. Your buttons should appear and there should be no "AMBIGUOUS LAYOUT" messages in the Xcode output pane. Huzzah!

Now what do you think will happen if the Center button is hidden so that it is no longer visible on the screen? No idea? Then let's try it out and see what happens.

Add this line to `viewDidLoad`:

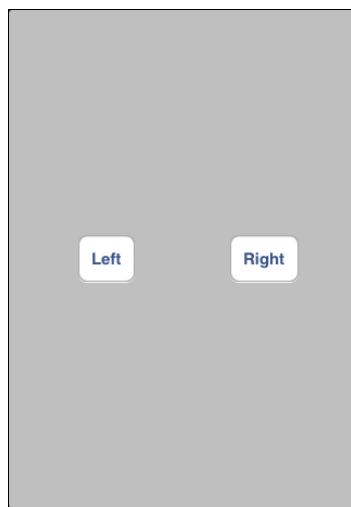
```
[leftButton addTarget:self
                  action:@selector(leftButtonPressed)
            forControlEvents:UIControlEventTouchUpInside];
```

Also add the method that will handle the button press:

```
- (void)leftButtonPressed
{
    centerButton.hidden = !centerButton.hidden;
}
```

This simply toggles the `hidden` property of the Center button between `YES` and `NO`.

Run the app and tap the Left button. The Center button disappears – but the constraints stay in place. The visibility of a button apparently has no influence on the constraints.



What if, after the Center button has disappeared, we want the Left and Right buttons to sit side-by-side like this:

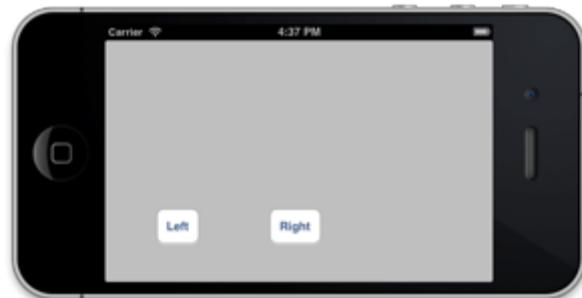


To make that happen, you have to remove the Center button from the screen. That automatically removes any related constraints and Auto Layout will perform its calculations again.

Change `leftButtonPressed` to:

```
- (void)leftButtonPressed
{
    if (centerButton.superview != nil)
        [centerButton removeFromSuperview];
    else
        [self.view addSubview:centerButton];
}
```

Run the app and tap the Left button. The Center button disappears as it should and everything seems as it was before. However, rotate the device to landscape and you'll see that the layout is now broken:



The buttons are not where they are supposed to be and the debug output screams, “AMBIGUOUS LAYOUT!” That’s not so strange, because all constraints have now disappeared. You can see this for yourself if you add `NSLog()` statements to the method:

```
- (void)leftButtonPressed
{
    NSLog(@"Before: constraints = %@", self.view.constraints);

    if (centerButton.superview != nil)
        [centerButton removeFromSuperview];
    else
        [self.view addSubview:centerButton];

    NSLog(@"After: constraints = %@", self.view.constraints);
}
```

Run the app and tap Left. The debug output says:

```
Before: constraints = (
    "<NSLayoutConstraint:0x7664ca0 UIRoundedRectButton:0x7663970.centerX =="
    "UIView:0x765ff30.centerX>",

    "<NSLayoutConstraint:0x7665580 UIRoundedRectButton:0x7663970.centerY =="
    "UIView:0x765ff30.centerY>",

    "<NSLayoutConstraint:0x7665e80 H:[UIRoundedRectButton:0x765fa40]-(NSSpace(8))-"
    "[UIRoundedRectButton:0x7663970]>",

    "<NSLayoutConstraint:0x7665f20 H:[UIRoundedRectButton:0x7663970]-(NSSpace(8))-"
    "[UIRoundedRectButton:0x7664060]>",

    "<NSLayoutConstraint:0x7665ec0 UIRoundedRectButton:0x765fa40.baseline =="
    "UIRoundedRectButton:0x7663970.baseline>",

    "<NSLayoutConstraint:0x7666200 UIRoundedRectButton:0x7663970.baseline =="
    "UIRoundedRectButton:0x7664060.baseline>"

)

After: constraints = ()
```

Because the Center button is no longer part of the view hierarchy, and the constraints for the Left and Right buttons were attached to the Center button, all those constraints were removed. Afterwards, `self.view.constraints` is an empty array.

In order to fix the view hierarchy after the Center button is gone, you need to put some new constraints into place:

```
- (void)leftButtonPressed
{
    NSLog(@"Before: constraints = %@", self.view.constraints);

    if (centerButton.superview != nil)
    {
        [centerButton removeFromSuperview];

        NSLayoutConstraint *constraint = [NSLayoutConstraint
            constraintWithItem:leftButton
            attribute:NSLayoutAttributeTrailing
            relatedBy:NSLayoutRelationEqual
            toItem:self.view
            attribute:NSLayoutAttributeCenterX
            multiplier:1.0f
            constant:-10.0f];

        [self.view addConstraint:constraint];

        constraint = [NSLayoutConstraint
            constraintWithItem:leftButton
            attribute:NSLayoutAttributeCenterY
            relatedBy:NSLayoutRelationEqual
            toItem:self.view
            attribute:NSLayoutAttributeCenterY
            multiplier:1.0f
            constant:0.0f];

        [self.view addConstraint:constraint];
    }

    constraint = [NSLayoutConstraint
        constraintWithItem:rightButton
        attribute:NSLayoutAttributeLeading
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterX
        multiplier:1.0f
        constant:10.0f];

    [self.view addConstraint:constraint];

    constraint = [NSLayoutConstraint
        constraintWithItem:rightButton
```

```
        attribute:NSLayoutAttributeCenterY
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterY
        multiplier:1.0f
        constant:0.0f];

    [self.view addConstraint:constraint];
}
else
{
    [self.view addSubview:centerButton];
}

NSLog(@"After: constraints = %@", self.view.constraints);
}
```

These new constraints place the Left button 10 points left of the horizontal center, and the Right button 10 points to the right of the center. Both buttons are also centered vertically.

Run the app and tap the Left button. That's better! Flip the app to landscape to make sure that the buttons stay centered. The debug output confirms that the new constraints are there:

```
After: constraints = (
    "<NSLayoutConstraint:0x758c380 UIRoundedRectButton:0x71823b0.trailing =="
    "UIView:0x7182900.centerX - 10>",

    "<NSLayoutConstraint:0x758c660 UIRoundedRectButton:0x71823b0.centerY =="
    "UIView:0x7182900.centerY>",

    "<NSLayoutConstraint:0x758cbc0 UIRoundedRectButton:0x7186a50.leading =="
    "UIView:0x7182900.centerX + 10>",

    "<NSLayoutConstraint:0x758d300 UIRoundedRectButton:0x7186a50.centerY =="
    "UIView:0x7182900.centerY>"
)
```

A second tap of the Left button will add the Center button back into the screen:



Obviously that button has no constraints set on it anymore, so you need to remove the existing constraints from the Left and Right buttons and add the original ones back again. You could do that inside the else-clause in `leftButtonPressed`, but it would duplicate a bunch of code from `viewDidLoad`.

It is better to centralize your constraints logic into `UIViewController`'s `updateViewConstraints` method. This method is called when the view controller first becomes visible and every time afterwards when the constraints need to change.

Add this method to the file:

```
- (void)updateViewConstraints
{
    NSLog(@"updateViewConstraints");

    [super updateViewConstraints];

    [self.view removeConstraints:self.view.constraints];

    if (centerButton.superview != nil)
    {
        NSLayoutConstraint *constraint = [NSLayoutConstraint
            constraintWithItem:centerButton
            attribute:NSLayoutAttributeCenterX
            relatedBy:NSLayoutRelationEqual
            toItem:self.view
            attribute:NSLayoutAttributeCenterX
            multiplier:1.0f
            constant:0.0f];

        [self.view addConstraint:constraint];

        constraint = [NSLayoutConstraint
            constraintWithItem:centerButton
            attribute:NSLayoutAttributeCenterY
            relatedBy:NSLayoutRelationEqual
            toItem:self.view
            attribute:NSLayoutAttributeCenterY
            multiplier:1.0f
```

```
    constant:0.0f];

    [self.view addConstraint:constraint];

    NSDictionary *viewsDictionary =
        NSDictionaryOfVariableBindings(
            leftButton, centerButton, rightButton);

    NSArray *constraints = [NSLayoutConstraint
        constraintsWithVisualFormat:
            @"[leftButton]-[centerButton]-[rightButton]"
        options:NSLayoutFormatAlignAllBaseline
        metrics:nil
        views:viewsDictionary];

    [self.view addConstraints:constraints];
}

else
{
    NSLayoutConstraint *constraint = [NSLayoutConstraint
        constraintWithItem:leftButton
        attribute:NSLayoutAttributeTrailing
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterX
        multiplier:1.0f
        constant:-10.0f];

    [self.view addConstraint:constraint];

    constraint = [NSLayoutConstraint
        constraintWithItem:leftButton
        attribute:NSLayoutAttributeCenterY
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterY
        multiplier:1.0f
        constant:0.0f];

    [self.view addConstraint:constraint];

    constraint = [NSLayoutConstraint
        constraintWithItem:rightButton
        attribute:NSLayoutAttributeLeading
        relatedBy:NSLayoutRelationEqual
```

```
        toItem:self.view
        attribute:NSLayoutAttributeCenterX
        multiplier:1.0f
        constant:10.0f];

    [self.view addConstraint:constraint];

    constraint = [NSLayoutConstraint
        constraintWithItem:rightButton
        attribute:NSLayoutAttributeCenterY
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterY
        multiplier:1.0f
        constant:0.0f];

    [self.view addConstraint:constraint];
}

}
```

You have seen most of this code before in `viewDidLoad` and `leftButtonPressed`. The only new line is:

```
[self.view removeConstraints:self.view.constraints];
```

That is necessary to remove the constraints from the Left and Right buttons when the Center button becomes visible again. Otherwise the old constraints will conflict with the new ones and Auto Layout will throw a fit.

Note: When you override `updateViewConstraints` you always need to call `[super updateViewConstraints]`. Don't forget it or the boogeyman will come get you!

With all the logic for making the constraints squared away in `updateViewConstraints`, the `leftButtonPressed` method can be a lot simpler now:

```
- (void)leftButtonPressed
{
    if (centerButton.superview != nil)
        [centerButton removeFromSuperview];
    else
        [self.view addSubview:centerButton];

    [self.view setNeedsUpdateConstraints];
```

```
}
```

As before, this method first changes the view hierarchy by either removing the Center button or adding it back. Then it calls `setNeedsUpdateConstraints` to tell the view that its constraints need to be updated. In response, UIKit calls the view controller's `updateViewConstraints` method.

You can also simplify `viewDidLoad`. It only needs to create the buttons:

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

    leftButton = [UIButton buttonWithType:
                  UIButtonTypeRoundedRect];
    leftButton.translatesAutoresizingMaskIntoConstraints = NO;
    [leftButton setTitle:@"Left" forState:UIControlStateNormal];
    [leftButton addTarget:self
                      action:@selector(leftButtonPressed)
            forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:leftButton];

    centerButton = [UIButton buttonWithType:
                  UIButtonTypeRoundedRect];
    centerButton.translatesAutoresizingMaskIntoConstraints = NO;
    [centerButton setTitle:@"Center"
                  forState:UIControlStateNormal];
    [self.view addSubview:centerButton];

    rightButton = [UIButton buttonWithType:
                  UIButtonTypeRoundedRect];
    rightButton.translatesAutoresizingMaskIntoConstraints = NO;
    [rightButton setTitle:@"Right"
                  forState:UIControlStateNormal];
    [self.view addSubview:rightButton];

    [self.view setNeedsUpdateConstraints];
}
```

Again, `viewDidLoad` calls `setNeedsUpdateConstraints`, which in turn causes this method to be called to add the initial constraints to the views.

Note: Remember that Auto Layout does not become active until you add at least one constraint. Because you removed the code from `viewDidLoad` that

adds the constraints, you have to trigger adding constraints manually by calling `setNeedsUpdateConstraints`.

Try it out. Tapping Left once removes the Center button and tapping it again puts the Center button back.

Congrats! You have just made a dynamic layout that changes its constraints on-the-fly. The cool thing is that the entire layout logic now lives in a single method, `updateViewConstraints`.

If you had to do this without Auto Layout, you would have to calculate the new frames of the buttons yourself, but now you only have to describe their relationships and let Auto Layout do the hard work for you.

Making it move with animations

Adding and removing constraints during runtime is fine, but since this is iOS, users expect cool animations as well. ☺ Fortunately, you can animate Auto Layout pretty easily.

Add the following to the bottom of `leftButtonPressed`:

```
[UIView animateWithDuration:0.3f animations:^{
    {
        [self.view layoutIfNeeded];
    }];
}];
```

That's really all you need to do to make the changes in the constraints animate. Run the app and try it out!

If you delete old constraints and create new ones, you need to call `[self.view layoutIfNeeded]` in the animation block. UIKit, Core Animation, and Auto Layout take care of the rest.

However, it's not the prettiest of animations. The Center button pops in and out of view without any subtlety. It would be nicer if the button faded in and out instead. To pull that off, you need to set the alpha of `centerButton` to 0 inside the animation block, but that will only work if you don't remove `centerButton` until *after* the animation completes.

This change in logic requires a new instance variable that keeps track of whether or not the `centerButton` is visible. Add it to the class:

```
@implementation ViewController
{
    UIButton *leftButton;
    UIButton *centerButton;
    UIButton *rightButton;
```

```
    BOOL buttonIsVisible;  
}
```

Initially the Center button is visible indeed, so set this variable to YES at the top of viewDidLoad, so that it is YES when updateViewConstraints is called for the first time:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    buttonIsVisible = YES;  
  
    . . .  
}
```

Inside updateViewConstraints, change the if-statement to:

```
if (buttonIsVisible)  
{  
    . . .
```

It no longer checks whether centerButton has a superview or not, because that will cease to work in light of the changes you're going to make to leftButtonPressed:

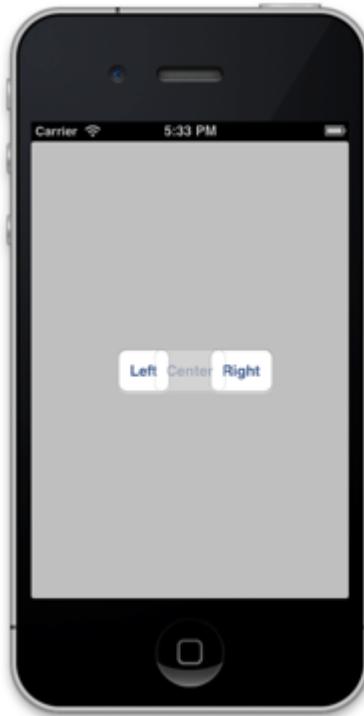
```
- (void)leftButtonPressed  
{  
    buttonIsVisible = !buttonIsVisible; // 1  
  
    if (buttonIsVisible) // 2  
    {  
        [self.view addSubview:centerButton];  
        centerButton.alpha = 0.0f;  
    }  
  
    [self.view setNeedsUpdateConstraints]; // 3  
  
    [UIView animateWithDuration:0.3f animations:^  
    {  
        [self.view layoutIfNeeded]; // 4  
        centerButton.alpha = buttonIsVisible ? 1.0f : 0.0f; // 5  
    }  
    completion:^(BOOL finished)  
    {
```

```
    if (!buttonIsVisible)
        [centerButton removeFromSuperview]; // 6
    }];
}
```

This method still functions the same as before, except that the order of things is slightly different:

18. Toggle the value of the `buttonIsVisible` variable. If it was `YES`, then it becomes `NO`, and vice versa.
19. If the button must be made visible again, then add the `centerButton` to the main view again, but set its `alpha` property to 0 so it is invisible.
20. Mark the constraints as needing an update. Note: this does not actually call `updateViewConstraints` right away. Instead, UIKit schedules this operation for some time in the future (for the next pass through the run loop).
21. In the animation block, call `layoutIfNeeded` to force Auto Layout to recalculate its layout right now. Because you did `setNeedsUpdateConstraints` to mark the constraints as being out-of-date, this first calls `updateViewConstraints` to delete the old constraints and add the new ones, and then it recalculates the new frames of the views. It's important that this recalculation happens in the animation block and not sooner because it represents the destination state of the animation.
22. Set the new `alpha` value for the Center button. If the button is becoming visible, the new `alpha` is `1.0f`, so that the button fades in during the animation. Conversely, if `buttonIsVisible` is `NO`, the new `alpha` is `0.0f`, so that the button fades out.
23. In the animation's completion block, which is performed after the animation has finished, the Center button is finally removed from the superview. At this point it is already invisible because its `alpha` is `0.0f`.

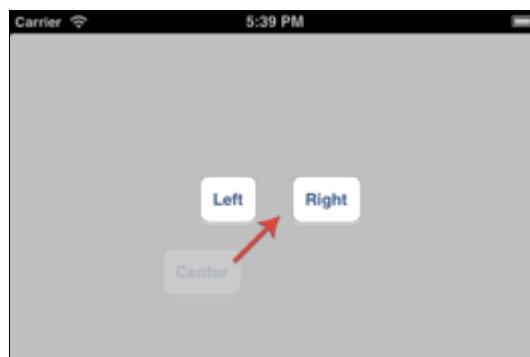
Run the app and try it out. You have to agree, the animation is a lot smoother now.



Note: During the animation, the layout is actually “wrong” for a little while. Because `centerButton` is only removed from the screen after the animation completes, there are no constraints to keep it in place during the animation. That’s usually not a problem since it’s only temporary. Just make sure that after the animation is done everything is in order again.

Naturally, there is a small problem with this code – hey, there is always a small problem somewhere, right? ☺

Run the app. Tap the Left button to make the Center button disappear. Rotate to landscape and tap the Left button again. Strangely enough, the Center button now appears to fly “into” position.



What happens here is that the Center button still has its old frame set to what it was in portrait. This view is no longer part of the view hierarchy and has no constraints, so it cannot auto-rotate along with the rest of the view elements, and it keeps its old position and size.

You can fix this in several ways. One way is to always keep the Center button in the view hierarchy and simply hide it. You never remove the button or the constraints that keep it centered, but simply set its `hidden` property to `YES`. That way the button still plays a part in the layout calculations, but users don't see it. Give it a shot and try this out for yourself!

There is another solution – immediately after adding the Center button back into the view hierarchy, you can add constraints to properly position it and then call `layoutIfNeeded` to force Auto Layout to calculate the correct frame for the button.

To make this happen, change the if-statement in `leftButtonPressed` to:

```
if (buttonIsVisible)
{
    [self.view addSubview:centerButton];
    centerButton.alpha = 0.0f;

    NSLayoutConstraint *constraint = [NSLayoutConstraint
        constraintWithItem:centerButton
        attribute:NSLayoutAttributeCenterX
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterX
        multiplier:1.0f
        constant:0.0f];

    [self.view addConstraint:constraint];

    constraint = [NSLayoutConstraint
        constraintWithItem:centerButton
        attribute:NSLayoutAttributeCenterY
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterY
        multiplier:1.0f
        constant:0.0f];

    [self.view addConstraint:constraint];

    [self.view layoutIfNeeded];
```

```
    }  
  
    . . .
```

This all happens before the animation starts, so the Center button will be in the correct place before it becomes visible again.

It is important that you do not forget to call `layoutIfNeeded` after adding the constraints. Without this, Auto Layout will not recalculate the layout before the animation happens and the Center button will still be in the wrong place.

Run the app and try it again. Now the Center button always reappears at the true center of the screen, even after you change the orientation of the device.

Note: There is an important difference between the methods `layoutIfNeeded` and `setNeedsLayout`, just as there is a difference between `displayIfNeeded` and `setNeedsDisplay`, and the methods `updateConstraintsIfNeeded` and `setNeedsUpdateConstraints`.

The “xxx if needed” version will perform the action right then and there, while “set needs xxx” will delay it to whenever UIKit has a spare moment. In this case, you want to perform the layout calculations immediately to give the button a proper frame before the animation starts, so `layoutIfNeeded` is the one to call.

One other thing before we continue. There is some code duplication going on here because you now create the Center X and Center Y constraints for the Center button in two places, once in `leftButtonPressed` and once in `updateViewConstraints`.

If you’re like me and want to save a few lines of code, you may be tempted to place the call to `layoutIfNeeded` in `updateViewConstraints` instead. After all, that method already adds these two center constraints to the button, so why not do this:

```
- (void)updateViewConstraints  
{  
    NSLog(@"updateViewConstraints");  
  
    [super updateViewConstraints];  
  
    [self.view removeConstraints:self.view.constraints];  
  
    if (buttonIsVisible)  
    {  
        // Add the center X and Y constraints  
        // . . .
```

```
[self.view layoutIfNeeded];  
  
    // Add the [leftButton]-[centerButton]-[rightButton]  
    // constraints . . .  
}  
. . .
```

As it turns out, this may work and it may not. When I tried this solution, sometimes the Center button ended up in the wrong position anyway. The UIKit documentation also explicitly says:

"You may not invoke a layout or drawing phase as part of your constraint update phase."

So it's better to stay safe and do this sort of thing outside `updateViewConstraints`.

When the constant is not constant

There are two ways to animate Auto Layout-based layouts:

1. By removing existing constraints and adding new ones.
2. By animating the "constant" value for the constraint objects.

An `NSLayoutConstraint` is immutable. Once you've created it you cannot change it, except for the constant. That's the `c` from the formula: $A = B*m + c$. You can set the `constant` property of an `NSLayoutConstraint` object to a new value and then start an animation block.

If you want to change a constraint at some later point, you need to keep a reference to that object, so add a new instance variable:

```
@implementation ViewController  
{  
    . . .  
    NSLayoutConstraint *centerYConstraint;  
}
```

In `updateViewConstraints`, store the reference to the Center Y constraint into this new variable:

```
- (void)updateViewConstraints  
{  
    . . .  
  
    if (buttonIsVisible)  
    {
```

```
    . . .

    constraint = [NSLayoutConstraint
        constraintWithItem:centerButton
        attribute:NSLayoutAttributeCenterY
        relatedBy:NSLayoutRelationEqual
        toItem:self.view
        attribute:NSLayoutAttributeCenterY
        multiplier:1.0f
        constant:0.0f];

    centerYConstraint = constraint; // add this

    [self.view addConstraint:constraint];

    . . .
}

else
{
    . . .

    centerYConstraint = nil;
}
}
```

Just to make sure you're not modifying an unused constraint when the Center button is invisible, you also set this instance variable to `nil` in the else-section.

In `viewDidLoad`, hook up a method to respond to taps on the Right button:

```
[rightButton addTarget:self
                  action:@selector(rightButtonPressed)
            forControlEvents:UIControlEventTouchUpInside];
```

And finally, add that method:

```
- (void)rightButtonPressed
{
    if (centerYConstraint.constant == 0.0f)
        centerYConstraint.constant = 100.0f;
    else
        centerYConstraint.constant *= -1.0f;

    [UIView animateWithDuration:0.5f animations:^{
        {
            [self.view layoutIfNeeded];
        }
    }];
}
```

```
    }];
}
```

What this does is pretty simple: it changes the `constant` property of the `centerYConstraint` object, and then creates an animation block that calls `layoutIfNeeded`. That's all you need to do.

Run the app and tap the Right button a few times to see what this does.

Notice that tapping the Right button has no effect if the Center button is hidden. In that case there is no Center Y constraint and the variable `centerYConstraint` is nil.

Note: After flipping to landscape, the buttons are centered again. Why is that? Look at the Xcode output pane: the `NSLog()` for `updateViewConstraints` gets triggered every time you rotate the device. That means new constraints are created and `centerYConstraint` now points to a new `NSLayoutConstraint` object with a constant of `0.0f`. That's not a problem for this app, but it's still something to be aware of if you override `updateViewConstraints` in your own apps.

$$A = B*m + c$$

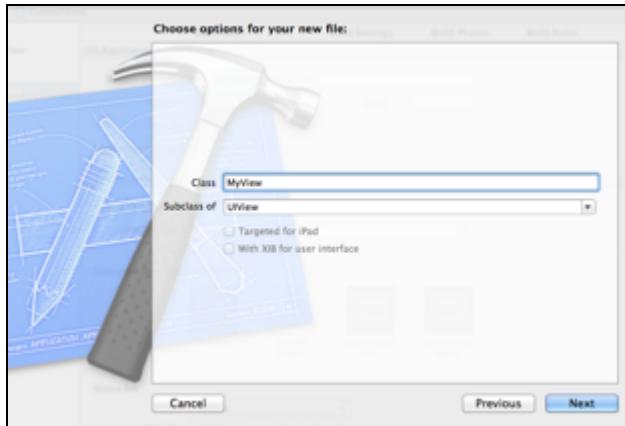


Making your own custom views

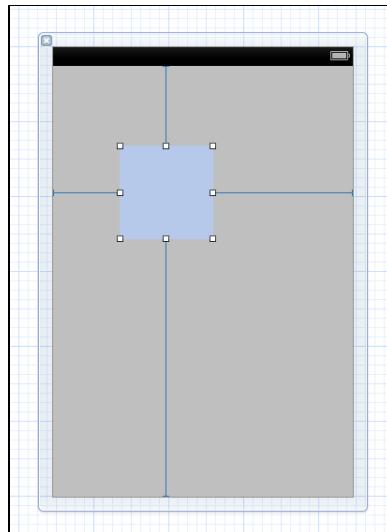
In this section you will explore custom views and intrinsic content size a bit further. You've already seen that buttons, labels, image views, and several of the other built-in controls have an intrinsic content size, which means that the view knows what its ideal size is, based on its content. You can also use this feature in any custom views you create, and this section shows you how. It involves going back to our old friend, Interface Builder.

Create a new project using the Single View Application template and name it "CustomView." No storyboards, but enable Automatic Reference Counting.

Add a new file to the project using the Objective-C class template. Name it "MyView," subclass of `UIView`.

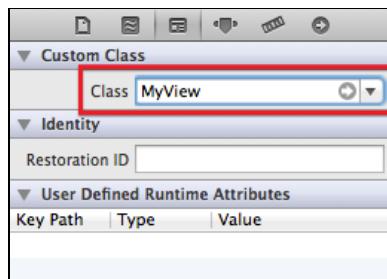


Open **ViewController.xib** and drag a new View object onto the nib. Scale it down to 100 by 100 points.



Notice that Interface Builder puts four space constraints around the view in order to keep it in place. A plain `UIView` does not have an intrinsic content size.

Select the new view and go to the Identity inspector. Set the Class field to "MyView."



You're going to give `MyView` an intrinsic content size that grows every time the user taps the view.

In `MyView.m`, add a new instance variable that keeps track of the view's current size:

```
@implementation MyView
{
    CGSize mySize;
}
```

This variable needs to have an initial value, and the view's `initWithCoder:` method is a good place to set that. You can remove the `initWithFrame:` method that was put there by the Xcode template. Also add a gesture recognizer that listens to taps on the view:

```
- (id) initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        mySize = CGSizeMake(30, 30);

        UITapGestureRecognizer *recognizer =
            [[UITapGestureRecognizer alloc] initWithTarget:self
                                              action:@selector(viewTapped:)];
        [self addGestureRecognizer:recognizer];
    }
    return self;
}
```

The view's size is reported by the method `intrinsicContentSize`. Whenever Auto Layout needs to redo its layout, this is the method it calls to determine how big the view is.

So implement the `intrinsicContentSize` method to simply return the value from the `mySize` variable:

```
- (CGSize) intrinsicContentSize
{
    return mySize;
}
```

Finally, add the method that will handle the taps:

```
- (void) viewTapped:(UITapGestureRecognizer *)recognizer
{
    mySize.width += 30.0f;
```

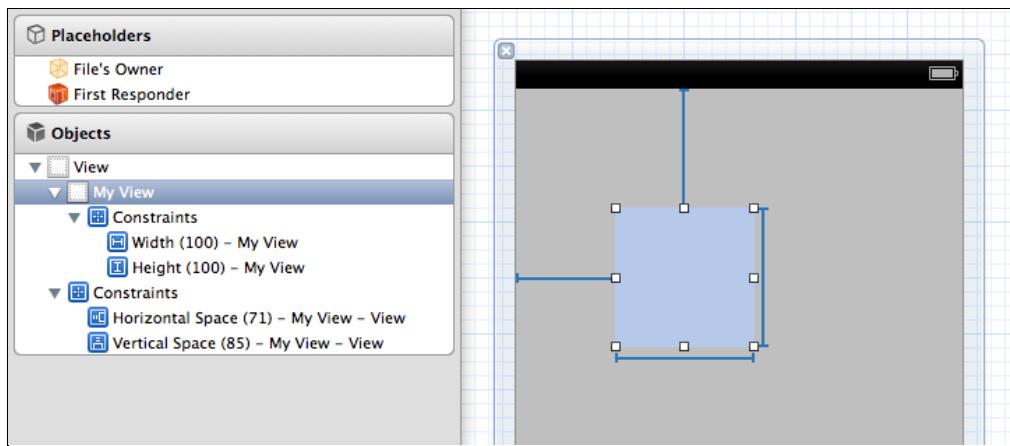
```
mySize.height += 30.0f;  
  
[self invalidateIntrinsicContentSize];  
}
```

It is important that whenever the size of the view changes, you let Auto Layout know about this by calling `invalidateIntrinsicContentSize`. In response, Auto Layout will redo its calculations and update the layout.

Run the app and tap the view. Does it work? No, not at all – the view stays at its original size.

The problem is the constraints that are currently set on the view in the nib. Unfortunately, Interface Builder is not smart enough to recognize that `MyView` now uses an intrinsic content size and therefore does not need that many constraints. Somehow you need to get rid of them, but Interface Builder won't let you...

Open **ViewController.xib**. Select `MyView` and choose **Pin\Width** from the menu. Select the view again and choose **Pin\Height**. You're doing this to force the view to have a specific width and height, so that you can safely remove the horizontal space to the screen border on the right, and the vertical space to the bottom. Do that, so that the only constraints remaining are these:



Of course, giving the view a fixed width and height still doesn't make the intrinsic content size work. But it gives you a bit more control over what Interface Builder does when you drag that view around.

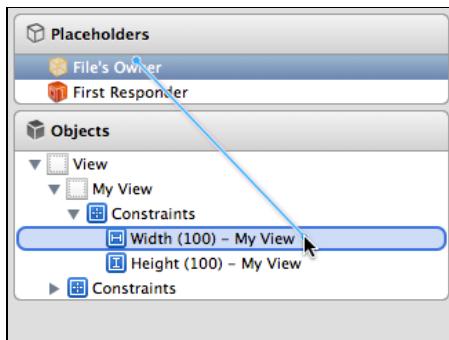
The important thing is that `MyView` will always keep its Width and Height constraints, so you can depend on them being there, no matter what other constraints Interface Builder decides to insert on its own.

You know by now that constraints are real objects of type `NSLayoutConstraint`. Because they are objects, you can also make outlet properties for them.

Add these property declarations to **ViewController.m**:

```
@interface ViewController : UIViewController
@property (nonatomic, strong) IBOutlet UIView *myView;
@property (nonatomic, strong) IBOutlet NSLayoutConstraint *widthConstraint;
@property (nonatomic, strong) IBOutlet NSLayoutConstraint *heightConstraint;
@end
```

In the nib, connect the `myView` outlet to the view (Ctrl-drag from File's Owner). Also connect the Width and Height constraints to their respective outlets (the easiest way to connect the constraints is to use the Document Outline, as shown below).



The only reason you're connecting these two constraints to outlets is so that the view controller can remove them again. Interface Builder won't just let you remove these constraints because it tries to keep the layout valid at all times, but in code you can do whatever you want.

Change `viewDidLoad` in **ViewController.m** to remove the Width and Height constraints from `MyView`:

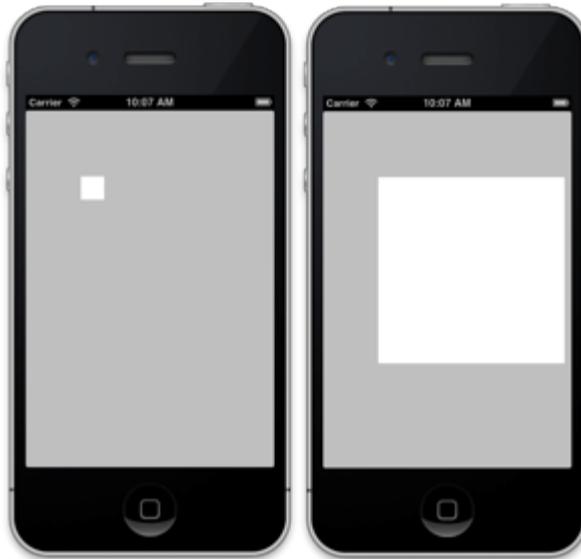
```
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSLog(@"Constraints before: %@", self.myView.constraints);

    [self.myView removeConstraint:self.widthConstraint];
    [self.myView removeConstraint:self.heightConstraint];

    NSLog(@"Constraints after: %@", self.myView.constraints);
}
```

Run the app again. Immediately you can see that the view is smaller, just 30 by 30 points instead of the 100x100 from the nib.



The Xcode output pane also shows that the width and height constraints have been removed:

```
Constraints before: (
    "<NSLayoutConstraint:0x71470f0 H:[MyView:0x7147e40(100)]>",
    "<NSLayoutConstraint:0x7147d20 v:[MyView:0x7147e40(100)]>"
)
Constraints after: (
)
```

Tap the view to see it grow. Its top-left corner stays in place because those constraints didn't change.

For fun, disconnect one of the outlets from the nib, for example the Height one. This prevents it from being removed in `viewDidLoad` because now the `heightConstraint` property doesn't point at anything.

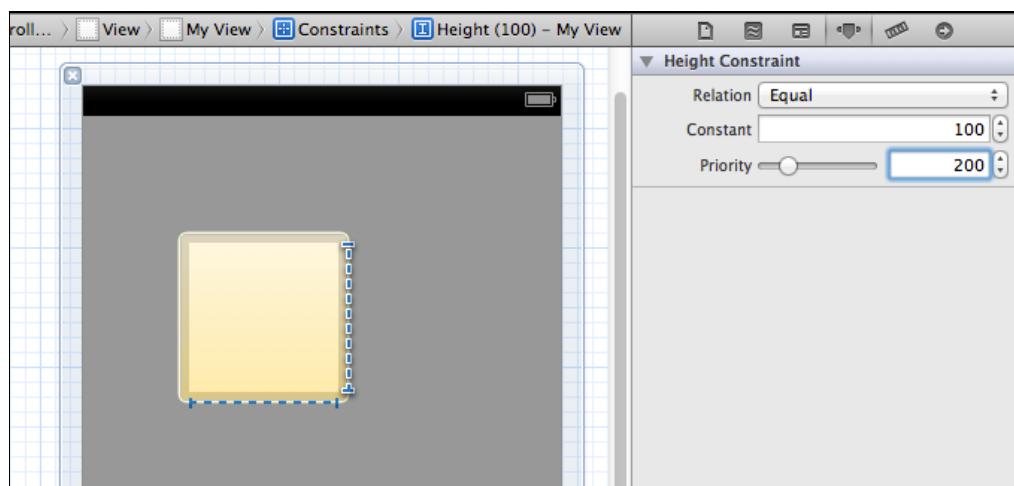
Run the app again. The custom view will only grow horizontally, because the Height constraint keeps the vertical dimension the same.



Does this seem like a lot of work? That's because it is! ☺

But there are other ways to skin an app (or a view). Remember Content Hugging? If you set the priority of the Width and Height constraints lower than the Content Hugging priority, then the intrinsic content size is respected even with those constraints on the control. That's a lot easier!

In the nib, disconnect the outlet for the Width and Height constraints. Also remove the code from `viewDidLoad` so that the view controller no longer tries to remove these constraints. In the nib, set the priority of both the Width and Height constraints to 200:



Try it out. Even though you gave `MyView` a fixed width and height in Interface Builder, during runtime its intrinsic content size takes over.

If you set the priority of the Width and Height constraints higher than 250 (the default priority value of Content Hugging) but lower than 750, the Content Compression Resistance priority, then `MyView` will not shrink any smaller than 100x100 points, but it will grow if the intrinsic size becomes larger than that.

Try it out! Set the priority of these two constraints to 700. Now it takes a few more taps, but eventually `mySize` is larger than 100x100 and then the view is allowed to grow.

Currently the intrinsic size of `MyView` changes both the width and height of the view, but if you just want your view to have an intrinsic content size in a single dimension, you can return a special value from `intrinsicContentSize`.

For example, if you only want to calculate the width depending on your content, then return `UIViewNoIntrinsicMetric` for the height component:

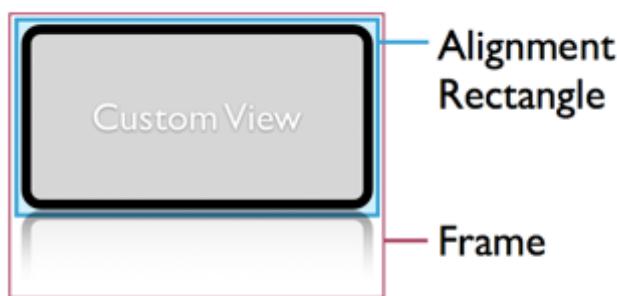
```
- (CGSize)intrinsicContentSize
{
    return CGSizeMake(mySize.width, UIViewNoIntrinsicMetric);
}
```

In that case, Auto Layout will always use the value of the constraints to calculate the height, but the width is still determined by `MyView` itself.

The alignment rectangle

If your custom view also draws ornamentation in addition to its content, such as a border, shadows, reflections or badges, then its frame may be larger than its actual content area. Auto Layout wants to work on just the content area, not the entire frame. Usually you don't care about the extra ornaments, you just want to position the content. That is what the *alignment rectangle* is for.

There can be a difference between a view's frame, which encompasses everything, and its alignment rectangle, which covers just the content. Auto Layout uses only the alignment rectangle in its calculations. You can think of the frame as adding extra padding around the alignment rectangle.



`UIView` has methods to convert between frame and alignment rectangles, `alignmentRectForFrame:` and `frameForAlignmentRect:`. To add extra padding around your custom view's content area, you can override these two methods in your `UIView`-based subclass, but it's easier to override the `alignmentRectInsets` method instead.

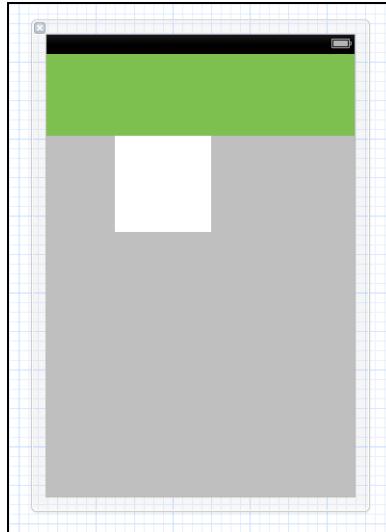
To see how this works, you're going to add a 10-point border around `MyView`'s content area. Add this method to `MyView.m`:

```
- (UIEdgeInsets)alignmentRectInsets
{
    return UIEdgeInsetsMake(10, 10, 10, 10);
}
```

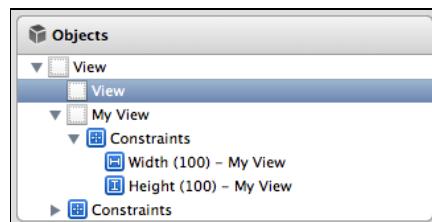
Make sure both Width and Height constraints in the nib have priority 200 again, and that `intrinsicContentSize` simply returns `mySize`, so that the intrinsic content size is used for both dimensions.

Run the app. Notice that the control is now larger, 50 by 50 points instead of 30 by 30. The extra padding from `alignmentRectInsets` is added around the content area to make the frame bigger.

To make this more visible, drag a new View object onto the main view. Place it directly above `MyView` and give it a different background color:



Make sure this view sits behind `MyView` in the hierarchy. You do this by dragging it above "My View" in the Document Outline:



Run the app again. You can see that `MyView` now overlaps the green view by 10 points. It didn't do that in the nib. That is the extra space around the alignment rect.



To complete this demonstration, override `drawRect:` in **MyView.m**:

```
- (void)drawRect:(CGRect)rect
{
```

```
[[UIColor redColor] setFill];

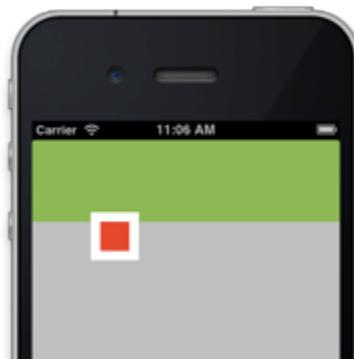
CGRect alignmentRect = [self
    alignmentRectForFrame:self.bounds];
UIRectFill(alignmentRect);

NSLog(@"Alignment rect: %@",  

    NSStringFromCGRect(alignmentRect));
}
```

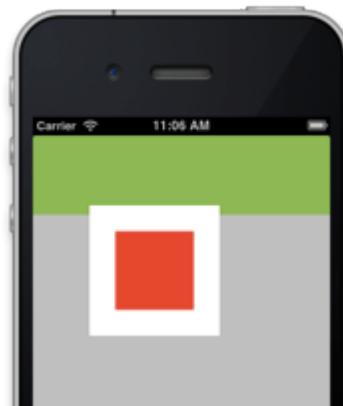
This will fill the actual content area with red so that you can clearly see which part of the frame is content and which is padding.

Run the app:



It's obvious now that the top of the red rectangle is what Auto Layout uses for layout, because that is still aligned with the green bar.

One problem: when you tap the view the whole thing resizes. However, there is no new `NSLog()` output, which means that `drawRect:` doesn't get called. Instead, the content of the view is scaled:



That just doesn't look right. To fix this, you need to add a new line to `viewTapped:`

```
- (void)viewTapped:(UITapGestureRecognizer *)recognizer
```

```
{  
    mySize.width += 30.0f;  
    mySize.height += 30.0f;  
  
    [self invalidateIntrinsicContentSize];  
    [self setNeedsDisplay]; // add this  
}
```

The call to `setNeedsDisplay` lets Auto Layout know that you also want to redraw this view as part of the layout update phase. That looks a lot better indeed:



In the Xcode output pane you can see that the alignment rectangle grows by 30 points in each direction with every tap.

A few notes on advanced layout

If your custom view contains subviews of its own, then you can also use Auto Layout to set the positions and sizes of these subviews. Set up your constraints in `initWithFrame:` or `initWithCoder:` and override the `+requiresConstraintBasedLayout` method to return `YES`. Piece of cake.

This works great for simple constraints that never have to change, but when you're designing a layout that needs to be more flexible, there are two other methods at your disposal: `updateConstraints` and `layoutSubviews`.

You have seen `updateViewConstraints` in `UIViewController`. You implemented this method to install the constraints on the buttons in the "dynamic layout" example. `UIView` has a similar method, named `updateConstraints`.

When you tell Auto Layout that the constraints for a view need to be refreshed, by calling `setNeedsUpdateConstraints` or `updateConstraintsIfNeeded`, it first calls `updateConstraints` on that view, followed by `updateViewConstraints` on the corresponding view controller.

So if you're looking for a good place to centralize the management of your constraints, then `updateConstraints` is it.

If your custom view does sophisticated layout, you can also override `layoutSubviews`. The default implementation of this method just sets the frames for the subviews based on Auto Layout's calculations. Sometimes you may want to change the constraints for those subviews based on the frames that Auto Layout has calculated, for example to dynamically remove subviews that no longer fit in the frame for the custom view; `layoutSubviews` is the place to do that.

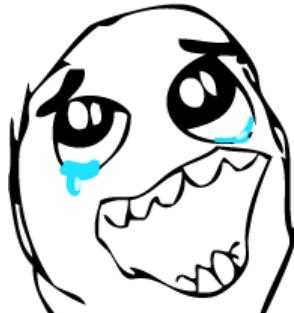
You can make this method do anything you want, as long as you make sure that the constraints and the frames are still in agreement after you're done. The video from WWDC 2012 session 228 has an interesting demo that shows how to do this.

Ladies and gentlemen, as your tour guide on this trip through the mysteries of Auto Layout, I am obliged to point out that it's still twenty or so pages until the end. If your fridge hasn't run out of refreshments because you were too busy reading to do any grocery shopping, then now is the time to fill your glass and have a snack.

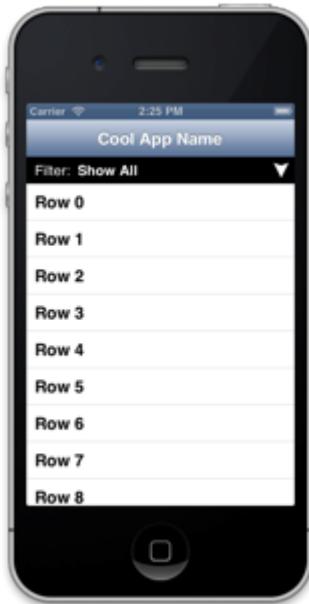
The end is in sight, but we're not there quite yet!

So how do you do this in a real app?

I'm sure you're fed up with examples that just have `UIButtons` and `UIViews`, so in this section you'll see everything that you've learned so far put into practice in an app that looks a bit more like a real app that you might find on the App Store.



Because this is only an example, the app has just one screen with a navigation bar and a table view. Between these two sits a bar that lets the user select which sorts of things the table view will display:

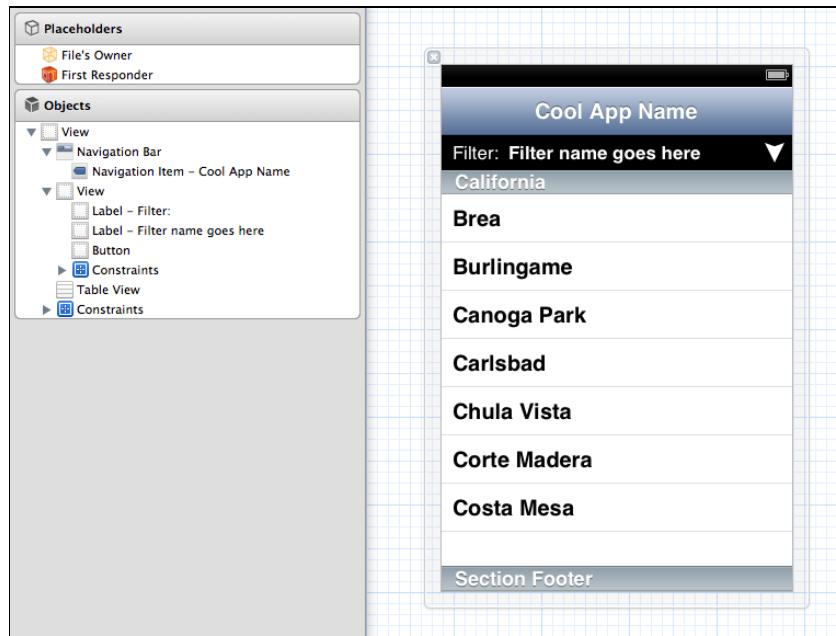


Tap on the button to show the filtering options; this is another table view:



You can find the starter source code for this app, "Filter," among the resources for this chapter. Run the app to see what it does (not much!) and take a quick look at the source code to see how it works.

The design of the app's main screen is done in Interface Builder:



Click around a bit to see the constraints that have been set. The navigation bar has a fixed size, because they always do. The filter bar has a fixed height constraint, and the table view has a flexible height. These three views are glued together vertically using Vertical Space constraints with 0 distance so there are no gaps between them.

Here's what you're going to implement: tapping on the arrow button should display a new table view that lists the filtering options. Tapping the button a second time should hide that table again. This is an example of dynamic layout, because this new table view will be inserted into the view hierarchy and constraints will need to change.

First, let's handle the button press. Add a new action method to **ViewController.m**:

```
- (IBAction)filterButtonPressed:(id)sender
{
    [self showFilterTable];
}
```

In the nib, connect this method to the button's "touch up inside" event. (Ctrl-drag from the button to File's Owner.)

What do you need to do inside the `showFilterTable` method? The view controller already has an instance variable for the `filterTableView`, but the table view object itself hasn't been created anywhere yet. So that would be the first order of business:

```
- (void)showFilterTable
```

```
{  
    filterTableView = [[UITableView alloc]  
                      initWithFrame:CGRectZero  
                      style:UITableViewStylePlain];  
  
    filterTableView.translatesAutoresizingMaskIntoConstraints  
        = NO;  
    filterTableView.dataSource = self;  
    filterTableView.delegate = self;  
  
    [self.view addSubview:filterTableView];  
}
```

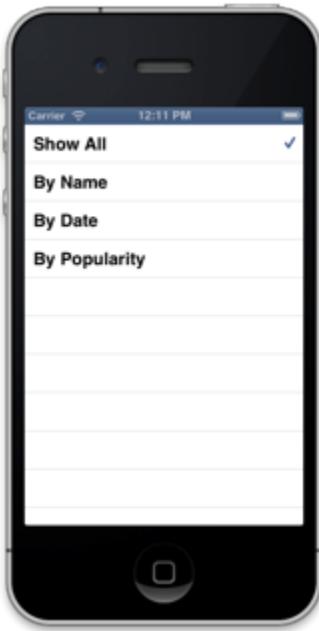
Notice that you can pass `CGRectZero` for the frame because Auto Layout will calculate the frame based on the constraints.

Speaking of constraints, what should those be? I prefer to build up apps step-by-step and right now I'm only interested in knowing that we can get the table view up on the screen somewhere, but I don't really care where that is. So let's add some basic constraints and fine-tune them later.

Add these lines to the end of `showFilterTable`:

```
NSDictionary *viewsDictionary =  
    NSDictionaryOfVariableBindings(filterTableView);  
  
NSArray *constraints = [NSLayoutConstraint  
    constraintsWithVisualFormat:@"H:[filterTableView]"  
    options:0  
    metrics:nil  
    views:viewsDictionary];  
  
[self.view addConstraints:constraints];  
  
constraints = [NSLayoutConstraint  
    constraintsWithVisualFormat:@"V:[filterTableView]"  
    options:0  
    metrics:nil  
    views:viewsDictionary];  
  
[self.view addConstraints:constraints];
```

Run the app and tap the button. The new table view should appear and it takes over the whole screen. The starter code already implements the data source for this table view, so you don't have to worry about filling it with content.



I want this table view to be smaller and white-on-black. Add these lines to `showFilterTable` to reduce the size of the font for the labels and the row height, and change the colors:

```
filterTableView.rowHeight = 24.0f;
filterTableView.backgroundColor = [UIColor blackColor];
filterTableView.separatorColor = [UIColor darkGrayColor];
```

And in `tableView:cellForRowAtIndexPath:` add the following within the `else` part of the second `if` condition:

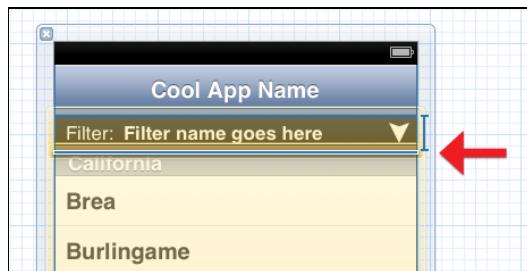
```
cell.textLabel.font = [UIFont systemFontOfSize:14.0f];
cell.textLabel.textColor = [UIColor whiteColor];
```

Run the app, and it should look like this after you tap the arrow button:



Cool, so the filter table view works. Now let's place it in between the filter bar and the main table view. How should you do that? By replacing some of the constraints, of course!

Currently the bottom of the filter bar is connected to the top of the main table view using a Vertical Space with size 0. You can see this in the nib:



That constraint has to go away. Instead, you will connect the bottom of the filter bar with the top of the filter table using a new constraint. You will also put a new constraint between the bottom of the filter table and the top of the main table view.

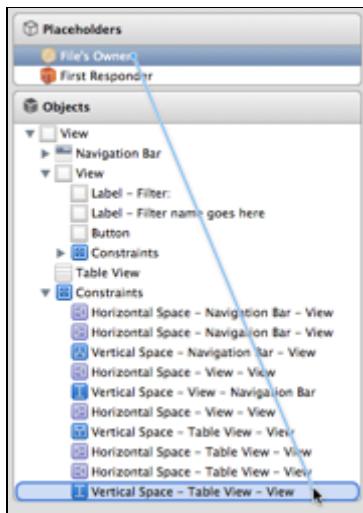
To remove a constraint that was created using Interface Builder, you first have to make an outlet for it. You're also adding an outlet for the filter bar because you need to refer to it later.

Add these properties to the class extension at the top of **ViewController.m**:

```
@property (nonatomic, weak) IBOutlet UIView *filterBar;  
@property (nonatomic, strong) IBOutlet NSLayoutConstraint
```

```
*spaceBetweenFilterBarAndMainTable;
```

In Interface Builder, Ctrl-drag from File's Owner to the Vertical Space constraint and connect it to the `spaceBetweenFilterBarAndMainTable` outlet. Also connect the view for the filter bar with the `filterBar` outlet.



In `showFilterTable`, remove the constraint for the vertical dimension (the one with the `@"V:|[filterTableView]|"` and the following line adding the constraint) and add the following code:

```
[self.view removeConstraint:  
    self.spaceBetweenFilterBarAndMainTable];  
  
NSLayoutConstraint *constraint = [NSLayoutConstraint  
    constraintWithItem:self.filterBar  
    attribute:NSLayoutAttributeBottom  
    relatedBy:NSLayoutRelationEqual  
    toItem:filterTableView  
    attribute:NSLayoutAttributeTop  
    multiplier:1.0f  
    constant:0.0f];  
  
[self.view addConstraint:constraint];  
  
constraint = [NSLayoutConstraint  
    constraintWithItem:filterTableView  
    attribute:NSLayoutAttributeBottom  
    relatedBy:NSLayoutRelationEqual  
    toItem:self.tableView  
    attribute:NSLayoutAttributeTop  
    multiplier:1.0f];
```

```
constant:0.0f];

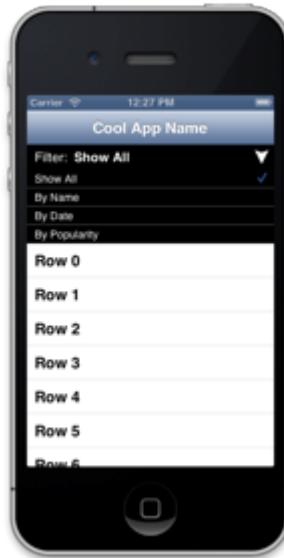
[self.view addConstraint:constraint];

constraint = [NSLayoutConstraint
    constraintWithItem:filterTableView
    attribute:NSLayoutAttributeHeight
    relatedBy:NSLayoutRelationEqual
    toItem:nil
    attribute:NSLayoutAttributeNotAnAttribute
    multiplier:1.0f
    constant:96.0f];

[self.view addConstraint:constraint];
```

This first removes the existing constraint and then adds the two new ones. It also puts a fixed height on the filter table; otherwise it would still extend all the way to the bottom of the screen.

Try it out! The filter table now pushes the main table down to make room:



Notice that you didn't need to call `setNeedsLayout` or anything – UIKit is smart enough to realize that the constraints have changed and that it therefore needs to recalculate the layout.

Could you also have done this using just the Visual Format Language? Absolutely!

Replace the above code with:

```
[self.view removeConstraint:
    self.spaceBetweenFilterBarAndMainTable];
```

```

viewsDictionary = @{
    @"filterTableView": filterTableView,
    @"filterBar": self.filterBar,
    @"mainTableView": self.tableView };

constraints = [NSLayoutConstraint
    constraintsWithVisualFormat:
        @"V:[filterBar][filterTableView(96)][mainTableView]"
    options:0
    metrics:nil
    views:viewsDictionary];

[self.view addConstraints:constraints];

```

That's a lot shorter. The visual format string is:

```
V:[filterBar][filterTableView(96)][mainTableView]
```

You need to read this vertically, so either tilt your head or pretend it is written like this:

```
[filterBar]
[filterTableView(96)]
[mainTableView]
```

This puts vertical space constraints of size 0 between these three views, and a fixed height of 96 points on filterTableView.

Note: In this app, the viewsDictionary isn't created with the `NSDictionaryOfVariableBindings()` macro, because two of the variables come from properties and you cannot write `[self.filterBar]` in the format string.

Hiding the filter table

Tapping the arrow button a second time should hide the filter options again. Implementing this is quite simple. When you remove a view from the screen, any constraints associated with it will be deleted automatically. So you just have to remove the `filterTableView` from the view hierarchy and add the `spaceBetweenFilterBarAndMainTable` constraint back in.

Change `filterButtonPressed:` to:

```

- (IBAction)filterButtonPressed:(id)sender
{
    if (filterTableView == nil)
        [self showFilterTable];

```

```
    else
        [self hideFilterTable];
}
```

And add the new `hideFilterTable` method:

```
- (void)hideFilterTable
{
    [filterTableView removeFromSuperview];
    filterTableView = nil;

    [self.view addConstraint:
        self.spaceBetweenFilterBarAndMainTable];
}
```

Because the `spaceBetweenFilterBarAndMainTable` property is strong, the object stayed alive even after it was removed from the view in `showFilterTable`, and you can simply add it back here. Because you held on to the existing constraint, there is no need to make a new one.

Run the app and try it out. You can now toggle the filter list on and off. Also give it a try in landscape.



It will be nice to make the table disappear after the user picks an item. That happens in the table view delegate method, so change it as follows:

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];

    if (tableView == filterTableView)
    {
        activeFilterIndex = indexPath.row;
        self.filterNameLabel.text =
    }
}
```

```
        filterNames[activeFilterIndex];  
  
        [self hideFilterTable];  
    }  
}
```

You can now select a row in the filter table:



Animation!

You know the drill by now: to make the magic of animation work with Auto Layout, you simply create an animation block and call `layoutIfNeeded`.

Add these lines to the bottom of `showFilterTable`:

```
[UIView animateWithDuration:0.3f animations:^  
{  
    [self.view layoutIfNeeded];  
}];
```

Run the app and tap the button. Hmm, that looks a bit weird. The main table view slides downwards all right, but the filter table seems to fly in from the top-left corner of the screen:



When you think about it, it makes sense. You initialized this table view with a frame of `CGRectZero`, so at the time the animation starts, the table view sits in the top-left corner and is very small. Core Animation will “tween” the view from that initial state to its final state.

But the result doesn’t look very good. Instead it is better to first set the table view’s initial position using constraints, force a layout update, and then set the position for the animation.

Add the new layout code in `showFilterTable`:

```
[self.view removeConstraint:  
    self.spaceBetweenFilterBarAndMainTable];  
  
viewsDictionary = @ {  
    @"filterTableView": filterTableView,  
    @"filterBar": self.filterBar,  
    @"mainTableView": self.tableView };  
  
// this is new  
constraints = [NSLayoutConstraint  
    constraintsWithVisualFormat:  
        @"V:[filterBar][filterTableView(0)][mainTableView]"  
    options:0  
    metrics:nil  
    views:viewsDictionary];  
  
[self.view addConstraints:constraints];  
  
[self.view layoutIfNeeded];  
  
[self.view removeConstraints:constraints];  
// until here  
  
constraints = [NSLayoutConstraint  
    constraintsWithVisualFormat:@"V:[filterBar][mainTableView(96)]"  
    options:0  
    metrics:nil  
    views:viewsDictionary];
```

The new bit creates constraints that place the filter table between the filter bar and main table view but set its height to 0 points, so it is effectively invisible but in the right place. Then it calls `layoutIfNeeded` to calculate the initial frame for the filter table. Immediately afterwards, it removes those constraints again and adds the ones you set before to make the `filterTableView` 96 points high.

Run the app. That looks a lot better!

Just remember that sometimes you need to use temporary constraints to move the new views into place before the animation.

Animating the filter options out of sight follows a similar procedure. Instead of removing the `filterTableView` from the screen outright, first remove the existing vertical constraints, and then restore the ones that wedge it between the filter bar and main table with height 0. You can create these constraints all over again, but it's just as easy to keep references to them.

Add two new instance variables:

```
@implementation ViewController
{
    . . .
    NSArray *verticalConstraintsBeforeAnimation;
    NSArray *verticalConstraintsAfterAnimation;
}
```

In `showFilterTable`, after you create the constraints with the formula,

```
v:[filterBar][filterTableView(0)][mainTableView]
```

do:

```
    verticalConstraintsBeforeAnimation = constraints;
```

And after creating the constraints for,

```
v:[filterBar][filterTableView(96)][mainTableView]
```

do:

```
    verticalConstraintsAfterAnimation = constraints;
```

This stores the constraints in the new instance variables, so you can refer to them later.

Finally, change `hideFilterTable` to:

```
- (void)hideFilterTable
{
    [self.view removeConstraints:
        verticalConstraintsAfterAnimation];
    [self.view addConstraints:
        verticalConstraintsBeforeAnimation];

    [UIView animateWithDuration:0.3f animations:^{
        {
            [self.view layoutIfNeeded];
        }
        completion:^(BOOL finished)
    }
}
```

```
[filterTableView removeFromSuperview];
filterTableView = nil;

[self.view addConstraint:
    self.spaceBetweenFilterBarAndMainTable];
};

}
```

This restores the constraints as they were before the “slide open” animation, so that the filter table appears to slide shut again. When the animation completes, the filter table view is removed completely and the original constraint is put back into place.

And that is how you make dynamic user interfaces by combining layouts made in Interface Builder with constraints made in code, topped off with a nice sauce of animation!

Internationalization and localization

Internationalization is the process of making your app support different languages, while localization is the act of providing the actual translations.

iPhones and iPads are used all over the world, and the fact is most people don’t speak English as their first language – or at all. Translating your apps is a pretty easy way to reach more customers, so there is no reason not to do it. And even if you’re not ready to have your apps translated right now, it’s smart to prepare for it.

There is more to internationalization than just translating text – you also need to deal with different ways to present dates and times, for example – but translating the text of your nibs and storyboards is a big part of it.

Before iOS 6, you had to make a copy of all your nibs and storyboards for each language you wished to support, and edit the text inside these files manually. That might not be too bad for version 1.0 of your app, but it quickly becomes a maintenance nightmare for all versions afterwards when you have to synchronize any changes that you make across all the different language versions. Decidedly not fun!



Fortunately, iOS 6 makes this a lot easier, and it's Auto Layout you have to thank for it. After all, labels and buttons now have an intrinsic content size, which means they automatically resize to fit whatever text you give them. Words in English are quite short compared to other languages, such as German, so your labels may need to be able to stretch a bit to fit the translated text. With Auto Layout that is no problem, and you can set up constraints that push other content out of the way, if necessary.

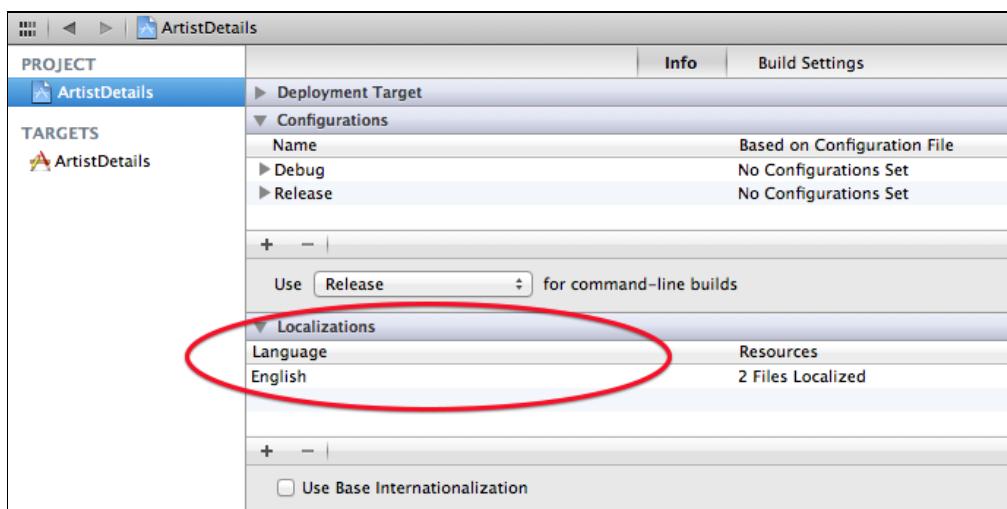
You have already seen this with the buttons in the "Artist Details" example from the previous chapter. If possible, the Delete, Cancel and Next buttons should have the same width, but the constraints still allow for these buttons to grow when another language requires it.

Let's take a look at that Artist Details example again and apply some real translations to it. If you didn't follow along with the last chapter or have already deleted the project, you can find the Artist Details source code in this chapter's resources.

In earlier versions of Xcode, you could go to the File inspector and tap the + button under the Localization header to add a new translation for the nib or storyboard file. This created a new "*language.Iproj*" folder with a copy of the resource.

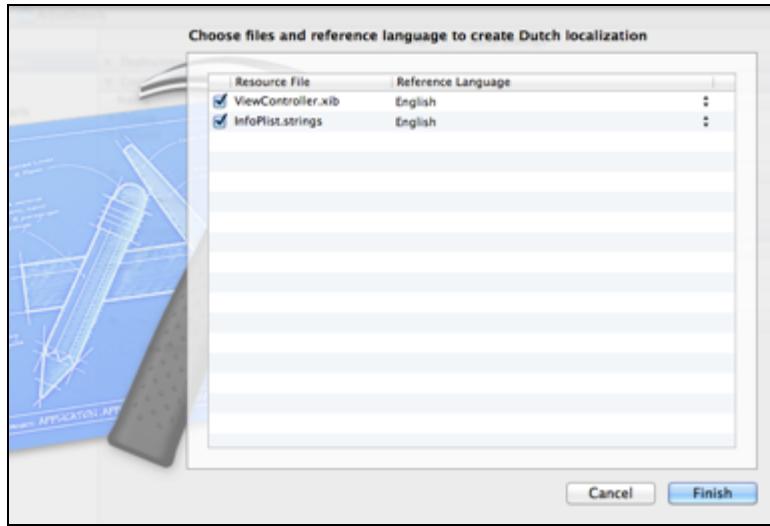
From Xcode 4.5 onwards, it no longer works that way. Instead, you create your localizations from the Project Settings screen.

Open the Artist Details project and go to the Project Settings screen. Under Localizations it currently lists only English, because that's how the Xcode template set up the project. Each supported language has its own **.Iproj** folder in the project that stores the translated files for that language. The **en.Iproj** folder for this app contains two localized files, InfoPlist.strings and ViewController.xib.



If you want to use the old system of localization, where you create copies of your nib or storyboard files and translate them individually, you can tap the + button to add a new localization. This works as before by creating a new **.Iproj** folder and

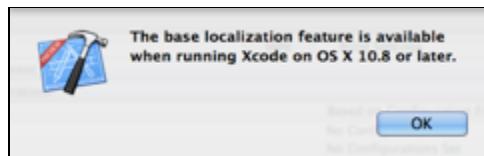
copying the resources into it. Xcode asks you which files you want to localize, and it copies them to that folder:



That's not what you want to do here, though. Instead, you want to use the new "Base Internationalization" feature. When enabled, this moves the nib from **en.lproj** into a special folder named **Base.lproj** and puts a strings file into **en.lproj** in its place. For each new translation, you don't get a copy of the nib, but just a file with all the text strings.

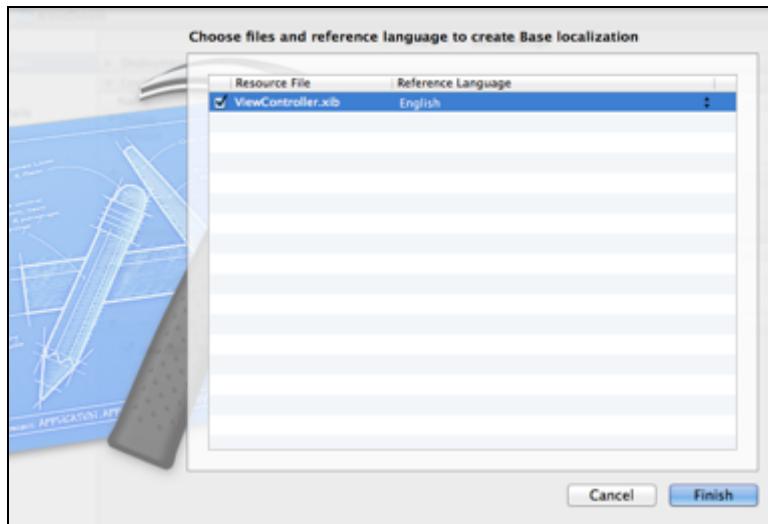
In the Project Settings screen, click "Use Base Internationalization."

Are you still running OS X Lion? Then you're out of luck, because you'll get a dialog like the following:



Unfortunately, the new Base Internationalization feature is only available when Xcode is running on OS X Mountain Lion. As a developer, it's always a good idea to upgrade your system to the latest version of OS X, so now is a good time to do that, if you haven't already. ☺

On Mountain Lion, Xcode asks which files you want to move to Base localization. This is the localization you will use for development and it doesn't necessarily have to be English.



Click Finish. Xcode now creates a new folder, **Base.Iproj**, and moves ViewController.xib out of en.Iproj and into this new Base.Iproj folder.

The Localizations section in the Project Settings screen now shows two localizations, Base and English:

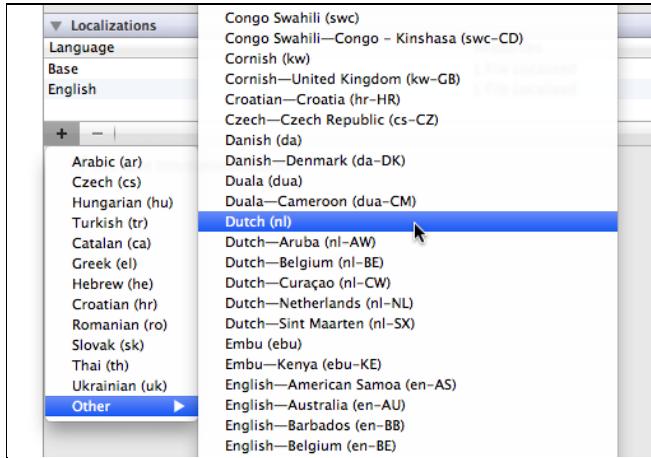
| Localizations | |
|---|------------------|
| Language | Resources |
| Base | 1 File Localized |
| English | 1 File Localized |
| + - | |
| <input checked="" type="checkbox"/> Use Base Internationalization | |

Both languages apparently have one file localized. For the Base localization that is ViewController.xib, and for English that is InfoPlist.strings, the file that contains translations for the text strings from the app's Info.plist file.

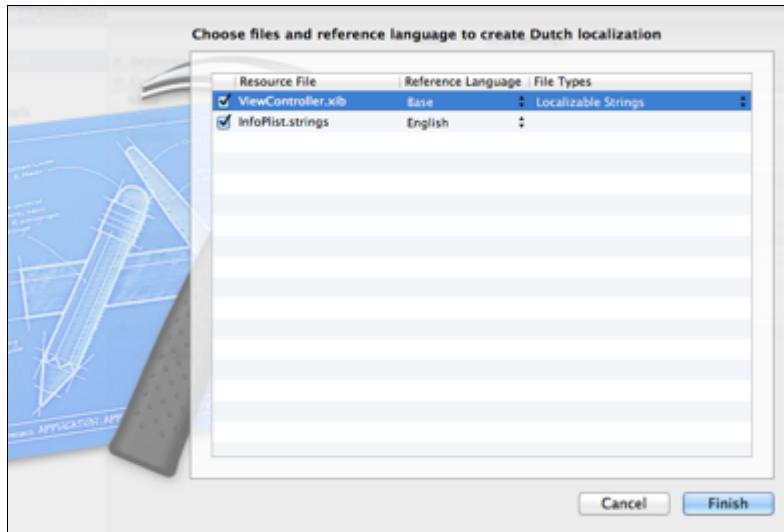
Note: There currently is no English translation for the nib file. That's not a problem because the text in the nib file for Base localization is already in English, but if your development language is something other than English, then you'd want to add an English translation as well.

Let's add a new localization for Dutch. Click the + button and from the big list, pick Dutch (nl).

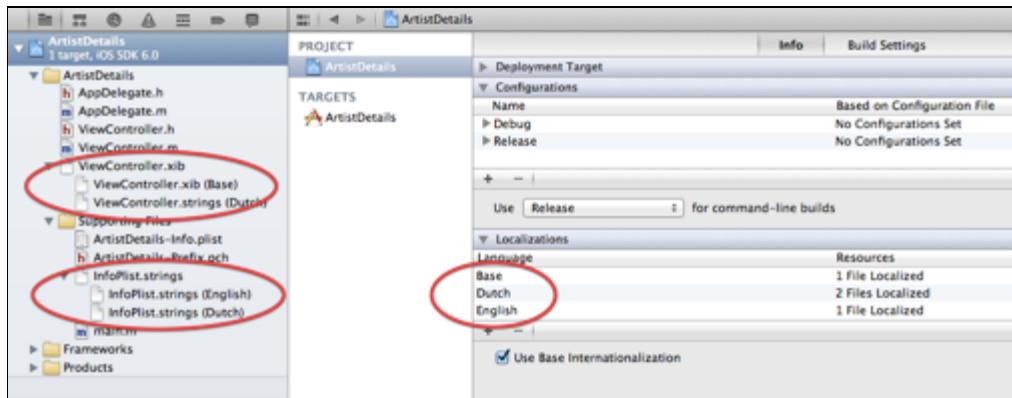
You might find Dutch in the main drop down, or you might have to select "Other" and go to the big list – it will depend on your system. Be sure to pick plain "Dutch (nl)", not one of the other options.



Xcode now asks which files you want to localize, and which existing translation should serve as the reference for the new one. Just stick with the defaults and click Finish.



Xcode creates a new folder, **nl.proj**, with InfoPlist.strings and ViewController.strings files. You can see in the Project Navigator which files are localized because they now have a little arrow in front of their names that can be expanded to show the available translations:



Click on **ViewController.strings (Dutch)** in the list. This opens a text file with content similar to the following (the objectIDs would be different):

```
/* Class = "IBUILabel"; text = "Artist Name"; ObjectID = "8"; */
"8.text" = "Artist Name";

/* Class = "IBUILabel"; text = "Notes:"; ObjectID = "13"; */
"13.text" = "Notes:";

/* Class = "IBUILabel"; text = "Rating:"; ObjectID = "16"; */
"16.text" = "Rating:";

/* Class = "IBUILabel"; text = "Album:"; ObjectID = "19"; */
"19.text" = "Album:";

/* Class = "IBUILabel"; text = "Release Year:"; ObjectID = "22";
*/
"22.text" = "Release Year:";

/* Class = "IBUILabel"; text = "Record Label:"; ObjectID = "26";
*/
"26.text" = "Record Label:";

...and so on...
```

These are the translations for the text from **ViewController.xib**. Interface Builder assigns its own IDs to the views in order to tell them apart, so that's why you see things like:

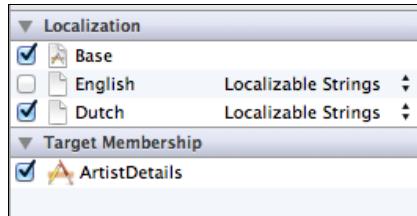
```
"8.text" = "Artist Name";
```

Here, 8 is the ID of the label at the top, and .text is obviously the label's text property. To translate the texts from the nib to Dutch, you simply change the string after the equals sign. For example, change this line:

```
"8.text" = "Artiest";
```

Click on **ViewController.xib** to open the nib in Interface Builder. It still appears in the Base localization because that is what you use during development.

Switch to the File inspector. In the Localization section you can see that it now lists Base and Dutch as available localizations, but English is unchecked:



To add an English location, simply check the corresponding box and Xcode will create a ViewController.strings file in the en.lproj folder. You might find this useful if your Base localization is in a language other than English.

Testing translations

So how do you test this? You need to run the app on the Simulator or your device, because there isn't a "preview in this localization" option in Interface Builder. First switch the Simulator or your device to the new language, and then run the app again.

You switch the language from the Settings app, under **General, International, Language**:

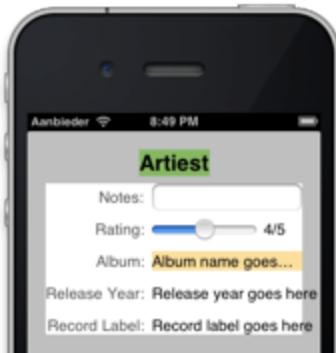


To switch to Dutch, choose "Nederlands" from the list of languages:



Tap the Done button. The Simulator or device will restart, which may take a few seconds. Once it comes back up, everything is now in Dutch. Welkom!

Run the app again and the artist label shows up in Dutch, too:



That's the only string you have translated so far, so everything else still uses the text from the Base localization.

Make the following changes to **ViewController.strings (Dutch)** (do not change the first part of the lines that says "<objected>.text", just the translated string):

```
"13.text" = "Opmerkingen:"; // Notes  
  
"16.text" = "Waardering:"; // Rating  
  
"19.text" = "Album:"; // Album  
  
"22.text" = "Jaar:"; // Release Year
```

```
"26.text" = "Maatschappij:"; // Record Label  
  
"40.normalTitle" = "Verwijder"; // Delete  
  
"46.normalTitle" = "Annuleer"; // Cancel  
  
"49.normalTitle" = "Volgende"; // Next
```

Save the file and run the app again. Now the app appears completely in Dutch:

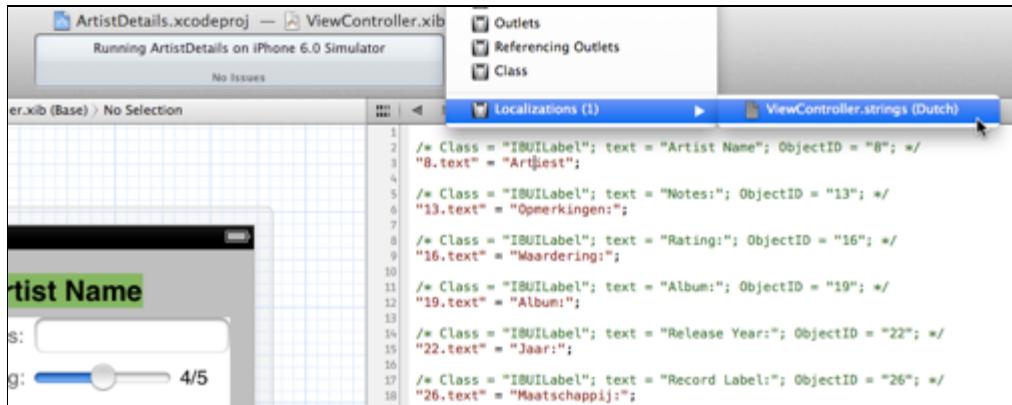


There is no reason to translate the strings “Album name goes here” and so on because these are only temporary placeholders.

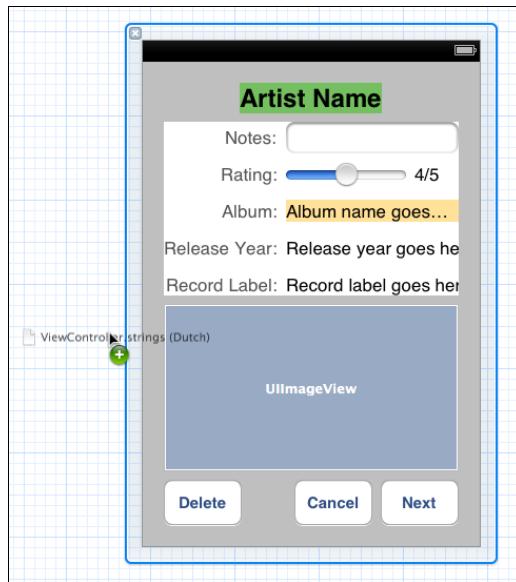
Notice that the labels all fit and that the buttons have properly resized. Because you were careful enough to set up constraints that made this possible when you designed the nib, localization is a breeze. All you need to do is hire a translator!



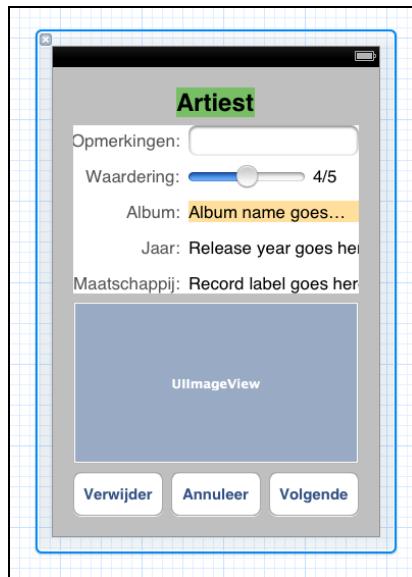
If you're a fan of Xcode's Assistant Editor, you'll be happy to know that it now has a Localizations category, so you can work with the nib or storyboard and the strings file side-by-side:



Earlier I mentioned that Interface Builder does not allow you to see a preview of your translations, but I'll let you in on a little trick you can use. Drag the strings file from the Project Navigator onto the view:



This directly applies the text from the translated strings to the labels and buttons in the nib:



However, you should know that this actually changes the text in the Base nib and you cannot undo it. If you also have an English translation, then you can drag that English strings file into the nib to restore the text. Another reason to be cautious with this trick is that it may cause Interface Builder to mess up your constraints.

Merging changes

If you make changes to the design of the nib – for example, you add a button – then how do you update your strings files? Interface Builder assigns an ID to each view and you need to know that ID to put it in the strings file, but where do you get it from?

There is currently no nice tool for this, and you will have to use the Terminal and FileMerge, or another “diff” program.

Drag a new button into the nib. It doesn’t really matter where you put it. Type **Cmd-S** to make sure the nib file is saved.

Open Terminal and go to the folder that contains the Artist Details source code. For example, if you put the `ArtistDetails` folder on the Desktop, you would do:

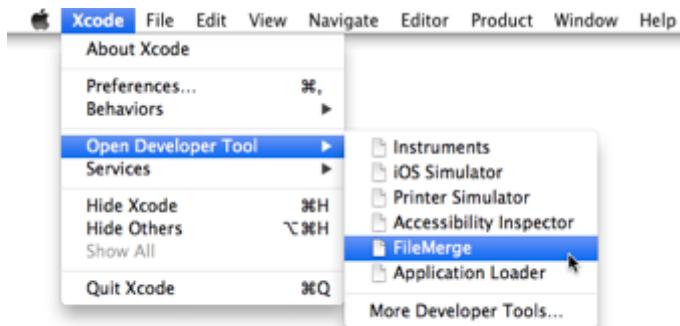
```
cd ~/Desktop/ArtistDetails/ArtistDetails
```

Now type:

```
ibtool --generate-strings-file new.strings Base.lproj/ViewController.xib
```

This creates a new text file, **new.strings**, with all the strings from the `ViewController.xib` file from the Base localization. You’re not going to add `new.strings` to the project – you’ll only use this file to find out what new strings were added. To do that, you need to compare the contents of `new.strings` with your existing translations using a tool such as FileMerge.

From Xcode's menu, choose **Open Developer Tool\FileMerge**:



Open the new.strings file and ViewController.strings in FileMerge, and it will show you where the new button was added and what its ID is:

```

/* Class = "IBUILabel"; text = "Release year goes here"; ObjectID = "321"; */
/* 321.text" = "Release year goes here";

/* Class = "IBUILabel"; text = "Record label goes here"; ObjectID = "324"; */
/* 324.text" = "Record label goes here";

/* Class = "IBUILabel"; text = "4/5"; ObjectID = "337"; */
/* 337.text" = "4/5";

```

→

```

/* Class = "IBUILabel"; text = "Release year goes here"; ObjectID = "321"; */
/* 321.text" = "Release year goes here";

/* Class = "IBUILabel"; text = "Record label goes here"; ObjectID = "324"; */
/* 324.text" = "Record label goes here;

/* Class = "IBUILabel"; text = "4/5"; ObjectID = "337"; */
/* 337.text" = "4/5";
/* Class = "IBUIButton"; normalTitle = "Button"; ObjectID = "400"; */
/* 400.normalTitle" = "Button";

```

This shows that a button with ID 400 was added since you originally created the strings files. You copy-paste these new lines into your localized ViewController.strings file and translate them.

This workflow can get on your nerves after a while, but fortunately there are several tools in the Mac App Store that can make merging translations a bit easier. Some examples are Linguan, Localization Helper, and nibTranslate. You can find others by searching the Mac App Store for “localization.”

Right-to-left languages

Remember leading and trailing? When you created Horizontal Space constraints in Interface Builder, or chose an attribute for your NSLayoutConstraints in code, you had the choice between leading and trailing, or left and right.

For languages such as English, these concepts are identical. Leading is left and trailing is right. But not all reading systems are the same – Hebrew and Arabic, for example, go from right-to-left. For such languages, the meaning of leading and trailing is reversed.

You can try this out for yourself by changing the language of your device to Hebrew or Arabic. Go to the Settings app, **International, Language** and choose עברית for Hebrew or عربية for Arabic.

Note: If your device is still set to Dutch, then the Settings app is named Instellingen. Go to **Algemeen, Internationaal, Taal**, and choose the language.

Choosing a right-to-left language doesn't automatically make your app switch right for left. You also need to add a Hebrew or Arabic localization. You do this from the Project Settings screen, just like you did before for Dutch. (Click the + button in the Localizations section and choose the new language from the popup menu.)

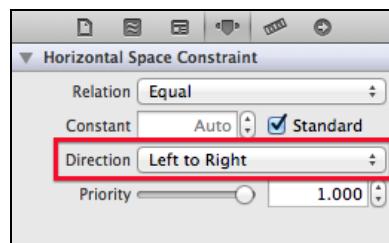
Run the app and now all the labels and buttons are swapped:



Of course the text is still in English, because it hasn't been translated yet.

You may not always want to use leading or trailing. Sometimes left or right is the proper choice, for example with image views. If an image view contains a photo, you probably don't want to swap that around.

You can set this up in the Attributes inspector for the constraint. Change the Direction field from "Leading to Trailing" into "Left to Right":

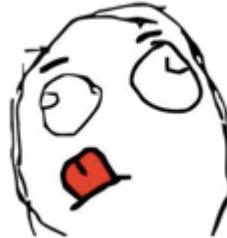


Do that for all the Horizontal Space constraints between the buttons at the bottom, and the app will show them as before when you had the phone in English:



As you can see, adding support for languages that go from right-to-left is pretty easy if you set up your constraints using trailing and leading.

!segaugnal tfel-ot-thgir ekil I



If you forgot how to switch the Simulator back to English, then simply choose **Reset Content and Settings** from its menu.

Localization tips

Here are some tips for making the most out of your localizations:

- Avoid using explicit width or height constraints on your buttons and labels and any other controls with text on them. Such constraints may cause the text to be truncated or clipped when you translate the app. Instead, make the user interface flexible so it can accommodate longer text.
- You're not required to use the Base Internationalization system. And if you do, it's not a requirement for all languages. You can still have specific nibs or storyboards

for one or more languages, just in case Auto Layout isn't flexible enough to automatically give you the layout you want.

- The Base Internationalization system also works if you don't use Auto Layout, but you won't reap all the benefits that Auto Layout has to offer, such as a UI that will automatically adapt to larger or smaller text strings.

Where to go from here?

Congratulations! If you were able to follow along with both chapters, you can now call yourself an Auto Layout master!

It can be tricky at first to get the hang of constraints and how to combine them into layouts that are not ambiguous or conflicting, but with some practice it all starts to make sense – and you had a lot of practice over the past few hundred pages! Go on, pat yourself on the back, you've earned it!



The big question is: should you switch to Auto Layout? For new apps that can afford to have iOS 6 as the minimum requirement, it is definitely worth doing, especially because it makes localization so much easier – and the more languages your app supports, the more money you will make!

If you have an existing app, you may decide to migrate it to Auto Layout, either all at once or nib-by-nib. Keep in mind that anything you can do with springs-and-struts and autosizing masks you can also do with Auto Layout, so it is backwards compatible in that sense. You can even combine autosizing masks with Auto Layout in the same view using the `translatesAutoresizingMaskIntoConstraints` property.

However, once you switch a view controller to Auto Layout, keep the following in mind:

- You should no longer set the frames of your views directly through the `frame`, `bounds` or `center` properties or their setter methods. Anywhere in your code that you change these properties you should now switch to using constraints instead.
- Keep an eye on places in the code where you're doing `addSubview` and `removeFromSuperview`, to make sure you don't need to add or remove any constraints there as well. Also look at `layoutSubviews` if you override that in your own views anywhere.

- Note that the frames of your views may not always be set by the time `viewDidLoad` is called, because Auto Layout does its calculations after that. So if you depend on those frames being known in `viewDidLoad`, you either need to move that logic into another method such as `viewDidAppear:` or `viewDidLayoutSubviews`, or you need to call `[self.view layoutIfNeeded]` inside `viewDidLoad` to force Auto Layout to do its calculations right then and there.
- If you get stuck and have no idea why your constraints are ambiguous or conflicting – or they just don't do what you want – you can inspect the constraints that have an effect on your view with `constraintsAffectingLayoutForAxis:`. This not only lists the view's own constraints, but also the other constraints that are pulling on it.
- And if all else fails, Instruments has a template for debugging Auto Layout. You can find it under **User\All\Cocoa Layout**. This template shows exactly what Auto Layout is doing while your app is running, and you can use the search field to find the history of what happened to a particular constraint.

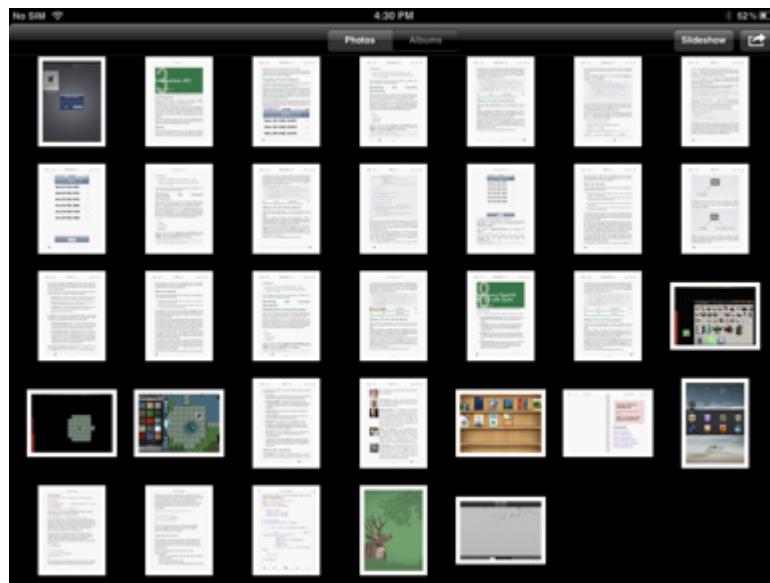
Don't forget to consult Apple's official documentation. The Cocoa Auto Layout Guide is quite comprehensive and it contains a full reference guide for the Visual Format Language. You can find it at the Developer Portal and in your Xcode documentation browser.

And that's it - I hope to see you create some cool and flexible layouts with Auto Layout in the future!

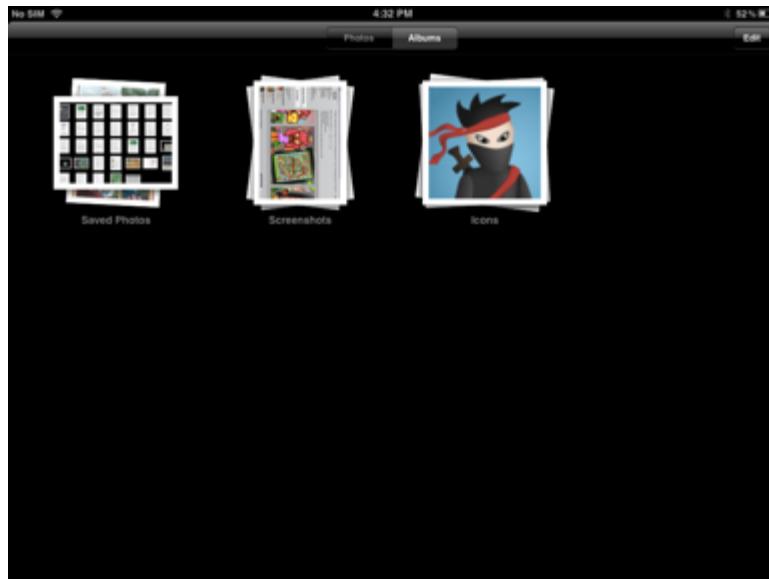
Chapter 5: Beginning UICollectionView

By Brandon Trebitowski

When Apple first launched the iPad in 2010, you might have been particularly impressed by the Photos app bundled with the device. It had a unique and stylish way of displaying photos via a multitude of layouts. You could view your photos in a nice grid view:



Or you could view your photo albums at the top level as stacks:



You could even transition between the two layouts with a cool pinch gesture. "Wow, I want that in my app!", you may have thought.

Well, implementing a grid view and other alternative layouts like this was possible to get working on your own, but also quite tricky! It required a lot of code and was difficult to get working exactly right. Couldn't there be an easier way?

Good news – now there is in iOS 6! Apple has introduced a new class called `UICollectionView` that makes adding your own custom layouts and layout transitions (like those in the Photos app) incredibly simple to build.

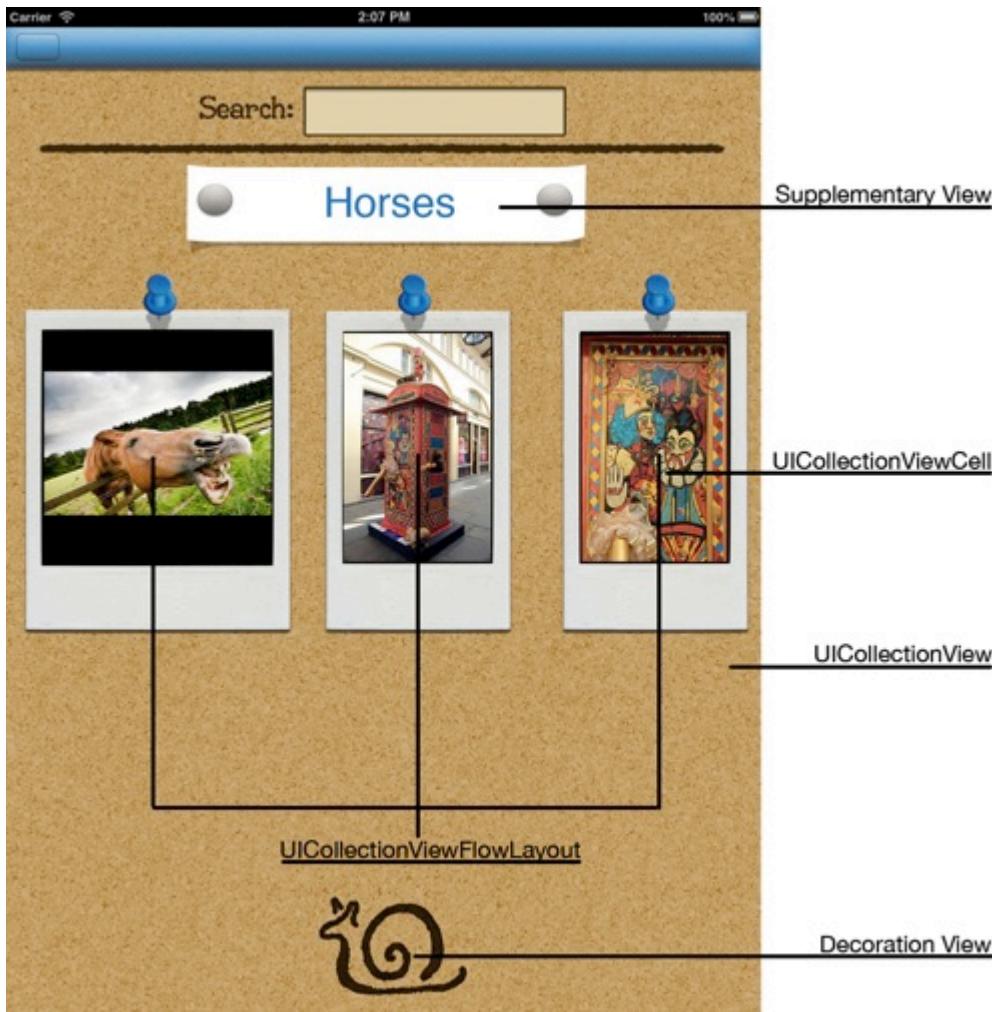
You're by no means limited to stacks and grids, because `UICollectionView` is extremely customizable. You can use it to make circle layouts, cover-flow style layouts, Pulse news style layouts – almost anything you can dream up!

The bottom line is you have an extremely powerful new way to present ordered data to users, and you should start learning about it! It's just as important (and helpful) of a class as `UITableView` is. The good news is if you're familiar with `UITableView`, you'll have no problem picking it up – using it is very similar to the table view data source and delegate pattern.

In this chapter, you'll get hands-on experience with `UICollectionView` by creating your own grid-based photo browsing app. By the time you are done this chapter, you will know the basics of using `UICollectionView` and will be ready to start using this amazing technology in your apps!

Anatomy of a UICollectionView Controller

Let's go right to an example of one of these babies in action. The `UICollectionViewController` family contains several key components, as you can see below:



Take a look at these components one-by-one:

1. `UICollectionView` – the main view in which the content is displayed, similar to a `UITableView`. Note that it doesn't necessarily have to take up the entire space inside the view controller – in the screenshot above, there's some space above the collection view where the user can search for a term.
2. `UICollectionViewCell` – similar to a `UITableViewCell` in `UITableView`. These cells make up the content of the view and are added as subviews to the `UICollectionView`. Cells can either be created programmatically, inside Interface Builder, or via a combination of the two methods.
3. **Supplementary Views** – if you have extra information you need to display that shouldn't be in the cells but still somewhere within the `UICollectionView`, you

should use supplementary views. These are commonly used for headers or footers of sections.

4. **Decoration View** – if you want to add some extra views to enhance the visual appearance of the `UICollectionView` (but don't really contain useful data), you should use decoration views. Background images or other visual embellishments are good examples of decoration views.

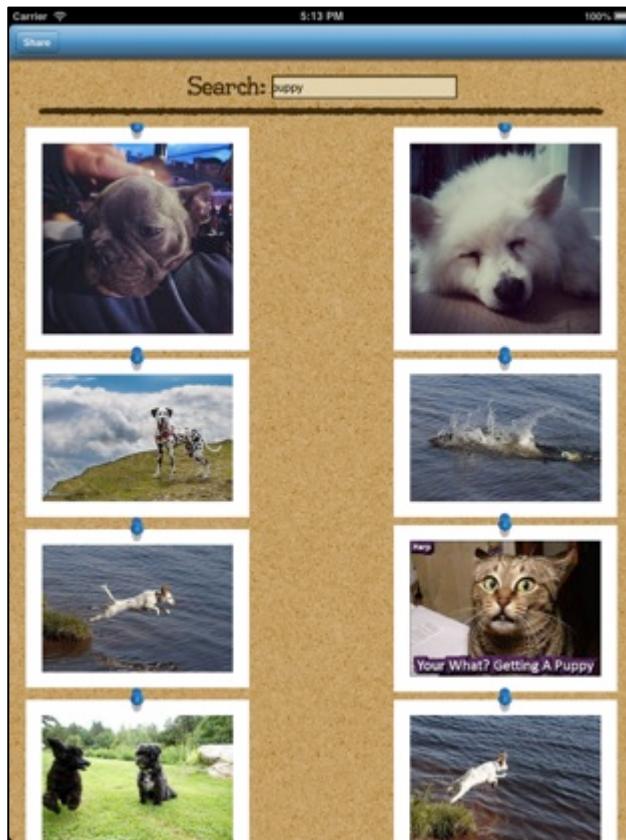
In addition to the above visual components, `UICollectionView` also has non-visual components that help with laying out content:

5. **UICollectionViewLayout** – `UICollectionView` does not know anything about how to set up cells on screen. Instead, its `UICollectionViewLayout` class handles this task. It uses a set of delegate methods to position every single cell in the `UICollectionView`. Layouts can be swapped out during runtime and the `UICollectionView` can even automatically animate switching from one layout to another!
6. **UICollectionViewFlowLayout** – You can subclass `UICollectionViewLayout` to create your own custom layouts (as you'll learn about in the next chapter), but Apple has graciously provided developers with a basic "flow-based" layout called `UICollectionViewFlowLayout`. It lays elements out one after another based on their size, quite like a grid view. You can use this layout class out of the box, or subclass it to get some interesting behavior and visual effects.

You will learn more about these elements in-depth throughout this chapter and the next. But for now, it's time for you to get your hands into the mix with a project!

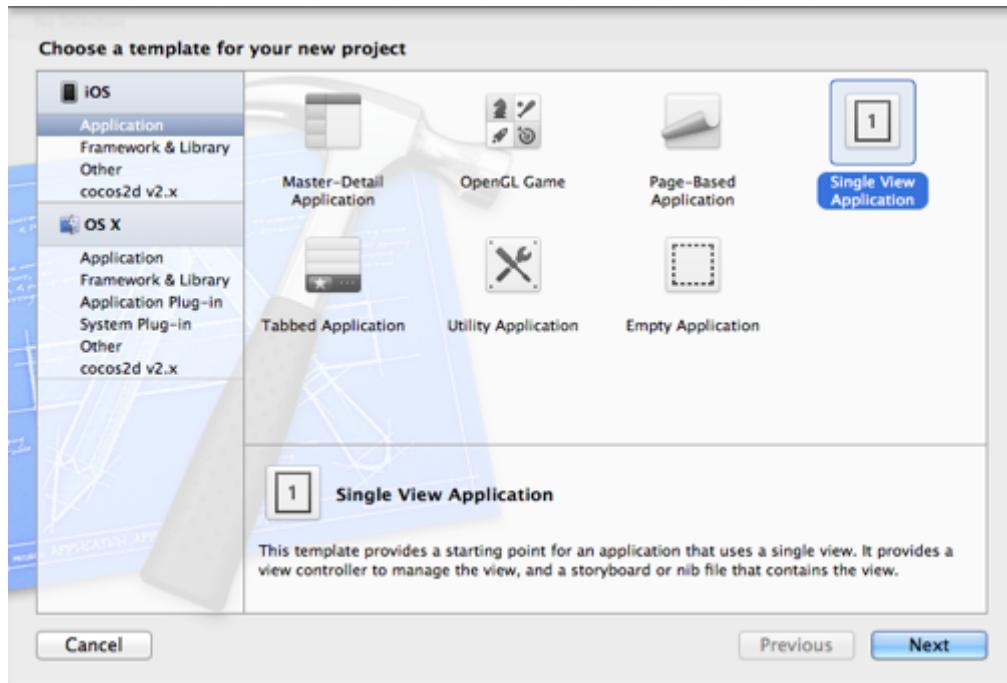
Introducing FlickrSearch

In the rest of this chapter, you are going to create a cool photo browsing app called FlickrSearch. It will allow you to search for a term on the popular photo sharing site Flickr, and it will download and display any matching photos on a beautiful corkboard-themed grid view:



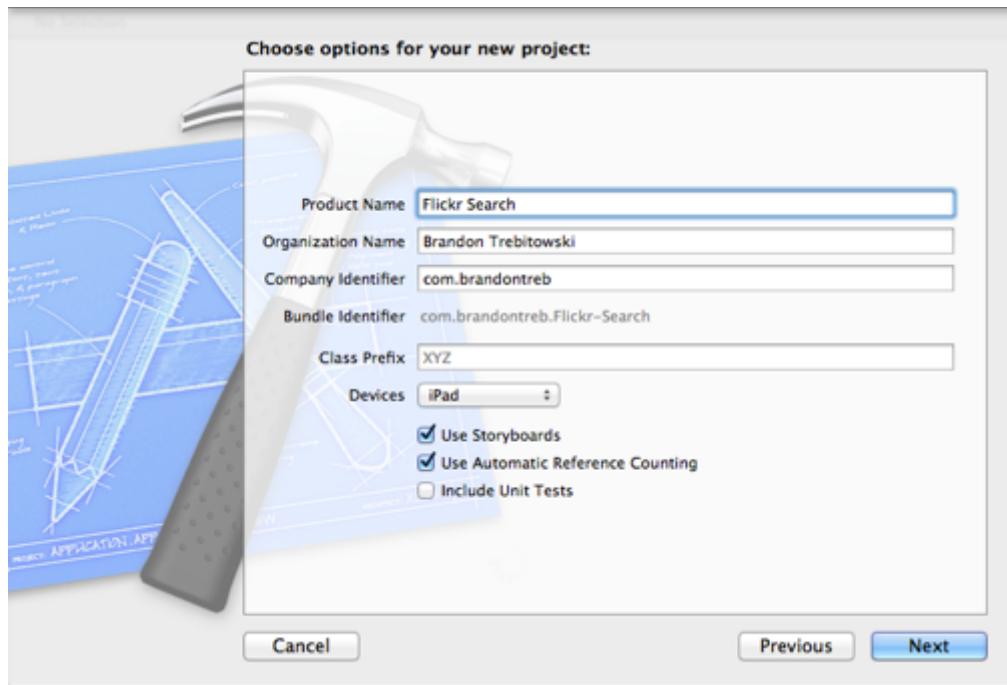
Before you begin, make sure you have the **Assets** folder that comes with the resources for this chapter handy. You won't get very far without them!

Ready to get started? Fire up Xcode and go to **File\New\Project...** and select the **iOS\Application\Single View Application** template.



This template will provide you with a simple `UIViewController` and storyboard to start out with, and nothing more. It's a good "almost from scratch" point to start from.

Click **Next** to fill out the information about the application. Set the Product Name to **FlickrSearch**, the device type to **iPad**, and make sure that the **Use Storyboards** and **Use Automatic Reference Counting** boxes are checked. Click **Next** to select the project location, and then click **Create**.



Compile and run, and you'll see that it's just a vanilla application with a single blank view.

Next you should import the assets - drag the images from the **Assets** folder mentioned earlier into your project in Xcode, making sure that the box **Copy items into destination group's folder (if needed)** is checked. Click **Finish**.

Pinning up the corkboard

You'll begin by creating a basic but stylish design to get your application looking smart. Once the initial design is in place, you'll add your UICollectionView.

Before you crack open the storyboard, declare some `IBOutlets` and `IBActions` so your class can interact with the interface items. Open `ViewController.m` and update the `@interface` declaration at the top to look like the code below:

```
@interface ViewController () <UITextFieldDelegate>

@property(nonatomic, weak) IBOutlet UIToolbar *toolbar;
@property(nonatomic, weak)
    IBOutlet UIBarButtonItem *shareButton;
@property(nonatomic, weak) IBOutlet UITextField *textField;

- (IBAction)shareButtonTapped:(id)sender;

@end
```

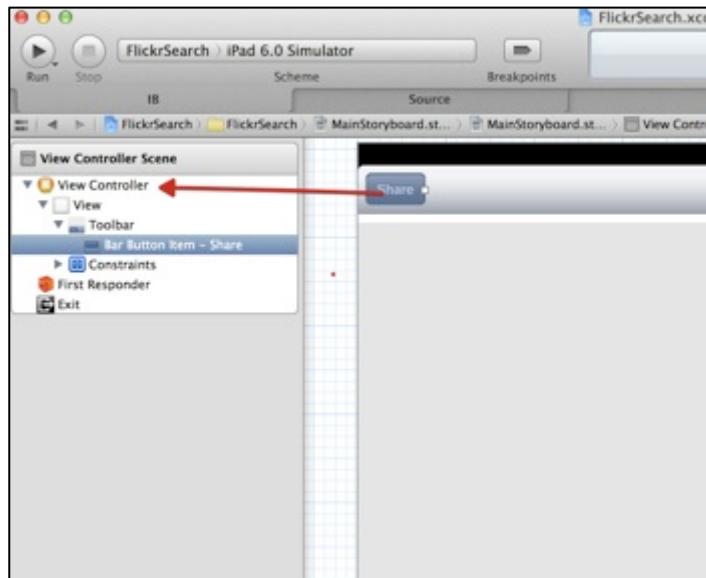
Also add an empty placeholder for `shareButtonTapped:` to the end of the file (you'll fill this in later on):

```
- (IBAction)shareButtonTapped:(id)sender {
    // TODO
}
```

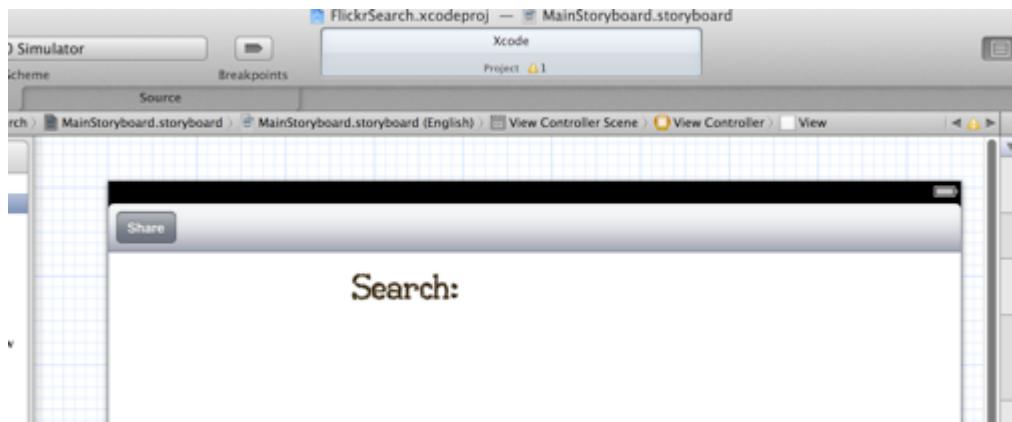
You are declaring these outlets and actions in the implementation file because they only need to be visible to the `viewController` class (as you learned in Chapter 2, "Programming in Modern Objective-C"). Now it's time to hook them up.

Open **MainStoryboard.storyboard**. Drag a Toolbar object from the Object Library (third tab on the lower half of the right sidebar) onto the main view and change the text of the button in the upper left to **Share** by double clicking it (or by changing the title property in the Attributes Inspector).

Next control-drag from the Share button (either in the interface view or the left sidebar) to the view controller object in the left sidebar. Select the `shareButtonTapped:` method from the list that appears to connect the button to that method.



Next, add a search label and search box. Drag an image view object onto your main view and set its image property to **search_text.png**. Currently the image looks terrible, but you can fix it. Set the mode property to center and position the search just under the toolbar. Alternatively, you can also use the Editor\Size to Fit Content menu option to resize the image view to fit its contents exactly.

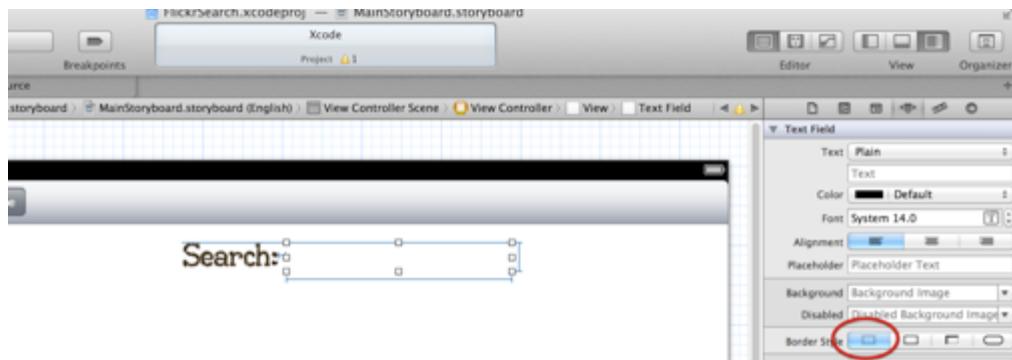


Note: If you're wondering why you used an image view instead of a label, that's a good question. ☺ In this particular instance, you want to have a specific look for your text. So you use an image that has the look you want for the app. Note this method has a major disadvantage in that it makes localization more difficult, but is the easiest way for a certain look if you're sure you just want one language.

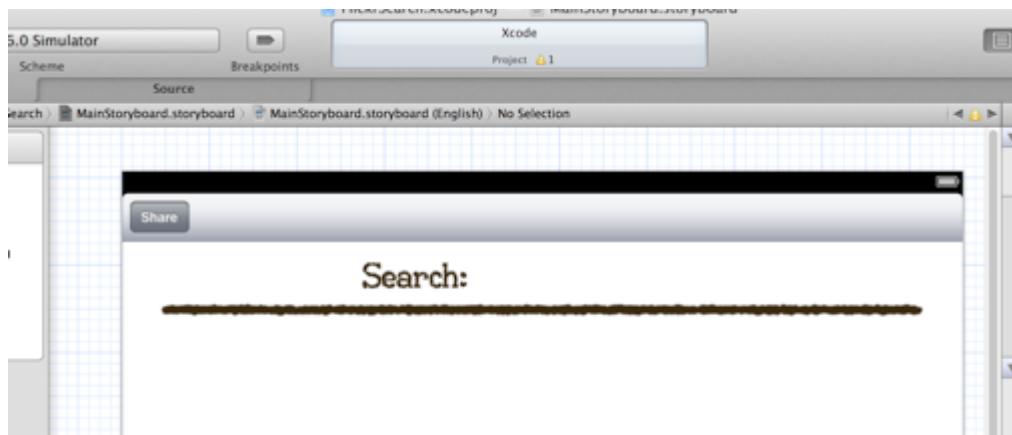
For the search box, drag a text field object onto your view and align it to the right of the search label. Make sure you set its border style to none (the icon in the

inspector with the dotted lines around it), as you will later give it custom styling in the code.

After you've added the text field, control-drag from the text field to the view controller object in the left sidebar and select "delegate" from the popup menu. This way, the `ViewController` class will be set as the delegate for this text field, which you will need so you can implement code to dismiss the keyboard upon return.



Finally, add a line under the search area to separate the search area from the results area. To do this, drag another image view object onto your view directly under the search box and label. Set the image property to `divider_bar.png`, size the image to fit, and adjust its position so that it's centered (or size it to fit the content). Your interface should now look something like this:



The last step is to hook up the `IBOutlets`. Click on View Controller in the left sidebar and then select the Connections Inspector (last tab on the upper half of the right sidebar). Drag from each of the `IBOutlets` you created (`shareButton`, `textField`, and `toolbar`) to their respective interface elements.

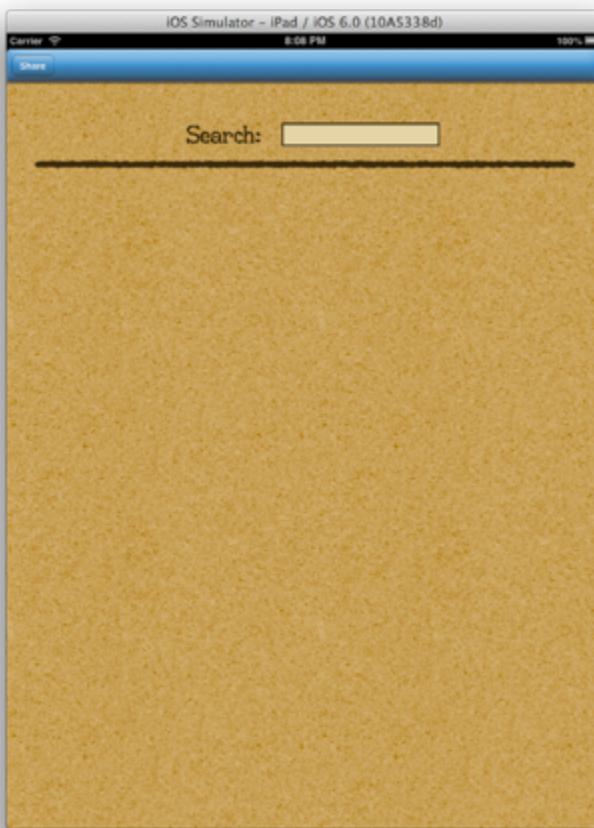


Now comes the fun part. You are going to stylize the view to make it look less bland in the storyboard. Open up **ViewController.m** and add the following to the end of **viewDidLoad**:

```
self.view.backgroundColor = [UIColor  
colorWithPatternImage:[UIImage imageNamed:@"bg_cork.png"]];  
  
UIImage *navBarImage = [[UIImage imageNamed:@"navbar.png"]  
resizableImageWithCapInsets:UIEdgeInsetsMake(27, 27, 27, 27)];  
[self.toolbar setBackgroundImage:navBarImage  
forToolbarPosition:UIToolbarPositionAny  
barMetrics: UIBarMetricsDefault];  
  
UIImage *shareButtonImage = [[UIImage imageNamed:@"button.png"]  
resizableImageWithCapInsets:UIEdgeInsetsMake(8, 8, 8, 8)];  
[self.shareButton setBackgroundImage:shareButtonImage  
forState:UIControlStateNormal barMetrics: UIBarMetricsDefault];  
  
UIImage *textFieldImage = [[UIImage  
imageNamed:@"search_field.png"]  
resizableImageWithCapInsets:UIEdgeInsetsMake(10, 10, 10, 10)];  
[self.textField setBackground:textFieldImage];
```

This sets the background of the entire view to a repeatable corkboard image (using the handy `UIColor:colorWithPatternImage` method), and sets the background image of the toolbar, share button, and text field to an image.

Build and run your app to see what the initial user interface will look like. You should see something like this:



Not bad - this is a great place to start! It looks like a bulletin board where you might want to tack all sorts of cool images. In the rest of this chapter, you will use UICollectionView to bring this design to life!

Fetching Flickr photos

Your first task for this section is to say the section title ten times fast. OK, just kidding. ☺

Flickr is a wonderful image sharing service that has a publicly accessible and dead-simple API for developers to use. With the API you can search for photos, add photos, comment on photos, and much more.

To use the Flickr API, you need an API key. If you are doing a *real* project, I recommend you sign up for one here:

<http://www.flickr.com/services/api/keys/apply/>.

However, for test projects like this, Flickr has a sample key they rotate out every so often that you can use without having to sign up. Simply perform any search at:

<http://www.flickr.com/services/api/explore/?method=flickr.photos.search> and copy

the API key out of the URL at the bottom – it follows the “&api_key=” all the way to the next “&”. Paste it somewhere in a text editor for later use.

For example, if the URL is:

http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key=6593783efea8e7f6dfc6b70bc03d2afb&format=rest&api_sig=f24f4e98063a9b8ecc8b522b238d5e2f

Then the API key is: 6593783efea8e7f6dfc6b70bc03d2afb

Note: If you use the sample API key, note that it is changed periodically. So if you’re doing this chapter over the course of several days, you might find that you have to get a new API key every so often. For this reason it might be easier to get an API key of your own from Flickr if you think you’re going to spend several days on this project. ☺

Since this chapter is about UICollectionView and not the Flickr API, I have created a set of classes for you that abstracts the Flickr search code. You can find them in the **Flickr Interface** folder in this chapter’s resources. Drag the four files into your project, making sure that the box **Copy items into destination group's folder (if needed)** is checked, and click Finish.

The two classes you imported are:

- **FlickrPhoto:** Provides a simple block-based API to perform a search and return an array of FlickrPhotos.
- **FlickrPhoto:** Data about a photo retrieved from Flickr – its thumbnail, image, and metadata information such as its ID.

Feel free to take a look at the code – it’s pretty simple and might inspire you to make use of Flickr in your own projects!

When you’re ready to go, move on to the next section – it’s time to do a little prep work before hooking into Flickr.

Preparing the data structures

You’re going to design this project so that after each time you perform a search, it displays a new “section” in the collection view with the results (rather than simply replacing the previous section). In other words, if you search for “ninjas” and then “pirates”, there will be a section of ninjas and a section of pirates in the table view. Talk about a recipe for disaster! ☺

To accomplish this, you’re going to need to create a data structure so you can keep the data for each section separate. If you’re thinking an `NSMutableDictionary` would be a good fit for this, you are correct. The keys of the dictionary will be the search

terms, and the values will be arrays of `FlickrPhoto` objects representing images that match the given search term.

Begin by building the array and dictionary to hold the search terms and results, and by creating the `Flickr` object that will do the searching. Open up **ViewController.m** and import the following classes:

```
#import "Flickr.h"
#import "FlickrPhoto.h"
```

Next, add a few properties to the `@interface` declaration:

```
@property(nonatomic, strong) NSMutableDictionary *searchResults;
@property(nonatomic, strong) NSMutableArray *searches;
@property(nonatomic, strong) Flickr *flickr;
```

Then initialize these properties by adding the following to the end of `viewDidLoad`:

```
self.searches = [@[] mutableCopy];
self.searchResults = [@{} mutableCopy];
self.flickr = [[Flickr alloc] init];
```

`searches` is an array that will keep track of all the searches made in the app, and `searchResults` will associate each search term to a set of results.

Next up, you'll learn how to populate these properties based on the user's input.

Getting good results

Before you can search Flickr, you need to enter an API key. Open up **Flickr.m** and replace the value of `kFlickrAPIKey` with the API key you obtained earlier. It should look something like this:

```
#define kFlickrAPIKey @"ca67930cac5beb26a884237fd9772402"
```

You are now ready to get your Flickr search on! Switch to **ViewController.m** and add the following code to the end of the file (but above `@end`):

```
#pragma mark - UITextFieldDelegate methods

- (BOOL) textFieldShouldReturn:(UITextField *)textField
{
    // 1
    [self.flickr searchFlickrForTerm:textField.text
        completionBlock:^(NSString *searchTerm, NSArray *results,
        NSError *error) {
            if(results && [results count] > 0)
            {
```

```
// 2
if (![self.searches containsObject:searchTerm])
{
    NSLog(@"Found %d photos matching %@", [results count], searchTerm);
    [self.searches insertObject:searchTerm atIndex:0];
    self.searchResults[searchTerm] = results;
}
// 3
dispatch_async(dispatch_get_main_queue(), ^{
    // Placeholder: reload collectionview data
});
} else {
    // 1
    NSLog(@"Error searching Flickr: %@", error.localizedDescription);
}
};

[textField resignFirstResponder];
return YES;
}
```

When the user hits the enter key on the keyboard, this method will be called (because earlier you set the view controller up as the delegate of the text field). Here is an explanation of the code:

1. Uses the handy `Flickr` wrapper class I provided to search Flickr for photos that match the given search term asynchronously. When the search completes, the completion block will be called with a reference to the searched term, the result set of `FlickrPhoto` objects, and an error (if there was one).
2. Checks to see if you have searched for this term before. If not, the term gets added to the front of the `searches` array and the results get stashed in the `searchResults` dictionary, with the key being the search term.
3. At this stage, you have new data and need to refresh the UI. Here the collection view needs to be reloaded to reflect the new data. However, you haven't yet implemented a collection view, so this is just a placeholder comment for now.
4. Finally, logs any errors to the console. Obviously, in a production application you would want to display these errors to the user.

Go ahead and run your app. Perform a search in the text box, and you should see a log message in the console indicating the number of search results, similar to this:

```
2012-07-10 21:44:16.505 Flickr Search[11950:14f07] Found 18 photos
matching 1337 h4x
```

```
2012-07-10 21:44:32.069 Flickr Search[11950:14f0b] Found 20 photos  
matching cat pix
```

Note that the results are limited to 20 by the `Flickr` class to keep load times down.

Now that you've got a list of photos to display, it's finally time to try `UICollectionView` and display them on the screen!

Preparing for the UICollectionView

As you probably already know, when you use a `UITableView` you have to set a data source and a delegate in order to provide the data to display and handle events (like row selection).

Similarly, when you use a `UICollectionView` you have to set a data source and a delegate as well. Their roles are the following:

- The data source (`UICollectionViewDataSource`) returns information about the number of items in the collection view and their views.
- The delegate (`UICollectionViewDelegate`) is notified when events happen such as cells being selected, highlighted, or removed.

And new to `UICollectionView`, you have a third protocol you must implement – a protocol specific to the layout manager you are using for the collection view. In this chapter you will be using the premade `UICollectionViewFlowLayout` layout manager, so you must implement the `UICollectionViewDelegateFlowLayout` protocol. It allows you to tweak the behaviour of the layout, configuring things like the cell spacing, scroll direction, and more.

In this section, you're going to implement the required `UICollectionViewDataSource`, `UICollectionViewDelegate`, and `UICollectionViewDelegateFlowLayout` methods on your view controller, so you are all set up to work with your collection view.

To start, indicate that the view controller implements the `UICollectionViewDelegate` and `UICollectionViewDataSource` protocols by adding them to the `@interface` declaration at the top of **ViewController.m**. The `@interface` line should look like this:

```
@interface ViewController : UIViewController <UITextFieldDelegate,  
UICollectionViewDataSource,  
UICollectionViewDelegateFlowLayout>
```

Note: You might wonder why `UICollectionViewDelegateFlowLayout` is listed, but `UICollectionViewDelegate` is not. This is because

`UICollectionViewDelegateFlowLayout` is actually a sub-protocol of `UICollectionViewDelegate`, so there is no need to list both.

Next it's time to implement those protocols!

UICollectionViewDataSource

Let's start with the data source. Add the following code to the end of **ViewController.m**:

```
#pragma mark - UICollectionView Datasource

// 1
- (NSInteger)collectionView:(UICollectionView *)view
    numberOfItemsInSection:(NSInteger)section {
    NSString *searchTerm = self.searches[section];
    return [self.searchResults[searchTerm] count];
}

// 2
- (NSInteger)numberOfSectionsInCollectionView:
    (UICollectionView *)collectionView {
    return [self.searches count];
}

// 3
- (UICollectionViewCell *)collectionView:(UICollectionView *)cv
    cellForItemAtIndexPath:(NSIndexPath *)indexPath {
    UICollectionViewCell *cell = [cv
        dequeueReusableCellWithReuseIdentifier:@"FlickrCell "
        forIndexPath:indexPath];
    cell.backgroundColor = [UIColor whiteColor];
    return cell;
}

// 4
/*- (UICollectionViewReusableView *)collectionView:
    (UICollectionView *)collectionView
    viewForSupplementaryElementOfKind:(NSString *)kind
    atIndexPath:(NSIndexPath *)indexPath
{
    return [[UICollectionViewReusableView alloc] init];
}*/
```

OK, but what do these methods do? Useful things, I promise. Read on:

1. `collectionView:numberOfItemsInSection:`: returns the number of cells to be displayed for a given section. Remember in this app, each search term (and its list of photo results) is in its own section. So this first finds the search term in the `searches` array, then looks up the photo results in the search term => `results` dictionary.
2. `numberOfSectionsInCollectionView`: returns the total number of sections, as you may have guessed from the name. It's a simple matter of returning the total number of searches.
3. `collectionView:cellForItemAtIndexPath`: is responsible for returning the cell at a given index path. Similarly to table view cells, collection view cells are put into a reuse queue and dequeued using a reuse identifier. You'll see in a moment how to register a specific cell class for a given reuse identifier. Unlike `UITableViewCell`, `UICollectionViewCell` doesn't have a default cell style. So the layout of the cell has to be specified by you. For now, this just returns an empty `UICollectionViewCell`.
4. `collectionView:viewForSupplementaryElementOfKind:atIndexPath` is very simple, even though it has a crazy signature. It is responsible for returning a view for either the header or footer for each section of the `UICollectionView`. The variable "kind" is an `NSString` that determines which view (header or footer) the class is asking for. This commented it out for the time being, as implementing it will cause issues in the near term. But rest assured, you will implement it later in the chapter!

UICollectionViewDelegate

Now that the `UICollectionViewDataSource` is implemented, you can turn your attention to `UICollectionViewDelegate`. Add the following code to the end of **ViewController.m**:

```
#pragma mark - UICollectionViewDelegate

- (void)collectionView:(UICollectionView *)collectionView
didSelectItemAtIndexPath:(NSIndexPath *)indexPath {
    // TODO: Select Item
}

- (void)collectionView:(UICollectionView *)collectionView
deselectItemAtIndexPath:(NSIndexPath *)indexPath {
    // TODO: Deselect item
}
```

For now, you're going to leave these methods as stubs. As their signatures indicate, these methods fire when you tap on a cell to select or deselect it. Note that `collectionView:didDeselectItemAtIndexPath:` is only called if the `UICollectionView` allows multiple selection – you'll see this for yourself later on.

UICollectionViewFlowLayoutDelegate

As I mentioned early in the section, every `UICollectionView` has an associated layout. You'll use the pre-made `UICollectionViewFlowLayout` for this project, since it's nice and easy to use and gives you the grid-view style you're looking for in this project.

Still in `ViewController.m`, add the following code to the end of the file:

```
#pragma mark - UICollectionViewDelegateFlowLayout

// 1
- (CGSize)collectionView:(UICollectionView *)collectionView
    layout:(UICollectionViewLayout*)collectionViewLayout
    sizeForItemAtIndexPath:(NSIndexPath *)indexPath {

    NSString *searchTerm = self.searches[indexPath.section];
    FlickrPhoto *photo =
        self.searchResults[searchTerm][indexPath.row];

    // 2
    CGSize retval = photo.thumbnail.size.width > 0 ?
        photo.thumbnail.size : CGSizeMake(100, 100);
    retval.height += 35;
    retval.width += 35;
    return retval;
}

// 3
- (UIEdgeInsets)collectionView:
    (UICollectionView *)collectionView
    layout:(UICollectionViewLayout*)collectionViewLayout
    insetForSectionAtIndex:(NSInteger)section {
    return UIEdgeInsetsMake(50, 20, 50, 20);
}
```

There are more delegate methods you can implement than this, but these are all you'll need for this project.

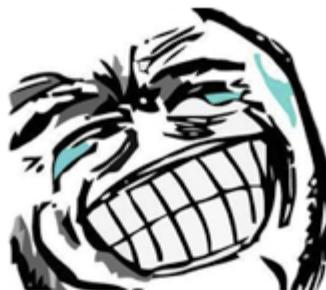
1. `collectionView:layout:sizeForItemAtIndexPath` is responsible for telling the layout the size of a given cell. To do this, you must first determine which `FlickrPhoto` you are looking at, since each photo could have different dimensions.
2. Here the ternary operator is used to determine which size should be returned. The reason this is even an issue is because you will be loading the Flickr photos asynchronously. This means that sometimes a photo might be nil or have a 0 width/height. In that case, a blank photo of size 100x100 will be displayed.

Finally, a height of 35px is added so that the photos have a nice border around them.

3. `collectionView:layout:insetForSectionAtIndex:` returns the spacing between the cells, headers, and footers.

With this infrastructure in place, you are now ready to add the `UICollectionView` and all associated subviews.

Bring it!



UICollectionView & friends

One of the great things about `UICollectionView` is like table views, Apple has made it incredibly easy to set up collection views visually in the Storyboard editor. You can drag and drop `UICollectionViews` into your view controller, and design the layout for your `UICollectionViewCells` right from within the Storyboard editor! Let's see how it works.

Adding a UICollectionView

Before you add the collection view to your storyboard, set up an `IBOutlet` so you can reference it. In `ViewController.m`, add the following to the `@interface` section:

```
@property(nonatomic, weak) IBOutlet  
UICollectionView *collectionView;
```

Now open **MainStoryboard.storyboard** and drag a collection view object (note: not a collection view controller) from the Object Library into your view. Position it just under the line image and size it so it fills all the space below:



I've set the background color of the collection view to blue so you can see its placement, but you should set the background color to transparent/clear. Otherwise, you won't be able to see the background corkboard that gives the app its defining look.



Now select the `UICollectionViewCell` that is automatically created with the collection view and delete it. Simply select the cell in the left sidebar (after expanding the collection view) and tap Delete.

Next, you need to set up the `delegate` and `dataSource` properties of the collection view. To do this, control-drag from the collection view to the view controller object in the scene inspector and select `dataSource`. Do this again, this time selecting `delegate`.

Finally, click on the view controller object in the scene inspector and switch to the Connections Inspector (you can select the last tab on the upper right sidebar or you can click `View\Utilities>Show Connections Inspector`). Drag from the `collectionView` property to your collection view inside of the storyboard to make the connection.

Now that this connection is made, I'm sure you're itching to see some data displayed. ☺ Fortunately, there are only two steps left. The first is to tell the `UICollectionView` what class it's supposed to use to create cells.

Add the following line at the end of **viewDidLoad** in **ViewController.m**:

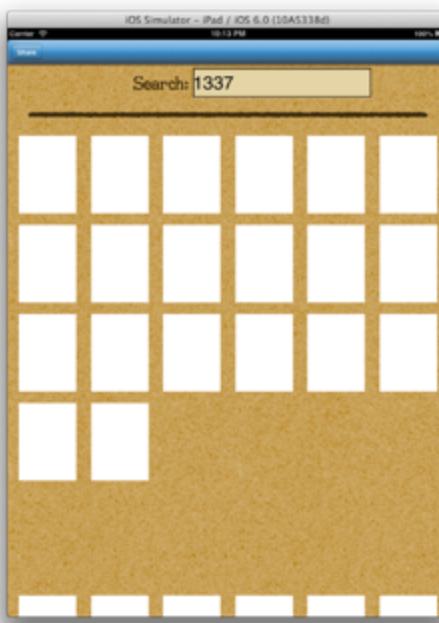
```
[self.collectionView registerClass:[UICollectionViewCell class]
    forCellWithReuseIdentifier:@"MY_CELL"];
```

Now, whenever the collection view needs to create a cell, it uses the default `UICollectionViewCell` class. You will be writing your own custom cells later, but this is just to get you up and running quickly.

The last step is to reload the collection view when new results have been found. Head back to **textFieldShouldReturn:** in **ViewController.m** and replace the comment “// Placeholder: reload collectionview data” with the following code:

```
[self.collectionView reloadData];
```

Build and run, and then start searching. As you search, you should see the view populating with white boxes. When you perform multiple searches, the boxes should have a gap in between them where the header view will go.



Congratulations – your collection view is now showing placeholder results for each row!

While it's no doubt pleasing to see that you got results back, the app still doesn't show the actual images produced by the search. Time to implement the code that fetches those images and displays them in custom `UICollectionViewCells`.

Creating custom UICollectionViewCells

By default, UICollectionViewCells don't allow for much customization beyond changing the background color. You will almost always want to create your own UICollectionViewCell subclass.

Go to **File\New\File...**, select the **iOS\Cocoa Touch\Objective-C** class template, and click **Next**. Name the class **FlickrPhotoCell**, set the subclass to **UICollectionViewCell** and click **Next**. Finally, select the location to save the file and click **Create**.

The FlickrPhotoCell will have only a single subview, which will be a **UIImageView** displaying the fetched image from Flickr. Before you create the user interface, let's set up the class for the cell. Replace the contents of **FlickrPhotoCell.h** with:

```
@class FlickrPhoto;

@interface FlickrPhotoCell : UICollectionViewCell

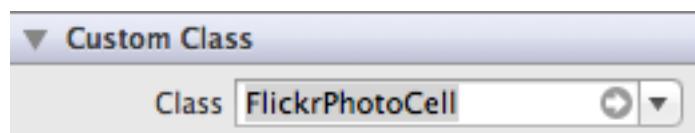
@property (nonatomic, strong) IBOutlet UIImageView *imageView;
@property (nonatomic, strong) FlickrPhoto *photo;

@end
```

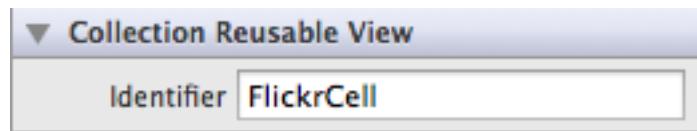
You made the **UIImageView** outlet public because the other classes might need to modify the displayed image after the photo has been asynchronously loaded. You've also added a reference to the photo that you're displaying, since you'll need that information later. Now you are ready to build the view.

When you added the **UICollectionView** to your main view, Interface Builder automatically created a **UICollectionViewCell** for you. Select **MainStoryboard.xib** to open it up in Interface Builder. Expand the list under the "Collection View" heading to reveal the cell. There are two preliminary steps that must be taken in order to use this cell, setting the cell's class and setting its identifier.

Click on the "Collection View Cell" heading and open the Identity Inspector. Type **FlickrPhotoCell** inside of the Class box to denote that this cell will be of the type **FlickrPhotoCell**.

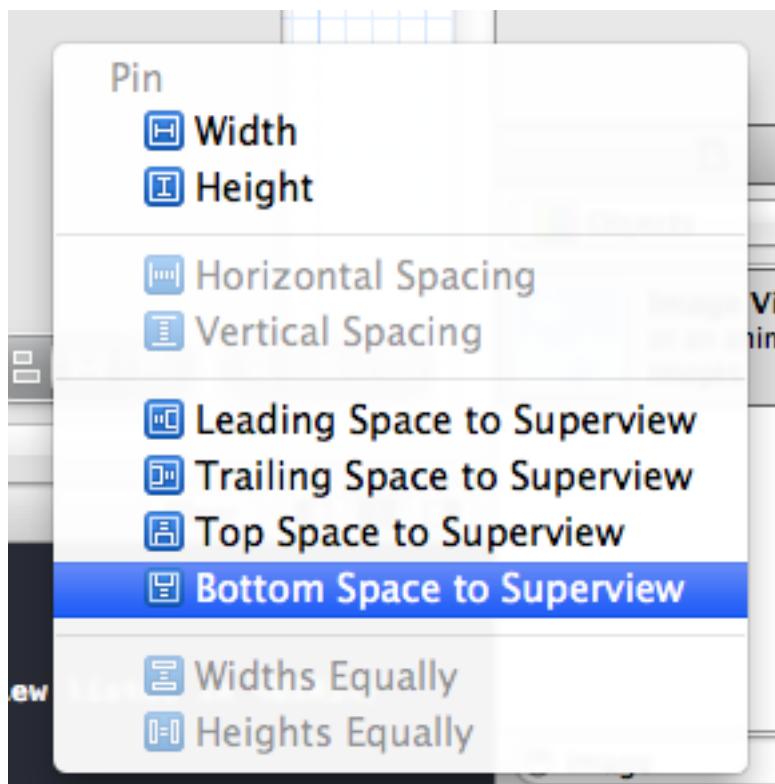


Now, open the Attributes Inspector and type in **FlickrCell** into the Identifier box. This is the reuse identifier that you will use in the `cellForItemAtIndexPath` method.



Next, drag and resize the cell to roughly 300x300 pixels to give you some room to work. The actual size here doesn't matter, as the view will be dynamically resized in the delegate.

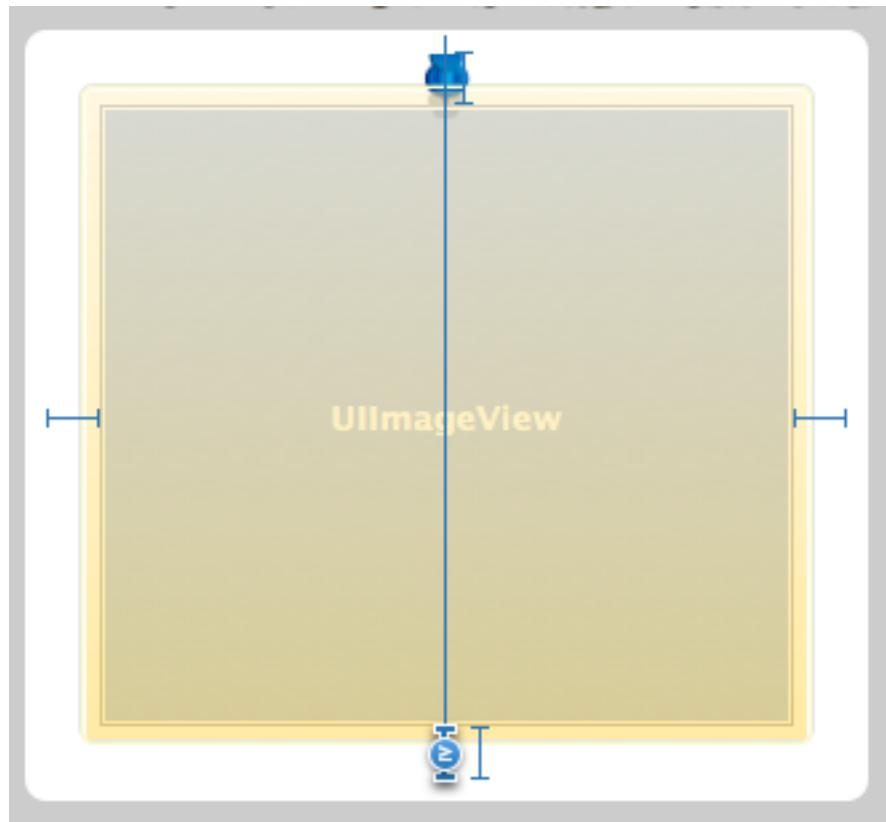
Drag an image view onto the main view. Resize it to fit within the cell view. Make sure you snap to the blue guides on all sides to ensure that the layout behaves correctly. There is only one user constraint you must add to get this to work correctly. With the image view selected, click the user constraints icon and select "Bottom Space To Superview".



Now, open the Attributes Inspector and change the mode to Aspect Fit so that the Flickr photos size appropriately.

Select the Flickr Photo Cell in the left sidebar and open up the Connections Inspector. Drag from the `imageView` outlet to your `UIImageView` to make the connection.

The last step in customizing the view is to add the pushpin at the top. Drag another image view on to your view, this time centering it at the very top of the cell. Also, change the mode to center in the Attributes Inspector and the image to `pushpin.png`. The final view should look similar to this:



With that done, you surely want to populate this view with content.

YES . . .



CONTENT

Start by telling the `UICollectionView` to use your `FlickrPhotoCell` class instead of the default `UICollectionView` class. Open `ViewController.m` and add the following import after the other imports:

```
#import "FlickrPhotoCell.h"
```

Now, replace `collectionView:cellForItemAtIndexPath:` with this:

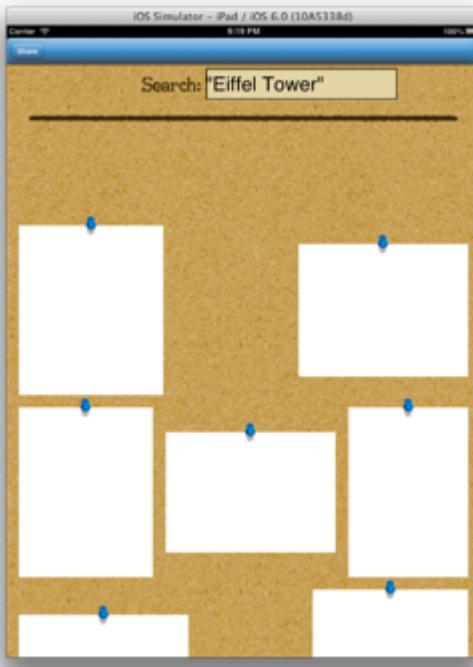
```
- (UICollectionViewCell *)collectionView:(UICollectionView *)cv
    cellForItemAtIndexPath:(NSIndexPath *)indexPath {
    FlickrPhotoCell *cell = [cv
        dequeueReusableCellWithReuseIdentifier:@"FlickrCell"
        forIndexPath:indexPath];

    NSString *searchTerm = self.searches[indexPath.section];
    cell.photo = self.searchResults[searchTerm][indexPath.row];

    return cell;
}
```

The first thing you'll notice is this code dequeues a `FlickrPhotoCell` instead of a `UICollectionViewCell`. It knows to use the one you created inside of the Storyboard based on the `FlickrCell` identifier. Next, it determines which photo you're referencing and sets the `photo` property accordingly.

Build and run the app, perform a search, and observe the results.



OK, the interface is starting to get closer to what you're after. At least it's now using your custom `UICollectionViewCell`. But why isn't it showing your photos?

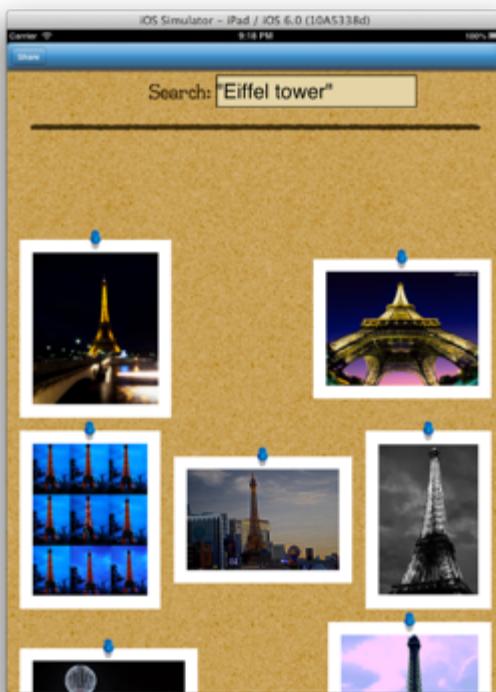
It's because when you set the photo property of the cell, you aren't updating the image of the UIImageView. To fix this, override the setter for the photo property in FlickrPhotoCell. First, add the following import to the top of **FlickrPhotoCell.m**:

```
#import "FlickrPhoto.h"
```

Then add the following to the end of the file (but before @end):

```
- (void) setPhoto:(FlickrPhoto *)photo {
    if (_photo != photo) {
        _photo = photo;
    }
    self.imageView.image = _photo.thumbnail;
}
```

Run the app again and perform a search. This time, you should see your photos appear in each cell!



Yes! Success! Notice that each photo fits perfectly inside its cell, with the cell echoing the photo's dimensions. The credit is due to the work you did inside of `sizeForItemAtIndexPath` to tell the cell size to be the size of the photo plus 35 pixels, as well as the Auto Layout settings you modified.

Note: If your view doesn't look like this or the photos are acting weird, it likely means your Auto Layout settings aren't correct. If you get stuck, try comparing your settings to the solution for this project.

The `sizeForItemAtIndexPath` code ensures that the cell is 35 pixels wider and taller than the image, and the Auto Layout rules ensure that the image view resizes and centers within the new cell frame.

At this point, you've now got a complete working (and quite cool) example of `UICollectionView` - give yourself a pat on the back!

Using `UICollectionViewReusableView` to make a header

Now let's make it even cooler. It would be nice if we could add a nice header before each set of search results, to give the user a bit more context about the photo set. You will create this header using a new class called `UICollectionViewReusableView`. Think of this class as kind of like a collection view cell, but used for other things like headers or footers.

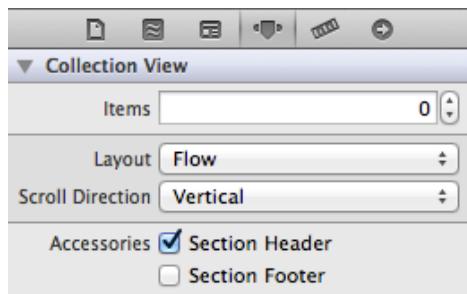
This view can be built inside of your storyboard and connected to its own class. Start off by adding a new file via **File\New\File...**, select the **iOS\Cocoa Touch\Objective-C class template** and click **Next**. Name the class **FlickrPhotoHeaderView** and make it a subclass of `UICollectionViewReusableView`. Click **Next** and then **Create** to save the file.

There are two outlets that you must set up before beginning. Open **FlickrPhotoHeaderView.m** and add the following code below the `#import` line:

```
@interface FlickrPhotoHeaderView ()  
@property(weak) IBOutlet UIImageView *backgroundImageView;  
@property(weak) IBOutlet UILabel *searchLabel;  
@end
```

This sets up a class extension where you define two `IBOutlets`. The `UILabel` will display the search text for a given group of items and the image view will be the background. The image view needs to be wired up via an outlet, since it will need to be dynamically resized to fit the `UILabel`.

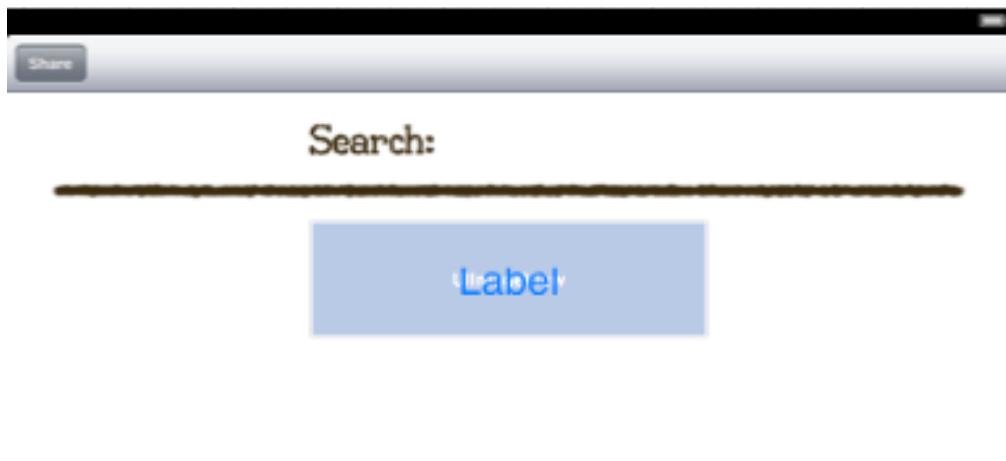
Next, open up **MainStoryboard.storyboard** and click on the collection view inside of the Scene Inspector on the left (you might need to drill down a couple of levels from the main view first). Open up the Attributes Inspector and check the Section Header box under Accessories:



If you look at the scene inspector on the left, a `UICollectionViewReusableView` has automatically been added under the Collection View. Click on the `UICollectionViewReusableView` to select it, and you can begin adding the subviews. To give you a little more space to work with, click the white handle at the bottom of the view and drag it down, making the view 90 pixels tall. (Or, you can set the size for the view explicitly via the Size Inspector.)

Drag an Image View from the Object Library onto your `UICollectionViewReusableView` and make sure that it's centered. The dimensions for the image view are not important at this point (but do make it at least 400 points or so wide), just make sure you align it with the center of the view using the guides. Alternatively, you can center the object easily by using Editor\Align from the menu and selecting horizontal and vertical centering, one after the other. Also, set the mode of the image view to center.

Next, drag a label directly on top of the image view, center it using the guide and make it as wide as the image view. Change its font size to System 32.0, set its alignment to center, and set its text color to some shade of blue. When you're done, the view should look something like this:



The last step here is to tell the `UICollectionViewReusableView` that it's a subclass of `FlickrPhotoHeaderView` and hook up the outlets you added earlier.

Click on the Collection Reusable View in the scene inspector and open the Identity Inspector. Set the class to `FlickrPhotoHeaderView`. Then open the Attributes

Inspector and set the Identifier to `FlickrPhotoHeaderView`. This is the identifier that will be used when dequeuing this view.

Also, go to the Attributes inspector and set the Reuse Identifier to **FlickrPhotoHeaderView**. This is how you will identify the header view in code.

Next, open the Outlet Inspector and drag from each of the outlets to their respective interface elements (`backgroundImageView` and `searchLabel`).

If you build and run the app, you still won't see a header (even if it is just a blank one with the word "Label"). That's because you commented out `collectionView:viewForSupplementaryElementOfKind:atIndexPath:` early on.

So let's fix that. Open **ViewController.m** and add the following import statement:

```
#import "FlickrPhotoHeaderView.h"
```

Next, replace the commented out

`collectionView:viewForSupplementaryElementOfKind:atIndexPath:` with the following code (and make sure to remove the comment delimiters!):

```
- (UICollectionViewReusableView *)collectionView:  
    (UICollectionView *)collectionView  
    viewForSupplementaryElementOfKind:(NSString *)kind  
    forIndexPath:(NSIndexPath *)indexPath {  
    FlickrPhotoHeaderView *headerView = [collectionView  
        dequeueReusableSupplementaryViewOfKind:  
            UICollectionViewElementKindSectionHeader  
            reuseIdentifier:@"FlickrPhotoHeaderView"  
            forIndexPath:indexPath];  
    NSString *searchTerm = self.searches[indexPath.section];  
    [headerView setSearchText:searchTerm];  
    return headerView;  
}
```

In the above code, you dequeue the header view for each section and set the search text for that cell. This tells the collection view which header to display for each section. The `setSearchText` method is obviously one that you haven't written yet, so you will see an error. Time to implement it!

Open **FlickrPhotoHeaderView.h** and add the following code before `@end`:

```
-(void)setSearchText:(NSString *)text;
```

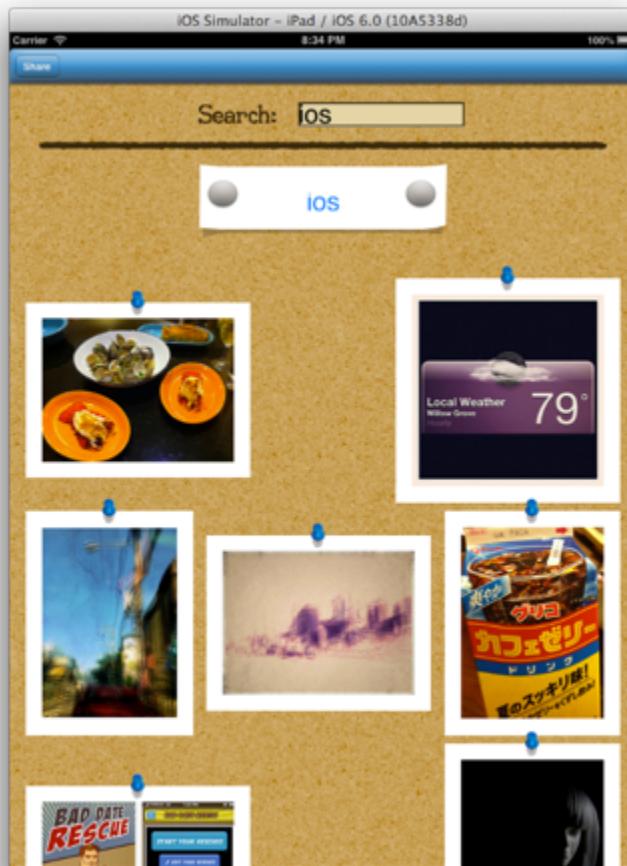
Then switch to **FlickrPhotoHeaderView.m** and add the following code:

```
-(void)setSearchText:(NSString *)text {  
    self.searchLabel.text = text;
```

```
UIImage *shareButtonImage = [[UIImage  
    imageNamed:@"header_bg.png"]  
    resizableImageWithCapInsets:  
    UIEdgeInsetsMake(68, 68, 68, 68)];  
  
self.backgroundImageView.image = shareButtonImage;  
self.backgroundImageView.center = self.center;  
}
```

setSearchText builds a new `UIImage` to span the background image, sets the label text, and then centers the text on the label.

This is a good spot to do a build and run. You will see that your UI is mostly complete.



Interacting with cells

The final section of this chapter will show you some ways to interact with collection view cells via touching and tapping. You'll take two different approaches. The first will bring up a modal view displaying the image in a larger window. The second will demonstrate how to support multiple-selection in order to share the images via email.

Single selection

Your first task is to create the modal view controller that will be displayed when the user touches a cell.

Go to **File\New\File...**, select the **iOS\Cocoa Touch\Objective-C class template** and click **Next**. On the following screen, name this class **FlickrPhotoViewController**, make it a subclass of `UIViewController`, and check **Targeted for iPad**. Make sure to leave `With xib for user interface` unchecked, as you are going to layout the view inside the storyboard. Click **Next** and then **Create** to create the class.

Open **FlickrPhotoViewController.h** and replace its contents with this code:

```
@class FlickrPhoto;

@interface FlickrPhotoViewController : UIViewController
@property(nonatomic, strong) FlickrPhoto *flickrPhoto;
@end
```

This adds a public property for the `FlickrPhoto` object that will be displayed in the modal popup.

Now, open **FlickrPhotoViewController.m** and add these imports to the top of the file:

```
#import "Flickr.h"
#import "FlickrPhoto.h"
```

Add this code inside the `@interface` section at the top:

```
@property (weak) IBOutlet UIImageView *imageView;
-(IBAction)done:(id)sender;
```

The outlet is for the image that you'll be displaying, and the action that is used when the user touches the Done button on the view to close the view.

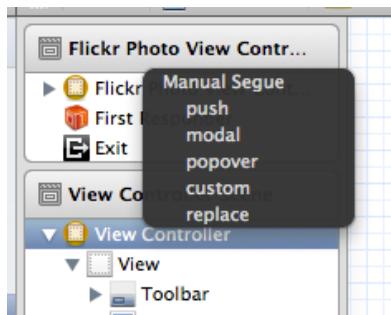
Also add a placeholder for the `done:` method to the end of the file:

```
- (IBAction)done:(id)sender {
```

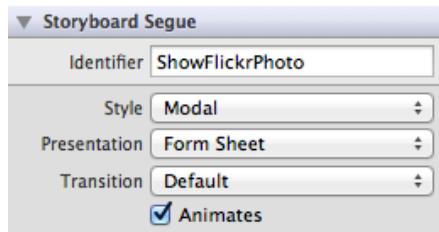
```
// TODO  
}
```

Now open **MainStoryboard.storyboard**. Drag a view controller object from the Object Library onto your main window. Select the new view controller, switch to the Identity Inspector, and change the class name to `FlickrPhotoViewController`.

Next, control-drag from your main view controller object to the new Flickr Photo View Controller and release. A context menu should pop up allowing you to create a segue. Select modal from this menu to create the segue.

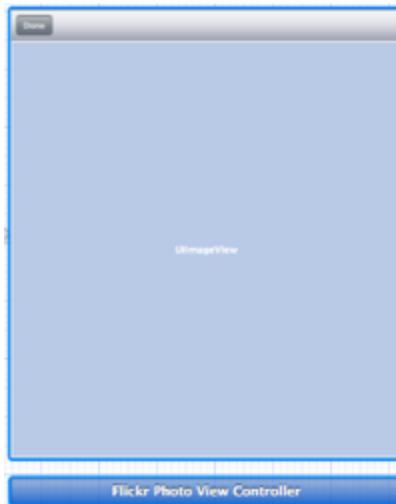


The next step is to configure the segue. Click on the segue and open the Attributes Inspector. Set the Identifier to `showFlickrPhoto` and the presentation to Form Sheet. Immediately, you should see your Flickr Photo View Controller shrink down to the size of a form sheet.



Now, drag a toolbar and an image view on to the Flickr Photo View Controller's main view. Change the text of the toolbar button to "Done" and control-drag from the button to the Flickr Photo View Controller object in the Scene Inspector. Select `done:` from the popup.

Next, control-drag from the Flickr Photo View Controller object to the image view you just put down. Select `imageView` from the popup to hook up the outlet.



Open **ViewController.m** and add the following property to the `@interface` section:

```
@property (nonatomic) BOOL sharing;
```

You'll set this boolean to true when the user is making a multi-selection to share images (which you'll implement next), but the normal setting will be false (which means tapping an image will bring up the modal detail view).

Place the following code in `collectionView:didSelectItemAtIndexPath:` (this is the callback you get for collection views when a row is tapped):

```
if (!self.sharing) {
    NSString *searchTerm = self.searches[indexPath.section];
    FlickrPhoto *photo =
        self.searchResults[searchTerm][indexPath.row];
    [self performSegueWithIdentifier:@"ShowFlickrPhoto"
        sender:photo];
    [self.collectionView
        deselectItemAtIndexPath:indexPath animated:YES];
} else {
    // Todo: Multi-Selection
}
```

If the user is not in sharing mode (for now they are not), you fetch the photo they tapped and perform the `ShowFlickrPhoto` segue. Notice that you are passing the photo as the sender. This allows you to determine which photo to display when the modal view is shown. Finally, the cell gets deselected so that it won't remain highlighted.

There is one more method you must implement in this class to make this presentation work correctly. Before a segue is performed, `prepareForSegue:sender` is called on the object performing the segue.

Make sure you import `FlickrPhotoViewController` at the top of **ViewController.m**:

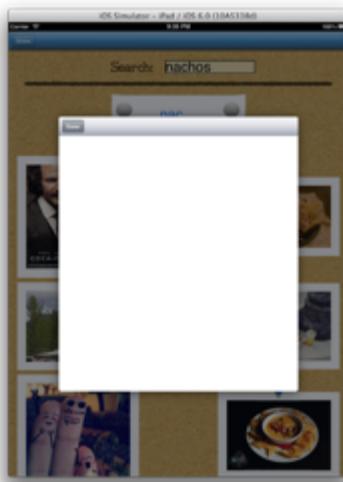
```
#import "FlickrPhotoViewController.h"
```

Next, add the following code to the end of the file:

```
#pragma mark - Segue
- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender {
    if ([segue.identifier isEqualToString:@"ShowFlickrPhoto"]) {
        FlickrPhotoViewController *flickrPhotoViewController =
            segue.destinationViewController;
        flickrPhotoViewController.flickrPhoto = sender;
    }
}
```

This method simply takes the sender of the segue (in this case, the photo tapped) and sets it as the `flickrPhoto` property of the destination view controller (an instance of `FlickrPhotoViewController` in this case). Now everything is hooked up.

Build and run, perform a search, and tap on a photo. You should see the modal view pop up with an empty image view.



Why a blank view? Why doesn't your image display? It's because you don't have any code in `FlickrPhotoViewController` to handle setting the image from the photo on to the image view.

To fix this, open up **FlickrPhotoViewController.m** and add the following code:

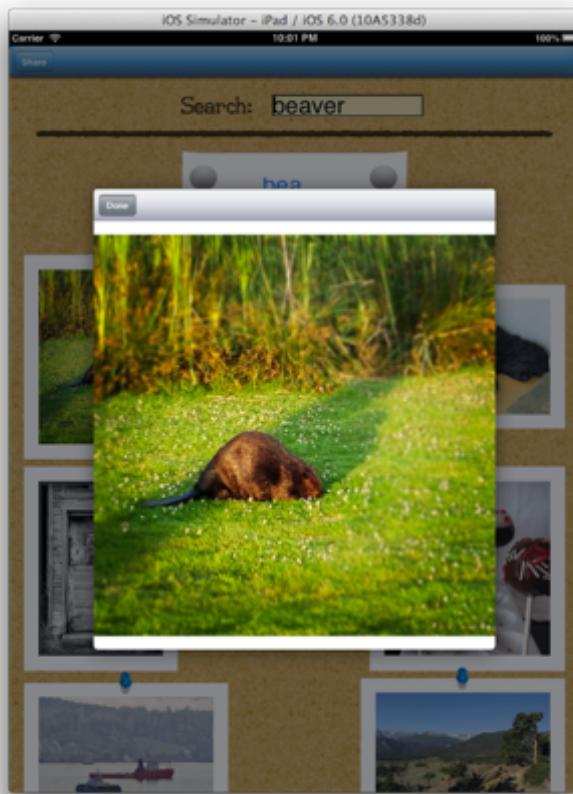
```
-(void)viewDidAppear:(BOOL)animated {
    // ...
    if(self.flickrPhoto.largeImage) {
```

```
    self.imageView.image = self.flickrPhoto.largeImage;
} else {
// 2
self.imageView.image = self.flickrPhoto.thumbnail;
// 3
[Flickr loadImageForPhoto:self.flickrPhoto thumbnail:NO
completionBlock:^(UIImage *photoImage, NSError *error) {
    if(!error) {
// 4
dispatch_async(dispatch_get_main_queue(), ^{
    self.imageView.image =
    self.flickrPhoto.largeImage;
});
}
}];
}
}
```

Let's go over this section by section.

1. If the large photo has already been fetched, simply set `imageView` to display that image.
2. If the large photo has not been fetched, display a stretched version of the thumbnail (Facebook's app uses this technique).
3. Tell Flickr to load the larger size for that photo.
4. If there wasn't an error, update the image view image in the main thread, since the photo now has a valid large photo.

Do another build and run. Perform a search and touch on a result. You should see the image presented as a modal view. It will initially appear blurry and then sharpen soon after as the larger image replaces the scaled thumbnail.



Note: If you have trouble where the thumbnail appears small before the full image loads, try setting the Content Hugging Priority and Content Compression Resistance Priority for the UIImageView to a very small amount (like 1).

Cool! Of course, if you try tapping the Done button to dismiss the image and view another image, you'll discover that the Done button does nothing. Doh, we forgot to implement the `done:` method. ☺

Add the following code to **FlickrPhotoViewController.m**, replacing the existing empty implementation of `done:`:

```
- (void)done:(id)sender {
    [self.presentingViewController
        dismissViewControllerAnimated:YES completion:^{}];
}
```

Now, when the user taps the Done button, the view will dismiss.

Multiple selection

Your final task for this chapter is to let the user select multiple photos and share them with a friend. The process for multi-selection on a `UICollectionView` is very similar to that of a `UITableView`. The only trick is to tell the collection view to allow multiple selection.

The process for selection works in the following way:

1. The user taps the Share button to tell the `UICollectionView` to allow multi-selection and set the sharing property to YES.
2. The user taps multiple photos that they want to share, adding them to an array.
3. The user taps the Done button (previously called Share), which brings up the mail composer interface.
4. Some HTML displaying the images is injected into the body of the email.
5. When the user sends the email or taps Cancel, the photos are deselected and the collection view goes back to single selection mode.

Start by creating the array that will hold the selected photos.

Open **ViewController.m** and add the following property declaration inside the `@interface` section:

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

Now add the following line at the end of `viewDidLoad`:

```
self.selectedPhotos = [@[ ] mutableCopy];
```

Now that the array has been set up, it's time to add some content to it. Replace the "Todo: Multi-Selection" comment in `collectionView:didSelectItemAtIndexPath:` with the following code:

```
NSString *searchTerm = self.searches[indexPath.section];
FlickrPhoto *photo =
    self.searchResults[searchTerm][indexPath.row];
[selectedPhotos addObject:photo];
```

This code simply determines which photo has been selected and adds it to the `selectedPhotos` array.

Next, replace the comment in `collectionView:didDeselectItemAtIndexPath:` with the following code:

```
if (self.sharing) {
    NSString *searchTerm = self.searches[indexPath.section];
    FlickrPhoto *photo =
```

```
    self.searchResults[searchTerm][indexPath.row];
    [self.selectedPhotos removeObject:photo];
}
```

This allows the user to deselect photos that they may have selected by accident.

Now, implement `shareButtonTapped:` by replacing the existing empty method with the following:

```
- (IBAction)shareButtonTapped:(id)sender {
    UIBarButtonItem *shareButton = (UIBarButtonItem *)sender;
    // 1
    if (!self.sharing) {
        self.sharing = YES;
        [shareButton setStyle:UIBarButtonItemStyleDone];
        [shareButton setTitle:@"Done"];
        [self.collectionView setAllowsMultipleSelection:YES];
    } else {
        // 2
        self.sharing = NO;
        [shareButton setStyle:UIBarButtonItemStyleBordered];
        [shareButton setTitle:@"Share"];
        [self.collectionView setAllowsMultipleSelection:NO];
    }

    // 3
    if ([self.selectedPhotos count] > 0) {
        [self showMailComposerAndSend];
    }

    // 4
    for (NSIndexPath *indexPath in
         self.collectionView.indexPathsForSelectedItems) {
        [self.collectionView
         deselectItemAtIndexPath:indexPath animated:NO];
    }
    [self.selectedPhotos removeAllObjects];
}
}
```

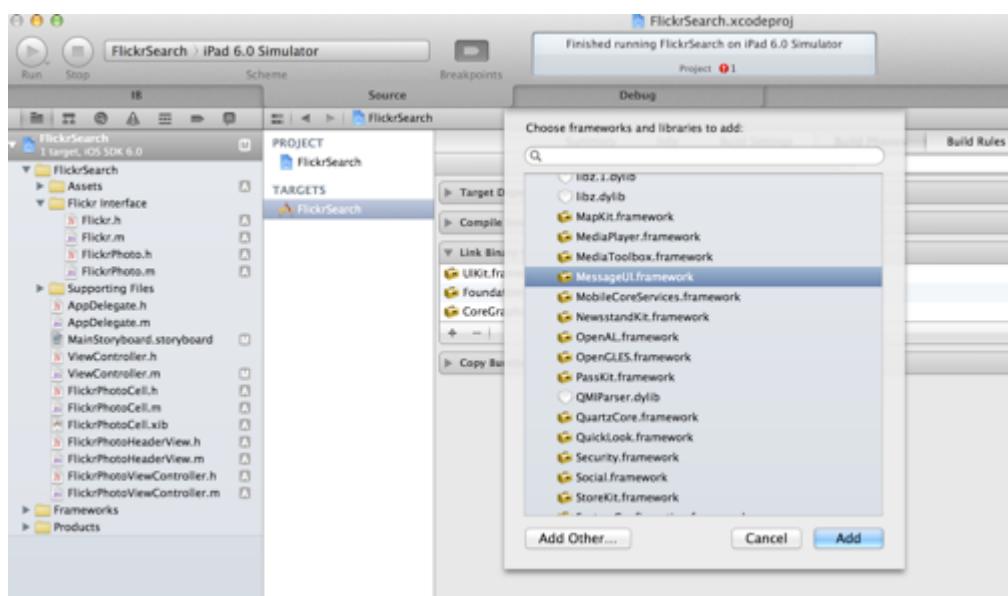
Here's what's happening in this code:

1. If the user currently isn't in sharing mode, this code sets the `UICollectionView` to allow multiple selection and changes the Share button title to Done.
2. If you got here, then the user is already in sharing mode and has tapped on the Done button. So switch the button title back to Share and disable `UICollectionView` multi-selection.

3. Check if the user has any selected photos, and if so, call `showMailComposerAndSend`.
4. Deselect all of the selected cells and remove all photos from the `selectedPhotos` array.

You won't be able to run the application yet since you still have to implement `showMailComposerAndSend`. You'll get to that momentarily.

Since this code uses `MFMailComposeViewController`, you must import the `MessageUI` framework into your project. To do this, click on the project root in the Project Navigator and then select the Flickr Search target. Then, click the **Build Phases** tab and expand the **Link Binary With Libraries** menu. Tap the (+) button, search for the **MessageUI** framework and click **Add** when you find it.



Also, be sure to add the `MessageUI` framework to the list of imports at the top of **ViewController.m**:

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

Then declare `ViewController` as supporting the `MFMailComposeViewControllerDelegate` protocol by changing the `@interface` line below the imports to:

```
@interface ViewController ()<UITextFieldDelegate,
UICollectionViewDataSource,
UICollectionViewDelegateFlowLayout,
MFMailComposeViewControllerDelegate>
```

With these preliminaries out of the way, go ahead and add `showMailComposerAndSend` to the end of the file:

```
- (void)showMailComposerAndSend {
    if ([MFMailComposeViewController canSendMail]) {
        MFMailComposeViewController *mailer =
        [[MFMailComposeViewController alloc] init];

        mailer.mailComposeDelegate = self;

        [mailer setSubject:@"Check out these Flickr Photos"];

        NSMutableString *emailBody = [NSMutableString string];
        for (FlickrPhoto *flickrPhoto in self.selectedPhotos)
        {
            NSString *url = [Flickr
                flickrPhotoURLForFlickrPhoto:
                flickrPhoto size:@"m"];
            [emailBody appendFormat:
                @"<div><img src='%@'></div><br>", url];
        }

        [mailer setMessageBody:emailBody isHTML:YES];

        [self presentViewController:mailer animated:YES
                           completion:^{}];
    } else {
        UIAlertView *alert = [[UIAlertView alloc]
            initWithTitle:@"Mail Failure"
            message:
            @"Your device doesn't support in-app email"
            delegate:nil
            cancelButtonTitle:@"OK"
            otherButtonTitles:nil];
        [alert show];
    }
}
```

This code first checks to see if the user is able to send mail. It should only return false if the user hasn't set up any mail accounts on their device. If that's the case, it alerts the user.

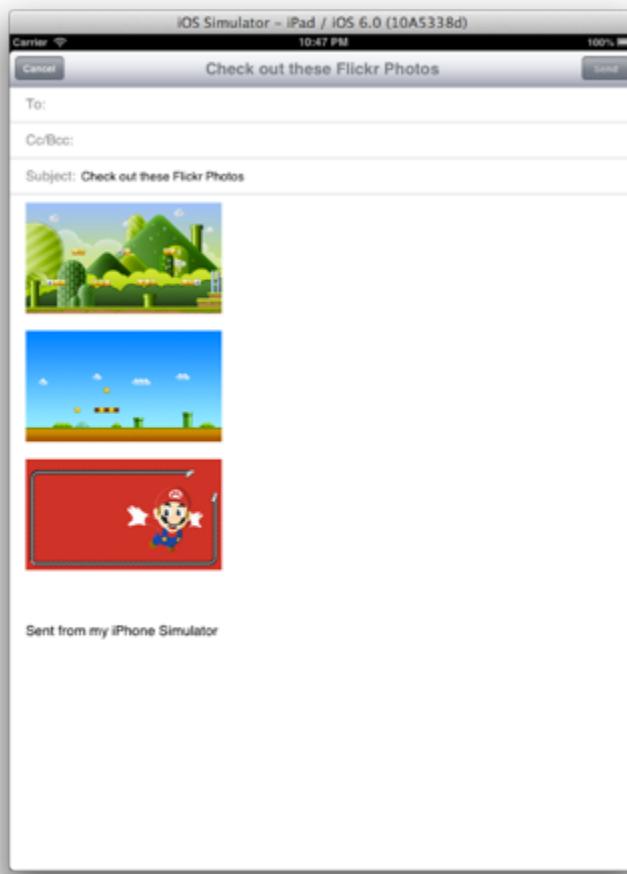
The body of the email message will be some basic HTML allowing you to display the images right in the email without adding them as attachments. Once the mail subject and body are set, the mail composer is displayed to the user.

You also need to handle user actions when the user sends the email or taps Cancel. Add the following delegate method for the mail composer at the bottom of

ViewController.m:

```
- (void)mailComposeController:(MFMailComposeViewController *)controller didFinishWithResult:(MFMailComposeResult)result error:(NSError *)error {
    [controller dismissViewControllerAnimated:YES completion:^{}];
}
```

Now the mail composer will dismiss itself after the user is done. Do a build and run and play around with sharing multiple photos.



There is one issue – when you select a photo, there's no visual indicator. The user will have no way to tell just by looking which ones they've selected and which ones they haven't. This can be fixed very easily by setting the `selectedBackgroundView` of your `FlickrPhotoCell`.

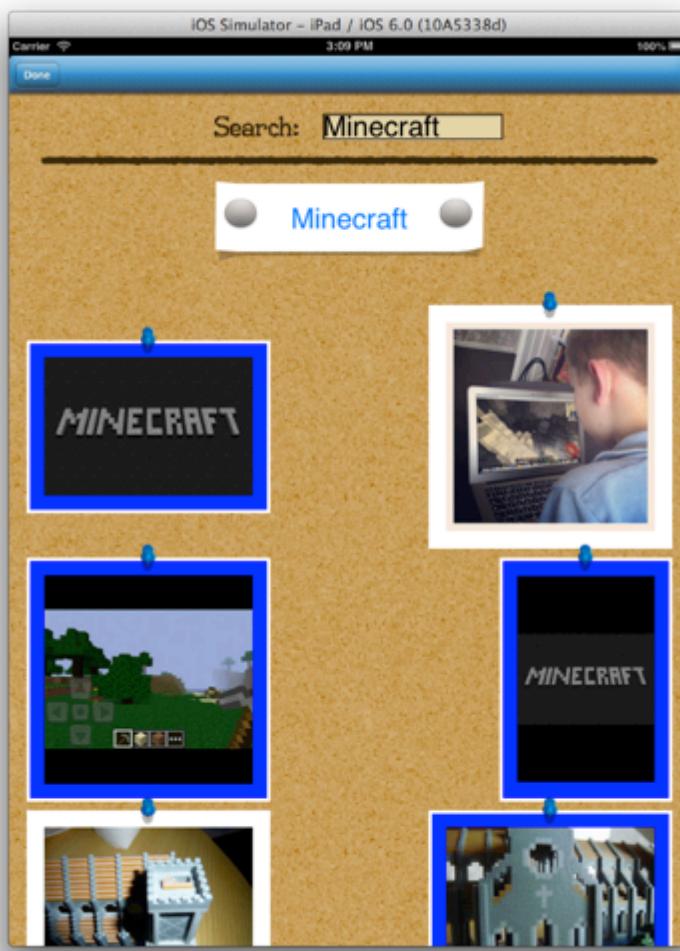
Open `FlickrPhotoCell.m`, remove `initWithFrame:`, and in its place add the following code:

```
- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder:aDecoder];
    if (self) {
        UIView *bgView = [[UIView alloc]

            initWithFrame:self.backgroundView.frame];
        bgView.backgroundColor = [UIColor blueColor];
        bgView.layer.borderColor = [[UIColor whiteColor]
                                    CGColor];
        bgView.layer.borderWidth = 4;
        self.selectedBackgroundView = bgView;
    }
    return self;
}
```

When the view is initialized from a xib file, `initWithCoder:` fires. The code creates a view with a blue background color and a white border and sets it as the `selectedBackgroundView` of the cell. Whenever the cell is in the `selected` state, the `backgroundView` is automatically swapped out for the `selectedBackgroundView` to indicate that the cell is selected.

Build and run, tap Share, and select some photos. It should look something like this:



Woot! That's a very clear indicator. Make sure deselection works as well by tapping the photos again; the blue highlighting should disappear.

Where to go from here?

Congratulations, you have finished creating your very own cool and stylish Flickr photo browser, complete with a cool `UICollectionView` based grid view!

In the process, you learned how to make custom `UICollectionViewCells`, create headers with `UICollectionViewReusableView`, detect when rows are tapped, implement multi-cell selection, and much more!

Implementing this same app prior to iOS 6 would have been a major pain and likely would have made you start searching for third party library solutions. ☺ Now, it's just as easy as using `UITableView` you know and love!

But guess what – there's more! You've only scratched the surface of what a `UICollectionView` can do. In the next chapter, you'll expand this project to support custom layouts and decoration views. You'll also implement the iPhone's home screen using a `UIPageViewController` and `UICollectionViews`. In the process, you'll learn how to add and delete elements from the collection view as well as reorder them.

I hope this chapter has got you excited about the ease of use and possibilities with the new iOS 6 collection views, and I hope to show you even more cool things you can do with them in the following chapter!

Chapter 6: Intermediate UICollectionView

By Matt Galloway

In the previous chapter, you learned a lot about `UICollectionView`, but there is much more to this class than meets the eye. `UICollectionView` supports different types of layouts, but so far you have only looked at one type of layout – the flow layout.

The flow layout is the only collection view layout that comes with the iOS 6 SDK. However, you can write your own – and by doing so you can easily create stunning interfaces and effects that previously you may have gazed upon with envy.

This chapter will walk you through creating three such custom layouts. First I'll show you how to mimic the Photos app behavior that lets users pinch photos in and out of album stacks.



Then you'll learn how to create a stacked column layout like used in [Pinterest](#). This layout has been gaining in popularity with developers because it works really well for a natural single-view display of variably sized images:



Finally, you'll implement the simple but elegant "cover flow" layout you know and love from Apple's Music app:



Most importantly, by getting your coding fingers dirty with these layouts, you'll gain advanced familiarity with the `UICollectionView` API. This will leave you primed and ready to create your own custom layouts to display data in any way you might imagine – and with many fewer lines of code than was ever before possible!

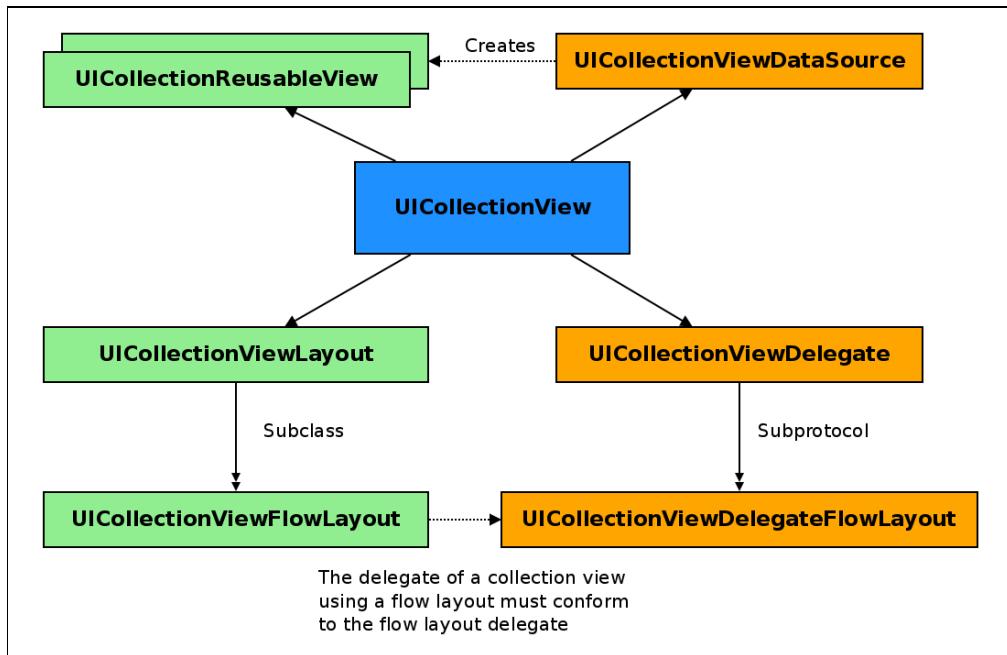
A deeper dive into collection view

Before you start to code, it's important to have a firm understanding of what `UICollectionView` does to build and maintains your layouts.

So let's take a look at all the classes related to a `UICollectionView`, how they work together, and the process by which they bring layouts to life.

Understanding the class hierarchy

The class hierarchy of `UICollectionView` is quite complex. Here's an organizational diagram (a description follows):



At the center is the `UICollectionView`, as you'd expect. When you create a `UICollectionView`, you give it a data source, a delegate, and layout:

- **The data source:** a class that implements the `UICollectionViewDataSource` protocol. Its main job is to tell the collection view about the data to display, and return `UICollectionViewReusableViews` for each element (i.e., cells, supplementary views, and decoration views). In last chapter, you set this to the view controller.
- **The delegate:** a class that implements `UICollectionViewDelegate` protocol. It is called when events happen on the collection view, such as when the user selects a cell.
- **The layout:** a subclass of `UICollectionViewLayout`. This is what this chapter is all about! Its job is to tell the collection view where to place all the different element views.

The only layout provided with UIKit called `UICollectionViewFlowLayout`. There is one slight twist when you use this layout – when you use it, the delegate of the collection view needs to be a `UICollectionViewDelegateFlowLayout`. This is a sub-protocol of `UICollectionViewDelegate` and just adds some more methods to the base protocol required for a flow layout, but not for other layouts.

Layout attributes

`UICollectionView` handles all the intricate cell management for you, and interfaces with `UICollectionViewDataSource` to find out what elements to display. You can implement the data source to tell the collection view how many sections and cells there are, and configure the cells that will eventually be displayed.

`UICollectionView` also interfaces with a `UICollectionViewLayout` subclass to figure out how to layout the cells on the screen. `UICollectionViewLayout` is an abstract base class with methods you override to create a custom layout.

Note: An abstract class is a class that you must subclass and can't instantiate directly. In Objective-C, it's not strictly true that you *can't* instantiate an instance of an abstract class, since there is no mechanism to enforce this. However, if the documentation specifies it as an abstract class, then you should subclass it.

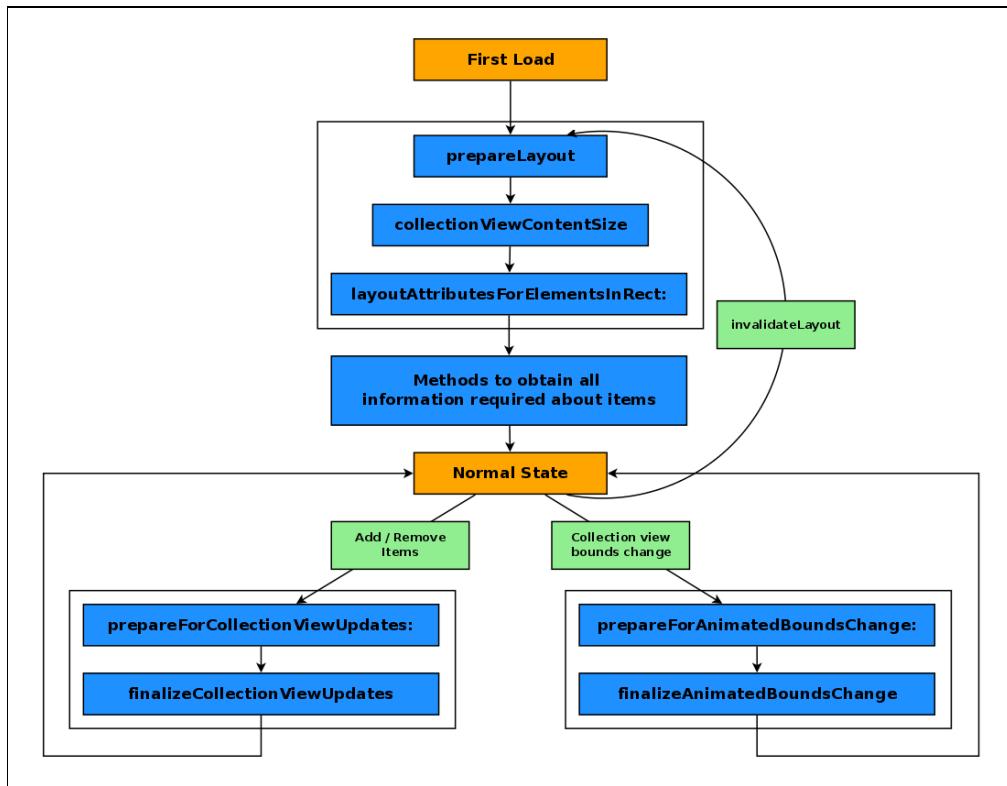
When the collection view wants to know where to place cells, supplementary views, and decoration views, it asks the layout for **layout attributes**. The class that encapsulates this data is called `UICollectionViewLayoutAttributes` and it describes the following layout details:

- **frame:** The location of the view within the coordinate system of the collection view.
- **center:** The center of the view within the coordinate system. This can be set along with the size parameter instead of the frame parameter.
- **size:** The size of the view. This can be set along with the center parameter instead of the frame parameter.
- **transform3D:** A `CATransform3D` object which is applied to the view if you want to rotate it, scale it, or transform it.
- **alpha:** The opacity of the view.
- **zIndex:** The ordering of the view with respect to other views. This can be used to define specifically whether the view should be on top of or below other views. A view with a higher z-index will be displayed on top of a view with a lower z-index.
- **hidden:** Whether or not the view is hidden.

The lifecycle of a layout

A collection view calls a lot of different methods on its layout to gather the information it needs to draw all the elements on screen in the right places. It can be a bit complicated to understand the cycle it goes through, so I've drawn a flow chart to help you.

In the diagram below, blue boxes represent methods called on the layout by the collection view, orange boxes are states that the collection view is in and green boxes are methods called by a user of the collection view. Take a quick look at the diagram, and then read on for a detailed explanation.



The cycle begins when the collection view doesn't know anything about the layout whatsoever. It first calls `prepareLayout` – this method is the layout's chance to set up any internal data structures it needs.

By this point, the collection view has determined everything about the *data* it needs, but still knows nothing about the layout. This means that you can happily call methods such as `numberOfSections` or `numberOfItemsInSection:` on the collection view from within `prepareLayout`.

Then the collection view asks its layout for the content size by calling `collectionViewContentSize`. This is where the layout must tell the collection view exactly how big the scrollable area needs to be for the user to be able to navigate to all items.

After this, the collection view calls `layoutAttributesForElementsInRect:` with the rectangle of the initially visible portion of the scrollable region. The layout must return layout attributes for all of the elements within that rectangle.

Now the collection view knows enough information to go off and ask its delegate for the views for each element and lay them out as appropriate on the screen. As the user scrolls around and new elements come into view, the collection view continues to call `layoutAttributesForElementsInRect:` to determine the layout for the newly-visible elements.

While in this state, no other methods on the layout need to be called until one of three things happens:

1. `invalidateLayout` is called on the layout. At this point, the layout will be asked again for all of the layout information, from the beginning. You will want to invalidate the layout if something changes that mean the layout attributes will change. For example if the layout attributes are dependent on a certain variable, and that variable changes, then you would invalidate the layout in its setter.
2. A method such as `insertItemsAtIndexPaths:` or `deleteItemsAtIndexPaths:` is called on the collection view. This might happen if the user adds or removes an item from the collection. This causes the collection view to call the `prepareForCollectionViewUpdates:` method on the layout to tell it that updates are about to happen. If inserting items, it asks for layout attributes for the new cells. Lastly, it calls `finalizeCollectionViewUpdates`. The collection view then goes back to the normal state.
3. The bounds of the collection view change. This might happen if the orientation changes, or you rearrange the layout on the screen. This causes the collection view to call `prepareForAnimatedBoundsChange:` on the layout. The bounds change is then applied, and `finalizeAnimatedBoundsChange` is called.

Normally, you don't need to worry too much about the intricacies of this cycle. Just be aware that it exists and refer back to it when you're wondering why a certain method just got called, or want to be certain about the sequence of events.

You'll get a lot of practice with this in this chapter, and by the end of the chapter, a lot of this may have become intuitive for you!

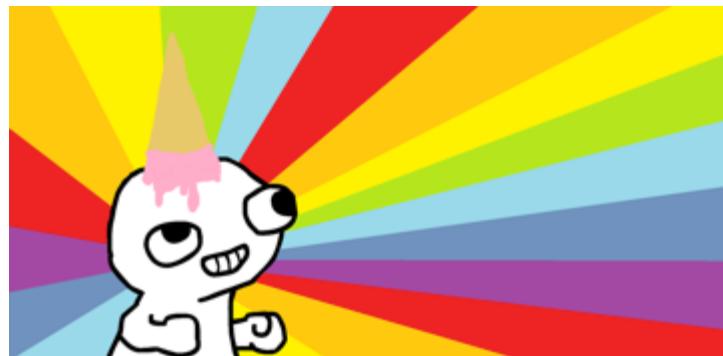
Methods to override in a custom layout

There are many methods that you can implement in a custom layout subclass. However, the system has been written in such a way that most of the time, only a few of these methods need to be implemented. Here are the three most important:

- **collectionViewContentSize** – This method returns the content size of the collection view, which means the size of the full contents of the collection view, not just the visible area. For example, a simple table would return a size with width equal to the view's width, and height equal to the sum of the cell heights.
- **layoutAttributesForItemAtIndexPath:(NSIndexPath*)** – This method takes an index path and returns the layout attributes for that cell.
- **layoutAttributesForElementsInRect:(CGRect)** – This method takes a rectangle and returns an array of layout attributes corresponding to all the elements (cells, supplementary views, and decoration views) that are visible in that rectangle.

If you remember from the last chapter, Apple has provided a default layout called `UICollectionViewFlowLayout`. It simply lays out views left-to-right and then top-to-bottom if in vertical mode, and vice-versa if in horizontal mode. You can subclass the flow layout rather than the base class if you like. You might want to do this if your custom layout is very similar to the flow layout, but just requires a few tweaks to the attributes of each cell.

In this chapter, you will subclass both the abstract base class and the flow layout to create some custom layouts. So let's get on with the layout party!



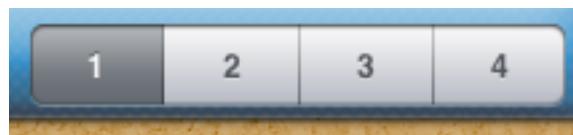
Reintroducing FlickrSearch

In this chapter, you're going to build upon the **FlickrSearch** project that you created in the previous chapter. Look in this chapter's resources to find an updated version of FlickrSearch to use as a starter project.

Note: Be sure to use the version of FlickrSearch in the chapter resources instead of the project you built last chapter, since there's a slight modification (as you'll read about next)!

If you haven't read the previous chapter, then in a nutshell, the sample app searches Flickr and displays the results in a collection view. As it is now, it uses a normal flow layout in vertical mode split into sections, one for each search term. But you're going to add some funky new layouts!

Open the FlickrSearch starter project and take a look around. Build and run it, and conduct a few searches. You will see that it's mostly the same as in the previous chapter, but with a new toggle control at the top. You will eventually wire this up to change the layout that the collection view displays.



One, two, three, four – time to start coding some more!

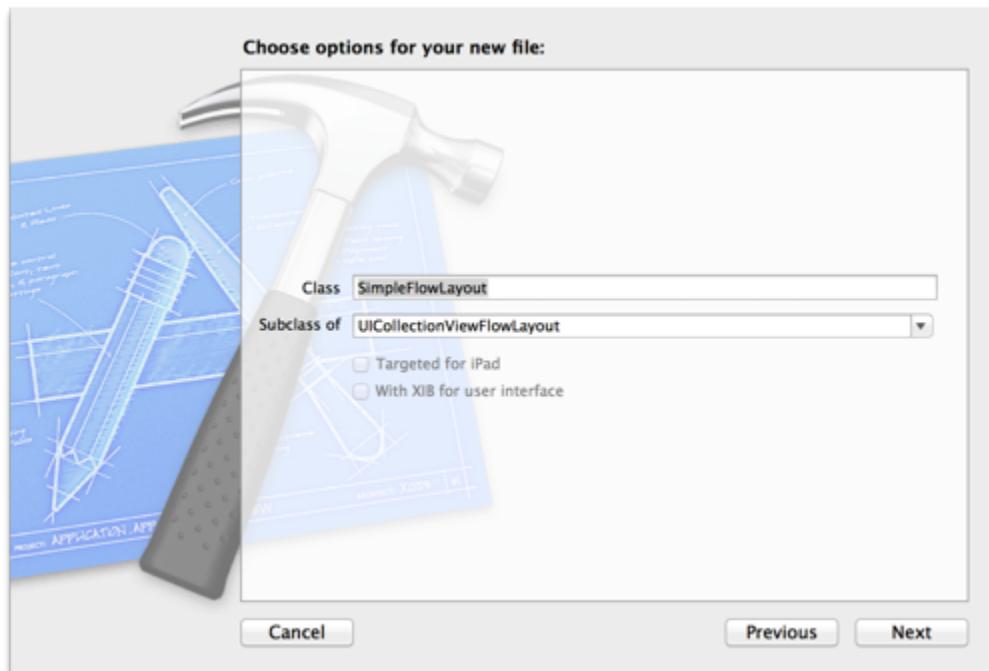
Pinch in, pinch out

First you're going to tackle the pinch layout, similar to the one used by the Photos app. For this to work, you need a flow layout to display a single photo from each search query. This single photo will act as representative of all the search results.

Then you need a pinch gesture recognizer to detect pinches over one of these photos. A pinch gesture from the user will open up *another* collection view to display the photos for that particular search query.

When you're finished, it will be just like opening up an album in the Photos app. Along the way, I'll also show you how to incorporate animations when adding and removing items from the collection view, to give the app even more spice!

First, create a new file by right-clicking on the Layout group in the project navigator, selecting **New File**, choosing the **iOS\Cocoa Touch\Objective-C class** template and clicking **Next**. Call the class **SimpleFlowLayout** and make it a subclass of **UICollectionViewFlowLayout**.



Note that it is deriving from the built-in layout – UICollectionViewFlowLayout – so you get all of that logic built-in.

The first thing you'll do is to add a couple of instance variables that will be used during the layout's operation. Open **SimpleFlowLayout.m** and add a class continuation category at the top:

```
@interface SimpleFlowLayout () {  
    NSMutableArray *_insertedIndexPaths;  
    NSMutableArray *_deletedIndexPaths;  
}  
@end
```

Next you will override some methods to make a custom animation for when cells are added and removed from the collection. To do this, open **SimpleFlowLayout.m** and add the following method:

```
- (void)prepareLayout
{
    _insertedIndexPaths = [NSMutableArray new];
    _deletedIndexPaths = [NSMutableArray new];
}
```

As you know, this method is called at the start of a layout run. Here you set up the two instance variables that are arrays to hold the index paths of inserted and deleted items. Now add the following methods:

```
- (void)prepareForCollectionViewUpdates:(NSArray*)updates
{
    [super prepareForCollectionViewUpdates:updates];
    for (UICollectionViewUpdateItem *updateItem in updates) {
        if (updateItem.updateAction ==
            UICollectionViewUpdateActionInsert)
        {
            [_insertedIndexPaths addObject:
                updateItem.indexPathAfterUpdate];
        } else if (updateItem.updateAction ==
            UICollectionViewUpdateActionDelete)
        {
            [_deletedIndexPaths addObject:
                updateItem.indexPathBeforeUpdate];
        }
    }
}

- (void)finalizeCollectionViewUpdates
{
    [_insertedIndexPaths removeAllObjects];
    [_deletedIndexPaths removeAllObjects];
}
```

These are called before and after the collection view updates. This is your chance to learn which items are being added and which are being removed. In the next couple of methods that you'll implement, you will need to know which items are currently being added or removed. So in the methods above, the inserted and deleted arrays are populated as required.

Now add the following methods:

```
- (UICollectionViewLayoutAttributes*)initialLayoutAttributesForAppearingItemAtIndexPath:(NSIndexPath*)itemIndexPath
{
```

```
if ([_insertedIndexPaths containsObject:itemIndexPath])
{
// 1
UICollectionViewLayoutAttributes *attributes =
[UICollectionViewLayoutAttributes
 layoutAttributesForCellWithIndexPath:itemIndexPath];

// 2
CGRect visibleRect =
(CGRect){.origin = self.collectionView.contentOffset,
.size = self.collectionView.bounds.size};
attributes.center = CGPointMake(CGRectGetMidX(visibleRect),
                                CGRectGetMidY(visibleRect));
attributes.alpha = 0.0f;
attributes.transform3D = CATransform3DMakeScale(0.6f,
                                              0.6f,
                                              1.0f);

// 3
return attributes;
} else {
return
[super initialLayoutAttributesForAppearingItemAtIndexPath:
 itemIndexPath];
}

-
UICollectionViewLayoutAttributes*)finalLayoutAttributesForDisappe
aringItemAtIndexPath:(NSIndexPath*)itemIndexPath
{
if ([_deletedIndexPaths containsObject:itemIndexPath]) {
UICollectionViewLayoutAttributes *attributes =
[UICollectionViewLayoutAttributes
 layoutAttributesForCellWithIndexPath:itemIndexPath];

CGRect visibleRect =
(CGRect){.origin = self.collectionView.contentOffset,
.size = self.collectionView.bounds.size};
attributes.center = CGPointMake(CGRectGetMidX(visibleRect),
                                CGRectGetMidY(visibleRect));
attributes.alpha = 0.0f;
attributes.transform3D = CATransform3DMakeScale(1.3f,
                                              1.3f,
                                              1.0f);
}
```

```
    return attributes;
} else {
    return
[super finalLayoutAttributesForDisappearingItemAtIndexPath:
itemIndexPath];
}
}
```

The two methods above tell the layout how to make cells appear and disappear when they are added and removed from the collection view.

The first method, `initialLayoutAttributesForAppearingItemAtIndexPath:`, is called when a new cell is added to the collection view and you need to return the layout attributes *from where* the cell should animate. It will animate *to* the normal place where it would be in the layout.

Here's a breakdown of the code:

1. You create layout attributes using the class method to get attributes for a normal cell at a given index path.
2. You set the various attributes. Here you set the cell so that when it appears it starts at the center of the current visible portion of the collection view, at an alpha of 0.0 and slightly scaled-down. This will provide the illusion that the cell is fading in from the middle of the view, and should look pretty.
3. You return the final attributes.

The rest of the code makes use of the second method, `finalLayoutAttributesForDisappearingItemAtIndexPath:`. This method is called when a cell is removed from the collection view, and it's largely a reverse of the first method. In the implementation above, you return layout attributes such that the cell animates back to the center of the collection view, fades out and scales up slightly, giving a nice-looking animation.

Both of the methods above first check if the index path being asked for is in the inserted or deleted items arrays respectively. This is because the collection view will ask its layout for initial and final attributes for *all* visible items even when just a single item is added or removed. So for items that are in the process of being added or removed you return the attributes to make the fancy attributes. In other cases super is called to hand off behavior to the super class.

Now open **ViewController.m** and import this new layout by adding the following line to the imports at the top:

```
#import "SimpleFlowLayout.h"
```

Then add a property to the class extension (the `@interface` section):

```
@property (nonatomic, strong) SimpleFlowLayout *layout2;
```

Finally, set up your new layout by adding the following code to the bottom of viewDidLoad:

```
self.layout2 = [[SimpleFlowLayout alloc] init];
self.layout2.scrollDirection =
UICollectionViewScrollDirectionVertical;
```

This layout will be displayed when you tap the **2** segment in the UI, so you need to wire this up in the action called when the segmented control is changed. Find the method called layoutSelectionTapped: and edit case 1: (i.e., the second button) to make it look like this:

```
case 1: {
    self.collectionView.collectionViewLayout = self.layout2;
}
break;
```

Build and run the app now, select the second layout using the segmented control at the top right, and perform a search. You should see the same layout of photos as you'd expect with a standard flow layout, but this time the photos all animate in nicely. w00t!



So far so good – but there's a lot more to do, so let's "crack on", as we say in England! 😊

Showing a single photo from each search query

The next step on your way to photo-pinching goodness is to make the layout show just the first photo of each search query, rather than every single photo.

To do this, you need to go through all the data source methods for the collection view and switch based on which layout is currently in use. When it's layout **2** (an instance of `SimpleFlowLayout`), there should be just one section with an item for each search query.

First find `numberOfSectionsInCollectionView:` (in **ViewController.m**) and make it look like this:

```
- (NSInteger)numberOfSectionsInCollectionView:  
    (UICollectionView*)cv {  
    if (cv == self.collectionView) {  
        if (cv.collectionViewLayout == self.layout2)  
        {  
            return 1;  
        } else {  
            return [self.searches count];  
        }  
    }  
    return 0;  
}
```

Now if the collection view's current layout is `layout2`, i.e., the one you created above, then the number of sections should be 1; otherwise the number of sections should be the number of searches that have been made.

Now find `collectionView:numberOfItemsInSection:` and make it look like this:

```
- (NSInteger)collectionView:(UICollectionView*)cv  
    numberOfItemsInSection:(NSInteger)section  
{  
    if (cv == self.collectionView) {  
        if (cv.collectionViewLayout == self.layout2)  
        {  
            return [self.searches count];  
        } else {  
            NSString *searchTerm = self.searches[section];  
            return [self.searchResults[searchTerm] count];  
        }  
    }  
    return 0;  
}
```

The code has been changed to return the number of search queries for the new layout, i.e., the number of photos in that particular section. Everything else is as before.

Finally, find `collectionView:cellForItemAtIndexPath:` and make it look like this:

```

- (UICollectionViewCell*)collectionView:(UICollectionView*)cv
    cellForItemAtIndexPath:(NSIndexPath*)indexPath
{
    FlickrPhotoCell *cell =
        [cv dequeueReusableCellWithReuseIdentifier:@"MY_CELL"
            forIndexPath:indexPath];

    FlickrPhoto *photo = nil;
    if (cv == self.collectionView) {
        if (cv.collectionViewLayout == self.layout2) {
            NSString *searchTerm =
                self.searches[indexPath.item];
            photo =
                self.searchResults[searchTerm][0];
        } else {
            NSString *searchTerm =
                self.searches[indexPath.section];
            photo =
                self.searchResults[searchTerm][indexPath.item];
        }
    }
    cell.photo = photo;

    return cell;
}

```

In this method, you set up a cell and return it. In the case of layout 2, you use the first photo from each set of search results, and for other layouts it is as before.

There are a few other delegate methods that need changing. First, change `collectionView:didSelectItemAtIndexPath:` to look like this:

```

- (void)collectionView:(UICollectionView *)cv
didSelectItemAtIndexPath:(NSIndexPath *)indexPath
{
    if (self.sharing) {
        NSString *searchTerm = self.searches[indexPath.section];
        FlickrPhoto *photo =
            self.searchResults[searchTerm][indexPath.item];
        [self.selectedPhotos addObject:photo];
    } else {
        FlickrPhoto *photo = nil;
        if (cv == self.collectionView) {
            // CHANGED
            if (cv.collectionViewLayout == self.layout2) {

```

```

        NSString *searchTerm =
            self.searches[indexPath.item];
        photo = self.searchResults[searchTerm][0];
    } else {
        NSString *searchTerm =
            self.searches[indexPath.section];
        photo =
            self.searchResults[searchTerm][indexPath.item];
    }
}
[self performSegueWithIdentifier:@"ShowFlickrPhoto"
                           sender:photo];
[self.collectionView
    deselectItemAtIndexPath:indexPath
    animated:YES];
}
}

```

Again, the photo selected in the case of layout 2 is the first in the relevant search query.

Now make `collectionView:layout:sizeForItemAtIndexPath:` look like this:

```

- (CGSize)collectionView:(UICollectionView*)cv
    layout:(UICollectionViewLayout*)cvl
sizeForItemAtIndexPath:(NSIndexPath*)indexPath
{
    FlickrPhoto *photo = nil;
    if (cv == self.collectionView) {
        // CHANGED
        if (cvl == self.layout2) {
            NSString *searchTerm =
                self.searches[indexPath.item];
            photo = self.searchResults[searchTerm][0];
        } else {
            NSString *searchTerm =
                self.searches[indexPath.section];
            photo =
                self.searchResults[searchTerm][indexPath.item];
        }
    }

    CGSize retval = photo.thumbnail.size.width > 0.0f ?
        photo.thumbnail.size : CGSizeMake(100.0f, 100.0f);
    retval.height += 35.0f;
    retval.width += 35.0f;
}

```

```
    return retval;
}
```

Once again, you find the right photo for the index path in the case of layout 2.

The last thing you need to do is alter what happens after the user does a new search. Previously, the layout added new sections each time the `insertSections:` method of `UICollectionView` was called. For the new layout, there won't be a new section added, but rather, just a single new item.

Find the `textFieldShouldReturn:` method where the Flickr search is initiated, and find the portion of code that begins with the comment, `// RUN AFTER SEARCH HAS FINISHED`. Make that portion of code look like this:

```
dispatch_async(dispatch_get_main_queue(), ^{
// RUN AFTER SEARCH HAS FINISHED
if (self.collectionView.collectionViewLayout == self.layout2) {
    [self.collectionView performBatchUpdates:^{
        [self.collectionView insertItemsAtIndexPaths:@[
            [NSIndexPath indexPathForItem:(self.searches.count-1)
                inSection:0]]];
    } completion:nil];
} else {
    [self.collectionView performBatchUpdates:^{
        [self.collectionView insertSections:
            [NSSet setWithIndex:0]];
    } completion:nil];
}
});
```

As with the previous changes, you once again switch based on the current layout. If it's layout 2, then just a single item is added to the view, whereas otherwise a full new section is added.

Now build and run the app, select layout 2 and search for a few different terms. Notice how each search query now produces just one photo, and that they animate nicely into view. You should see something like this:



Deleting searches

You added an animation for removing cells from the view, and you may be wondering how to go about seeing this in action! Currently there is no functionality in the app to delete a search query. You'll need to add that in to see the animation.

To do this, you'll first use a gesture recognizer to detect a long press on a cell. This action (the long press) will delete the results associated with that query.

Add a property in the class extension to hold the long press gesture recognizer:

```
@property (nonatomic, strong)  
UILongPressGestureRecognizer *longPressGestureRecognizer;
```

Then add the following code to the end of `viewDidLoad`:

```
self.longPressGestureRecognizer =  
[[UILongPressGestureRecognizer alloc]  
initWithTarget:self  
action:@selector(handleLongPressGesture:)];
```

This creates a gesture recognizer that calls a method named `handleLongPressGesture:` after a long press has been recognized. You now need to add the gesture recognizer to the collection view, but you only want to have it in layout 2. A good place to put it is back in the `layoutSelectionTapped:` method, as

that's run to switch to the right layout. Edit `case 0` and `case 1` in that method to look like this:

```
case 0:  
default: {  
    self.collectionView.collectionViewLayout = self.layout1;  
    [self.collectionView  
        removeGestureRecognizer:self.longPressGestureRecognizer];  
}  
    break;  
case 1: {  
    self.collectionView.collectionViewLayout = self.layout2;  
    [self.collectionView  
        addGestureRecognizer:self.longPressGestureRecognizer];  
}  
    break;
```

Here you add the gesture recognizer when using `layout2` and remove it when using the other layout, so that long presses are only detected when `layout2` is active.

Now you need to implement the callback method for when the gesture is recognized. Add the following method to **ViewController.m**:

```
-  
(void)handleLongPressGesture:(UILongPressGestureRecognizer*)recognizer {  
    // 1  
    if (recognizer.state == UIGestureRecognizerStateRecognized)  
    {  
        // 2  
        CGPoint tapPoint =  
            [recognizer locationInView:self.collectionView];  
  
        // 3  
        NSIndexPath *item =  
            [self.collectionView  
                indexPathForItemAtPoint:tapPoint];  
  
        // 4  
        if (item) {  
            // 5  
            NSString *searchTerm = self.searches[item.item];  
  
            // 6  
            [self.searches removeObjectAtIndex:item.item];  
            [self.searchResults removeObjectForKey:searchTerm];  
        }  
    }  
}
```

```
// 7
[self.collectionView performBatchUpdates:^{
    [self.collectionView
        deleteItemsAtIndexPaths:@[item]];
} completion:nil];
}
}
```

Here's what this method does:

1. If the recognizer is in the "recognized" state, i.e., a long press has been detected, then...
2. You find the point within the collection view's coordinate system where the long press occurred.
3. You find the collection view item that was tapped based on the tap location.
4. If you find an item (there's no guarantee: the long tap might have been in whitespace between items) then...
5. You find the search term for the cell by looking at the item. Remember that there is just one section now containing a cell for each search term.
6. You remove the data from the search results and the list of queries.
7. Finally, you tell the collection view to delete the item, which will cause it to animate away.

That wasn't so hard, was it? Now build and run the app, select layout 2 once more and search for something. Then long-press and release on the single photo and watch it animate away! Boom!



Note that at the time of writing this chapter, there is currently a bug with UICollectionView where when you delete an item it makes everything fade out, and even the ones that weren't deleted fade in. In previous versions of iOS 6 this worked fine, so I expect this will be fixed sometime soon.

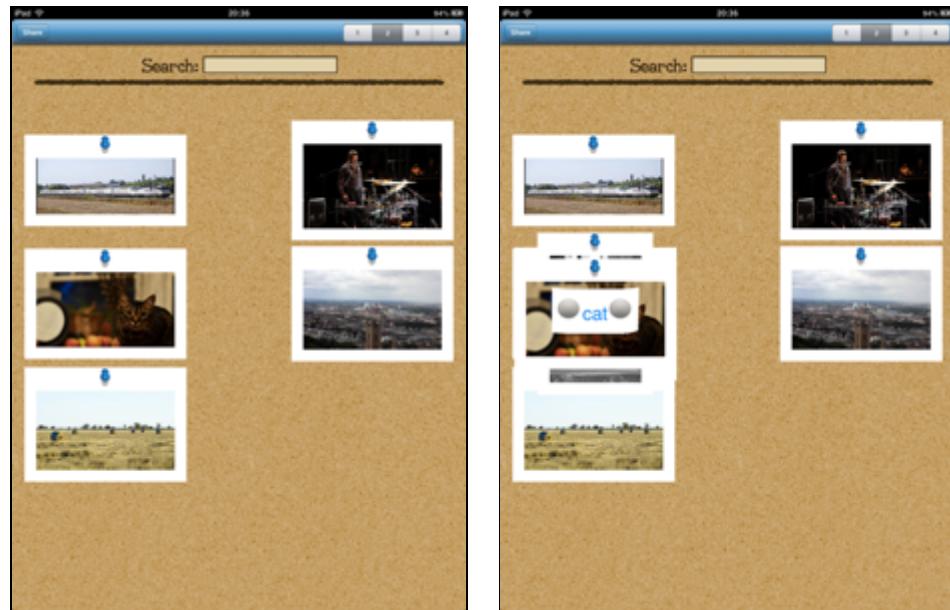
Pinch me, I must be dreaming!

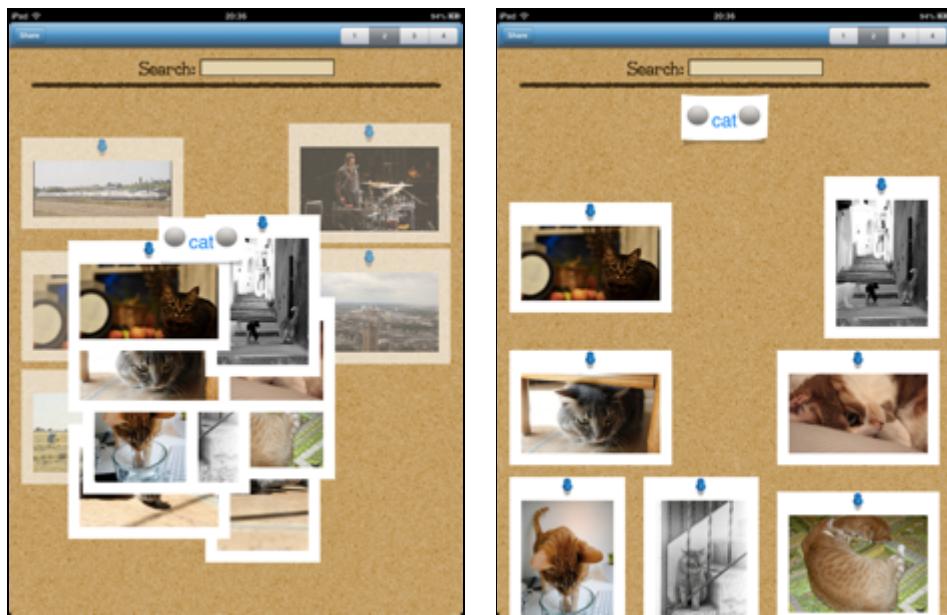
Now you're getting to the good stuff – you're going to implement the pinch part of this layout! This is quite tricky and will take more than a bit of coding. So take a deep breath, read through this section carefully, and enjoy!

You're going to add a gesture recognizer to the collection view to detect when the user pinches out on a cell to expand it. The moment a pinch is detected, another collection view will be placed on top of the original one. The contents of the new collection view will display all the photos in the search query related to the one that was pinched. Initially, all the photos will be stacked one upon the other, but as the pinch expands, so will the photos.

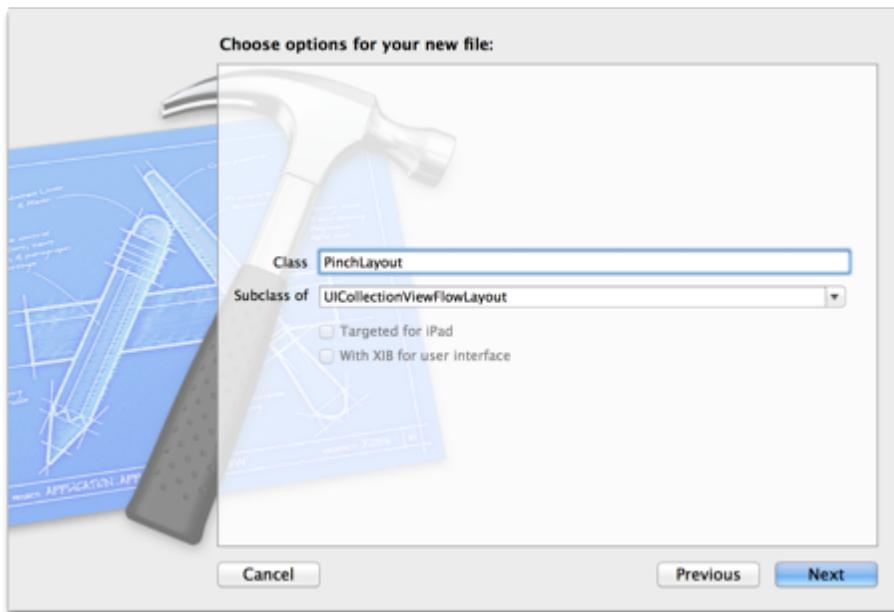
You will also add a gesture recognizer to the new collection view to detect a pinch inwards that will dismiss the expanded view. This new collection view will need a custom layout. It will be a subclass of `UICollectionViewFlowLayout`, but will be told the current pinch scale and the center coordinates of the pinch so that it can adjust the layout attributes of the cells accordingly.

Below are screenshots showing the pinch in action. Going from left to right and then top to bottom, you can see the view before the pinch, as the pinch is happening, and finally you can see the photos fully “pinched out.”





You know you want it! Add a new file to the **Layout** project group with the **iOS\Cocoa Touch\Objective-C class** template. Name the class **PinchLayout**, and make it a subclass of **UICollectionViewFlowLayout**.



Then open **PinchLayout.h** and replace its contents with this:

```
@interface PinchLayout : UICollectionViewFlowLayout  
  
@property (nonatomic, assign) CGFloat pinchScale;  
@property (nonatomic, assign) CGPoint pinchCenter;
```

```
@end
```

These properties keep track of the center of the pinch, and a scale factor ranging from 0 to 1 (0=stacked together, 1=fully pinched out). You'll use these to move the photos around in the layout.

Now open **PinchLayout.m** and add the following method:

```
- (UICollectionViewLayoutAttributes*)  
    layoutAttributesForItemAtIndexPath:(NSIndexPath*)indexPath  
{  
    UICollectionViewLayoutAttributes *attributes =  
        [super layoutAttributesForItemAtIndexPath:indexPath];  
    [self applySettingsToAttributes:attributes];  
    return attributes;  
}
```

This is the first of the methods of `UICollectionViewLayout` that you need to subclass. It is called by the collection view when it needs to know layout attributes for a certain item.

Since `PinchLayout` is a subclass of `UICollectionViewFlowLayout`, the attributes are first called from the super-class implementation, as this will return the attributes of the location where the flow layout wants to put the item. Then you call a method that will modify the various settings based on the current pinch values.

Next add the following method:

```
- (NSArray*)layoutAttributesForElementsInRect:(CGRect)rect {  
    // 1  
    NSArray *layoutAttributes =  
        [super layoutAttributesForElementsInRect:rect];  
  
    // 2  
    [layoutAttributes enumerateObjectsUsingBlock:  
        ^(UICollectionViewLayoutAttributes *attributes,  
            NSUInteger idx, BOOL *stop)  
    {  
        [self applySettingsToAttributes:attributes];  
    }];  
  
    // 3  
    return layoutAttributes;  
}
```

This method is another override of `UICollectionViewLayout` and is called whenever the collection view needs to know the layout attributes for elements that are visible within a certain rectangle.

It is your job to return layout attributes for all relevant elements. Here is what the method does:

1. You defer to the super-class implementation to get a list of the elements that it thinks are in the rectangle.
2. You enumerate over the attributes and, as with the previous method, you call the method to modify the settings based on the current pinch values.
3. Finally, you return the list of attributes.

And now add the method to modify the settings based on the current pinch values:

```
- (void)applySettingsToAttributes:  
    (UICollectionViewLayoutAttributes*)attributes  
{  
    // 1  
    NSIndexPath *indexPath = attributes.indexPath;  
    attributes.zIndex = -indexPath.item;  
  
    // 2  
    CGFloat deltaX = self.pinchCenter.x - attributes.center.x;  
    CGFloat deltaY = self.pinchCenter.y - attributes.center.y;  
    CGFloat scale = 1.0f - self.pinchScale;  
  
    // 3  
    CATransform3D transform =  
        CATransform3DMakeTranslation(deltaX * scale,  
                                    deltaY * scale,  
                                    0.0f);  
    attributes.transform3D = transform;  
}
```

This is what the method does:

1. You set the z-index of the cell to the negative of the item index. This means that the first photo in the list will appear on top of the stack when the view is pinched fully inwards. That's how it should look when you pinch out, since you're picking up the whole album and expanding it.
2. The attributes you're passing into this method are what the super-class (i.e., `UICollectionViewFlowLayout`) has decided upon. In order to get the effect that the user is pinching out the photos, you need to translate each photo so that when the pinch scale is 0 (i.e., at the beginning of a pinch), all the photos are on top of one another.

The way I've done it here is to determine the distance between the pinch center and the attribute's center (i.e., the natural position of the cell). This distance will be used to translate according to the current pinch scale. The scale factor by which the cell needs to be moved is 1 minus the pinch scale, so that at a pinch scale of 0, all the cells will be fully translated towards the pinch center point, and at a pinch scale of 1, all cells will be in their natural position.

3. Finally, you generate a transform to move the cell the required amount and apply it to the attributes.

This method may seem a bit complicated, but if you sit down and think about it, you should be able to understand the logic here.

X, Y, Z... Got it!



Finally, you need to implement two more methods in `PinchLayout`. These are the setters for `pinchScale` and `pinchCenter`. So add the following methods:

```
- (void)setPinchScale:(CGFloat)pinchScale {
    _pinchScale = pinchScale;
    [self invalidateLayout];
}

- (void)setPinchCenter:(CGPoint)pinchCenter {
    _pinchCenter = pinchCenter;
    [self invalidateLayout];
}
```

In both of these methods, you are invalidating the layout when either property changes. This will cause the collection view to recalculate the layout and ask once again for all the layout attributes. You need to do this, since the layout attributes depend upon both of these values.

The next step is wiring up this layout in the main view controller. Open `ViewController.m` and add an import for the new layout:

```
#import "PinchLayout.h"
```

Within the pinch gesture recognizer, you will be determining how far out the pinch currently is by looking at the `UIPinchGestureRecognizer's scale` property. You need to define a minimum and maximum that you care about. So to hold these values, add the following code above below the `#import` lines:

```
static const CGFloat kMinScale = 1.0f;
static const CGFloat kMaxScale = 3.0f;
```

Note: You'll often see these sorts of constants done using a `#define`. I prefer the `static const` approach, because of the type safety it provides. The compiler will do the optimization of not actually including the value as a symbol, but just replacing it wherever it is used in the code. Be careful, though, as it can only do this optimization if the variable is declared as a `const`.

Now add some properties to the class extension:

```
@property (nonatomic, strong)
    UIPinchGestureRecognizer *pinchOutGestureRecognizer;
@property (nonatomic, strong)
    UICollectionView *currentPinchCollectionView;
@property (nonatomic, strong) NSIndexPath *currentPinchedItem;
```

These properties hold the gesture recognizer to detect the pinching out, the collection view added for the currently pinched search query, and the currently pinched item, respectively.

Next, add the following to the end of `viewDidLoad` to set up the pinch out gesture recognizer:

```
self.pinchOutGestureRecognizer =
[[UIPinchGestureRecognizer alloc]
    initWithTarget:self
    action:@selector(handlePinchOutGesture:)];
```

This recognizer needs to be added to the main collection view, so in `layoutSelectionTapped:`, make case 0 and case 1 look like this:

```
case 0:
default: {
    self.collectionView.collectionViewLayout = self.layout1;
    [self.collectionView
        removeGestureRecognizer:self.pinchOutGestureRecognizer];
    [self.collectionView
        removeGestureRecognizer:self.longPressGestureRecognizer];
```

```
        }
        break;
    case 1: {
        self.collectionView.collectionViewLayout = self.layout2;
        [self.collectionView
            addGestureRecognizer:self.pinchOutGestureRecognizer];
        [self.collectionView
            addGestureRecognizer:self.longPressGestureRecognizer];
    }
    break;
}
```

Just as with the long press recognizer added earlier, the pinch out recognizer is added and removed as required.

Now implement the gesture recognizer method:

```
- (void)handlePinchOutGesture:
    (UIPinchGestureRecognizer*)recognizer
{
    // 1
    if (recognizer.state == UIGestureRecognizerStateBegan) {
        // 2
        CGPoint pinchPoint =
            [recognizer locationInView:self.collectionView];
        NSIndexPath *pinchedItem =
            [self.collectionView
                indexPathForItemAtPoint:pinchPoint];

        if (pinchedItem) {
            // 3
            self.currentPinchedItem = pinchedItem;

            // 4
            PinchLayout *layout = [[PinchLayout alloc] init];
            layout.itemSize = CGSizeMake(200.0f, 200.0f);
            layout.minimumInteritemSpacing = 20.0f;
            layout.minimumLineSpacing = 20.0f;
            layout.sectionInset =
                UIEdgeInsetsMake(20.0f, 20.0f, 20.0f, 20.0f);
            layout.headerReferenceSize =
                CGSizeMake(0.0f, 90.0f);
            layout.pinchScale = 0.0f;

            // 5
            self.currentPinchCollectionView =
                [[UICollectionView alloc]
```

```
        initWithFrame:self.collectionView.frame
collectionViewLayout:layout];
self.currentPinchCollectionView.backgroundColor =
[UIColor clearColor];
self.currentPinchCollectionView.delegate = self;
self.currentPinchCollectionView.dataSource = self;
self.currentPinchCollectionView.autoresizingMask =
UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
[self.currentPinchCollectionView
registerNib:
[UINib nibWithNibName:@"FlickrPhotoCell"
bundle:[NSBundle mainBundle]]
forCellReuseIdentifier:@"MY_CELL"];
[self.currentPinchCollectionView
registerNib:
[UINib nibWithNibName:@"FlickrPhotoHeaderView"
bundle:[NSBundle mainBundle]]
forSupplementaryViewOfKind:UICollectionViewElementKindSectionHeader
withReuseIdentifier:@"FlickrPhotoHeaderView"];

// 6
[self.collectionViewContainer
addSubview:self.currentPinchCollectionView];

// 7
UIPinchGestureRecognizer *recognizer =
[[UIPinchGestureRecognizer alloc]
initWithTarget:self
action:@selector(handlePinchInGesture:)];
[_currentPinchCollectionView
addGestureRecognizer:recognizer];
}

} else if (recognizer.state ==
UIGestureRecognizerStateChanged) {
if (self.currentPinchedItem) {
// 8
CGFloat theScale = recognizer.scale;
theScale = MIN(theScale, kMaxScale);
theScale = MAX(theScale, kMinScale);

// 9
CGFloat theScalePct =
(theScale - kMinScale) / (kMaxScale - kMinScale);
```

```
// 10
PinchLayout *layout =
(PinchLayout*)_currentPinchCollectionView.collectionViewLayout;
layout.pinchScale = theScalePct;
layout.pinchCenter =
[recognizer locationInView:self.collectionView];

// 11
self.collectionView.alpha = 1.0f - theScalePct;
}

} else {
    if (self.currentPinchedItem) {
        // 12
        PinchLayout *layout =
(PinchLayout*)_currentPinchCollectionView.collectionViewLayout;
        layout.pinchScale = 1.0f;
        self.collectionView.alpha = 0.0f;
    }
}
}
```

Phew, that's a long method! Here's the explanation:

1. Gesture recognizers go through a few different states. This method needs to do different things based on if the gesture has just begun, if it has changed (meaning that the pinch has either grown or moved), or if the gesture has ended.

Note: If you aren't familiar with gesture recognizers, check out this tutorial: <http://www.raywenderlich.com/6567/uigesturerecognizer-tutorial-in-ios-5-pinches-pans-and-more>

2. If the gesture has just begun, then get the point at which the pinch occurred. Then use the point to get the cell over which the pinch occurred.
3. If an item was found (remember the pinch could have been over empty space instead of a cell), then the index path is assigned to the property created earlier. You'll use this later as a flag to determine if there is a pinch in progress.
4. You create and set up a new pinch layout, as desired, and initialize the pinch scale to 0.
5. You create a new collection view with the layout set to the pinch layout. You set its frame to be the same as the main collection view, and its autoresizing mask as required for rotation of the UI to work. The background color needs to be clear so that you can see through to the main collection view behind it. You set up the delegate and data source, and register the cell and header view NIBs, as required.

6. If you look in the storyboard for the app, then you'll see in View Controller's interface that there is a view wired up to the `collectionViewContainer` property. The main collection view is a sub-view of this, and to overlay the new collection view, it is added as another sub-view of the container view.
7. Finally, you add a pinch gesture recognizer to the new collection view to handle pinching in, so that the user can get back to the list of search queries.
8. When the pinch moves, expands, or contracts, the recognizer will be in the "changed" state. Here you check the `currentPinchedItem` property to ensure that there is a pinch in progress. If yes, then the scale is determined from the recognizer's `scale` property, and then is limited to the minimum and maximum values you defined earlier.
9. Remember the `pinchScale` property on `PinchLayout` that ranges from 0 to 1 and defines how far out the photos should be spread? You determine the scale percentage by using some standard math.
10. Then you take the layout from the current pinch collection view and apply the scale and center.
11. Finally, you change the alpha of the main collection view so that when the pinch first starts it is 1, and when the pinch is fully expanded it's 0. This will give the illusion of the view fading out as the pinch grows, just like in the Photos app.
12. For other gesture recognizer states (usually if the recognizer finishes or is cancelled), the pinch scale on the pinch layout is set to 1 and the alpha of the main collection view is set to 0. This is just to ensure that the user finishes the pinch with the pinch collection view fully expanded.

As noted above, you've added a second pinch recognizer, this time to the pinch collection view, to handle pinching back in to the main view. Implement the handler by adding the following method to **ViewController.m**:

```
- (void)handlePinchInGesture:  
    (UIPinchGestureRecognizer*)recognizer  
{  
    if (recognizer.state == UIGestureRecognizerStateBegan) {  
        // 1  
        self.collectionView.alpha = 0.0f;  
    }  
    else if (recognizer.state == UIGestureRecognizerStateChanged)  
    {  
        // 2  
        CGFloat theScale = 1.0f / recognizer.scale;  
        theScale = MIN(theScale, kMaxScale);  
        theScale = MAX(theScale, kMinScale);  
  
        CGFloat theScalePct = 1.0f -  
            ((theScale - kMinScale) / (kMaxScale - kMinScale));
```

```
// 3
PinchLayout *layout =
(PinchLayout*)self.currentPinchCollectionView.collectionViewLayout
;
layout.pinchScale = theScalePct;
layout.pinchCenter =
[recognizer locationInView:self.collectionView];

// 4
self.collectionView.alpha = 1.0f - theScalePct;
} else {
// 5
self.collectionView.alpha = 1.0f;

[self.currentPinchCollectionView removeFromSuperview];
self.currentPinchCollectionView = nil;
self.currentPinchedItem = nil;
}
```

This is quite similar to the pinch out recognizer handler:

1. When the recognizer begins, you set the alpha of the main collection view to 0, just to ensure that the initial condition is definitely correct.
 2. As with the pinch out handler, you determine the scale from the recognizer, limit it to the minimum and maximum constants, and then you calculate the percentage. The difference in this case is that the scale the recognizer reports will start at 1 and decrease. So you take the inverse of the value to pass to the pinch layout.
 3. Then you find the pinch layout and set the scale and center properties.
 4. Finally, you set the main collection view's alpha just as in the pinch out handler, such that this time, the view fades in as the pinch gets narrower.
 5. When the recognizer finishes, you set the main collection view's alpha to 1, remove the pinch collection view from the view hierarchy, and reset the state.

You're almost there, but there is still a little bit left to do. You've added a new collection view, and so the data source methods need to be wired up to show some data. Here goes...

First, replace `numberOfSectionsInCollectionView:` with this:

```
- (NSInteger)numberOfSectionsInCollectionView:  
    (UICollectionView*)cv  
{  
    if (cv == self.collectionView) {  
        if (cv.collectionViewLayout == self.layout2) {
```

```

        return 1;
    } else {
        return [self.searches count];
    }
} else if (cv == self.currentPinchCollectionView) {
    // ADDED
    return 1;
}
return 0;
}
}

```

All that's required here is to return 1 in the case of the pinch collection view, since it's just displaying one set of photos.

Next, change `collectionView:numberOfItemsInSection:` to make it look like this:

```

- (NSInteger)collectionView:(UICollectionView*)cv
    numberOfItemsInSection:(NSInteger)section
{
    if (cv == self.collectionView) {
        if (cv.collectionViewLayout == self.layout2) {
            return [self.searches count];
        } else {
            NSString *searchTerm = self.searches[section];
            return [self.searchResults[searchTerm] count];
        }
    } else if (cv == self.currentPinchCollectionView) {
        // ADDED
        NSString *searchTerm =
            self.searches[self.currentPinchedItem.item];
        return [self.searchResults[searchTerm] count];
    }
    return 0;
}
}

```

If it's the pinch collection view asking, then you find the number of items from the currently-pinched item's results.

Update `collectionView:cellForItemAtIndexPath:` to make it look like this:

```

- (UICollectionViewCell*)collectionView:(UICollectionView*)cv
    cellForItemAtIndexPath:(NSIndexPath*)indexPath
{
    FlickrPhotoCell *cell =
        [cv dequeueReusableCellWithIdentifier:@"MY_CELL"
                                forIndexPath:indexPath];
}
}

```

```

FlickrPhoto *photo = nil;
if (cv == self.collectionView) {
    if (cv.collectionViewLayout == self.layout2) {
        NSString *searchTerm =
            self.searches[indexPath.item];
        photo = self.searchResults[searchTerm][0];
    } else {
        NSString *searchTerm =
            self.searches[indexPath.section];
        photo =
            self.searchResults[searchTerm][indexPath.item];
    }
} else if (cv == self.currentPinchCollectionView) {
    // ADDED
    NSString *searchTerm =
        self.searches[self.currentPinchedItem.item];
    photo = self.searchResults[searchTerm][indexPath.item];
}
cell.photo = photo;

return cell;
}

```

The bit added here finds the right photo to set on the cell in the case of the pinch collection view. Once again, you find the search result using the currently-pinched item.

Now modify `collectionView:viewForSupplementaryElementOfKind:atIndexPath:` to:

```

- (UICollectionViewReusableView*)collectionView:
    (UICollectionView*)cv
    viewForSupplementaryElementOfKind:(NSString*)kind
                                forIndexPath:(NSIndexPath*)indexPath
{
    FlickrPhotoHeaderView *headerView =
        [cv dequeueReusableCellWithReuseIdentifier:
            UICollectionElementKindSectionHeader
            forIndexPath:indexPath];
    NSString *searchTerm = nil;
    if (cv == self.collectionView) {
        searchTerm = self.searches[indexPath.section];
    } else if (cv == self.currentPinchCollectionView) {
        // ADDED
    }
}

```

```
    searchTerm =
        self.searches[self.currentPinchedItem.item];

    }
    [headerView setSearchText:searchTerm];
    return headerView;
}
```

You call this method to get the header view (which is a kind of supplementary view) for each section in a collection view. The bit that's changed here adds in the pinch collection view case that selects the right search term for the currently-pinched item.

Next, change `collectionView:didSelectItemAtIndexPath:` to:

```
- (void)collectionView:(UICollectionView*)cv
didSelectItemAtIndexPath:(NSIndexPath*)indexPath
{
    if (self.sharing) {
        NSString *searchTerm =
            self.searches[indexPath.section];
        FlickrPhoto *photo =
            self.searchResults[searchTerm][indexPath.item];
        [self.selectedPhotos addObject:photo];
    } else {
        FlickrPhoto *photo = nil;
        if (cv == self.collectionView) {
            if (cv.collectionViewLayout == self.layout2) {
                NSString *searchTerm =
                    self.searches[indexPath.item];
                photo = self.searchResults[searchTerm][0];
            } else {
                NSString *searchTerm =
                    self.searches[indexPath.section];
                photo =
                    self.searchResults[searchTerm][indexPath.item];
            }
        } else if (cv == self.currentPinchCollectionView) {
            // ADDED
            NSString *searchTerm =
                self.searches[self.currentPinchedItem.item];
            photo =
                self.searchResults[searchTerm][indexPath.item];
        }
        [self performSegueWithIdentifier:@"ShowFlickrPhoto"
            sender:photo];
    }
}
```

```
[self.collectionView  
    deselectItemAtIndexPath:indexPath  
    animated:YES];  
}  
}
```

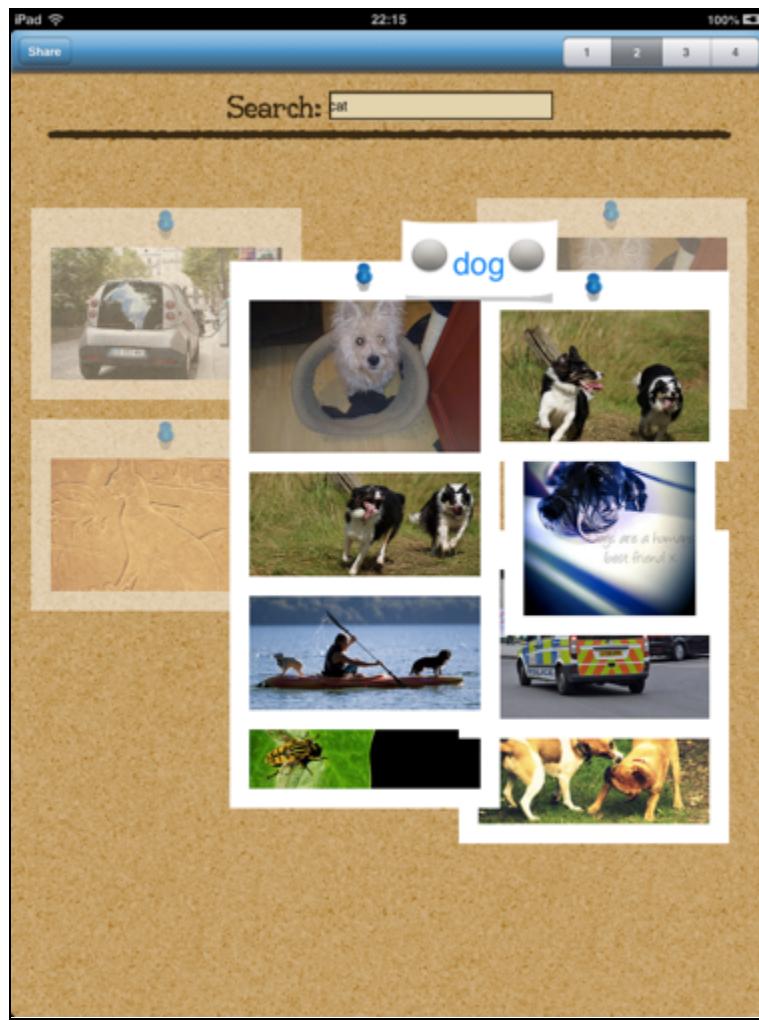
In the case of the pinch collection view, you find the selected photo from the search query for the currently-pinched item, and the index into those results is the item index of the path that was selected.

Finally, change `collectionView:layout:sizeForItemAtIndexPath:` to:

```
- (CGSize)collectionView:(UICollectionView*)cv  
    layout:(UICollectionViewLayout*)cvl  
sizeForItemAtIndexPath:(NSIndexPath*)indexPath  
{  
    FlickrPhoto *photo = nil;  
    if (cv == self.collectionView) {  
        if (cvl == self.layout2) {  
            NSString *searchTerm =  
                self.searches[indexPath.item];  
            photo = self.searchResults[searchTerm][0];  
        } else {  
            NSString *searchTerm =  
                self.searches[indexPath.section];  
            photo =  
                self.searchResults[searchTerm][indexPath.item];  
        }  
    } else if (cv == self.currentPinchCollectionView) {  
        // ADDED  
        NSString *searchTerm =  
            self.searches[self.currentPinchedItem.item];  
        photo = self.searchResults[searchTerm][indexPath.item];  
    }  
  
    CGSize retval = photo.thumbnail.size.width > 0.0f ?  
        photo.thumbnail.size : CGSizeMake(100.0f, 100.0f);  
    retval.height += 35.0f;  
    retval.width += 35.0f;  
    return retval;  
}
```

Yet again, you find the correct photo in the pinched collection view case, and use it to calculate the size for the cell.

Now build and run the app, switch to layout 2 and have a play! You can now pinch to expand and contract each section, like this:



And that's the pinch layout completed! That was a big one, so give yourself a pat on the back!

Take a moment to look again at the code in the layout class. It's incredibly simple – you didn't have to write much code to make the pinch work. Before collection views, this would have been thousands of lines of code!

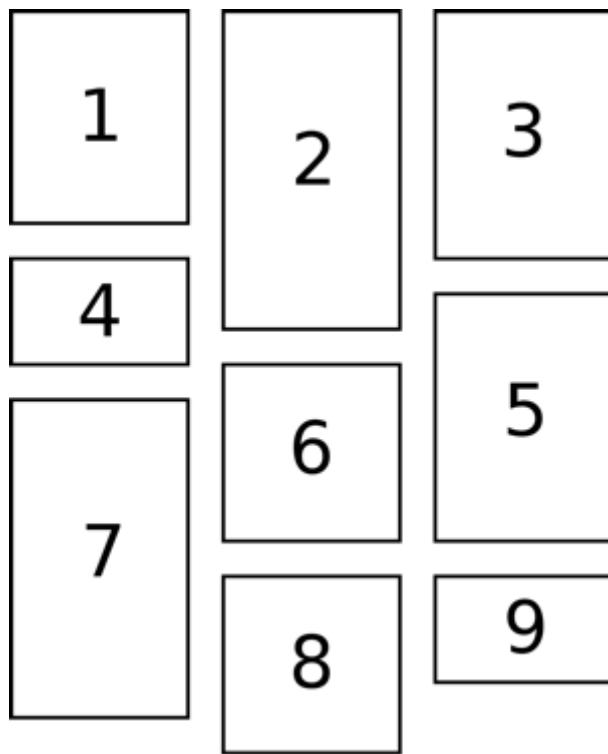
That's your second payoff for finishing this impressive layout: a direct appreciation for how collection views provide a deceptively easy-to-use interface for what, behind the scenes, is a complicated task indeed.

Stacked grid layout

Next you're going to learn all about the stacked grid layout, where the view is split into a certain number of columns, and then cells are arranged such that they are placed as high up on the page as possible. It's the reverse-Tetris of layouts!

For example, consider the nine differently-sized items below. The numbers indicate each item's index, and the diagram shows how the stacked grid layout will order

them. They are first placed as high as they can be in the view, and if two or more columns are the same height, then the left-most column is used.



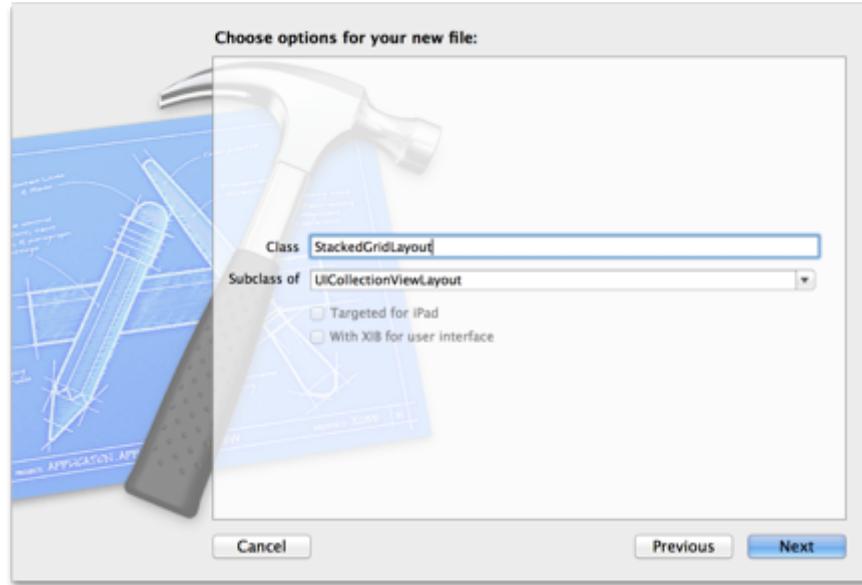
You have very likely seen this kind of layout in certain popular apps, as it has recently become a rather common interface paradigm. It's a great layout for photos, because it allows a list of differently-sized items to grow indefinitely, yet still ends up with naturally balanced columns.

If you were to put item 1 into column 1, item 2 into column 2, item 3 into column 3, and then wrap back round so item 4 went into column 1 and so on, you would find that the columns could easily end up with very different heights if the photos are not regularly resized.

So now that you know what you're about to build, on with the layout!

Your first custom delegate

Add a new file to the **Layout** project group with the **iOS\Cocoa Touch\Objective-C class** template. Name the class **StackedGridLayout**, and make it a subclass of **UICollectionViewLayout**.



Note that you are not deriving from the built-in `UICollectionViewFlowLayout` this time – you're creating a custom layout from scratch!

Open **StackedGridLayout.h**. The first thing you're going to do is create a custom delegate that inherits from `UICollectionViewDelegate`. You need this because there are a few different things your custom layout has to know, such as how many columns there are in a section.

Add the following protocol definition above the `@interface` section:

```
@protocol StackedGridLayoutDelegate <UICollectionViewDelegate>

// 1
- (NSInteger)collectionView:(UICollectionView*)cv
    layout:(UICollectionViewLayout*)cvl
    numberOfColumnsInSection:(NSInteger)section;

// 2
- (CGSize)collectionView:(UICollectionView*)cv
    layout:(UICollectionViewLayout*)cvl
    sizeForItemAtWidth:(CGFloat)width
    atIndexPath:(NSIndexPath *)indexPath;

// 3
- (UIEdgeInsets)collectionView:(UICollectionView*)cv
    layout:(UICollectionViewLayout*)cvl
    itemInsetsForSectionAtIndex:(NSInteger)section;

@end
```

There are similarities with `UICollectionViewDelegateFlowLayout`. You can see this for yourself if you open **UICollectionViewFlowLayout.h** (Cmd-Shift-O, then type the filename) and look at the protocol definition there. Just like the flow layout delegate, your delegate inherits from `UICollectionViewDelegate`, meaning that the methods in that protocol may also be defined.

Here's what the added methods do:

1. Returns the number of columns for a given section.
2. Returns the size for an item, given the width of the column it will go into. For example, in the case of showing photos, the delegate works out how tall the cell needs to be to maintain the aspect ratio of the photo.
3. Returns the insets of a given item. These will be used to inset the cell, such that there are gaps between items in a column and between columns, if so desired.

Now add a property to `stackedGridLayout`:

```
@property (nonatomic, assign) CGFloat headerHeight;
```

This property, `headerHeight`, will eventually be used to define how big the header of each section should be. You could have added this as a delegate method, but I wanted to show you how these kinds of layout parameters can also be implemented using properties.

You can even have a delegate *and* a property – just like `UICollectionViewFlowLayout` has a delegate and property for parameters such as the minimum line spacing. The property defines a default for all sections, whereas the delegate method passes in the section index just in case you want to have a different value for each section.

Data model for storing section and column information

This layout cannot simply be a tweaked flow layout, which is why you created it as a subclass of the abstract base class. This means that you're going to have to handle a lot more implementation details than you did with the pinch layout. But don't fear, as I'm here to walk you through it all!

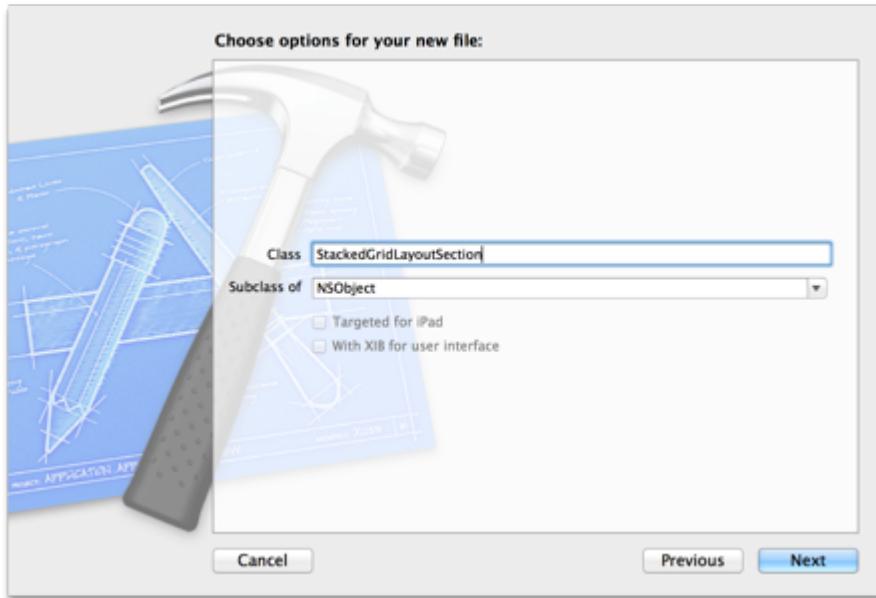
What you need to do is to work out exactly where each cell will be displayed so that you can tell the collection view how big its content is, and eventually, where to display each cell.

The best way to do this is to pre-calculate it all when the layout is prepared. So the first thing you need is a data model to store the layout information. I'm going to show you how to create such a data model using a section object.

The layout will create a section object for every section that exists. As the layout is prepared, each section object will be given items as their sizes are determined via the delegate. It will put each item into whichever column is currently the shortest.

The section will also be responsible for working out the frame of each item. The layout can then ask for the frame when it needs to display the cells. Remember that the frame is the location of the view within the coordinate system of the collection view.

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template. Name the class **StackedGridLayoutSection**, and make it a subclass of **NSObject**.



Open **StackedGridLayoutSection.h** and replace it with:

```
@interface StackedGridLayoutSection : NSObject

@property (nonatomic, assign, readonly) CGRect frame;
@property (nonatomic, assign, readonly) UIEdgeInsets itemInsets;
@property (nonatomic, assign, readonly) CGFloat columnWidth;
@property (nonatomic, assign, readonly) NSInteger numberOfRowsInSection;

- (id)initWithOrigin:(CGPoint)origin
              width:(CGFloat)width
            columns:(NSInteger)columns
      itemInsets:(UIEdgeInsets)itemInsets;

- (void)addItemOfSize:(CGSize)size
   forIndex:(NSInteger)index;

- (CGRect)frameForItemAtIndex:(NSInteger)index;

@end
```

This class will hold information about each item in a section and report back each item's frame.

Now open **StackedGridLayoutSection.m** and create a class extension at the top of the file, below the `#import` line:

```
@interface StackedGridLayoutSection () {
    CGRect _frame;
    UIEdgeInsets _itemInsets;
    CGFloat _columnWidth;
    NSMutableArray *_columnHeights;
    NSMutableDictionary *_indexToFrameMap;
}
@end
```

This adds some variables that will be used internally to track the state of the section. Notice that there are variables for:

- The current frame of the entire section.
- The insets for each item.
- The width of each column.
- An array to contain the current height of each column.
- A dictionary to map an item index to the frame of that item.

There'll be more on all of these shortly, but for now, implement the first method – the initializer – like so:

```
- (id)initWithOrigin:(CGPoint)origin
              width:(CGFloat)width
             columns:(NSInteger)columns
        itemInsets:(UIEdgeInsets)itemInsets
{
    if ((self = [super init])) {
        _frame = CGRectMake(origin.x, origin.y, width, 0.0f);
        _itemInsets = itemInsets;
        _columnWidth = floorf(width / columns);
        _columnHeights = [NSMutableArray new];
        _indexToFrameMap = [NSMutableDictionary new];

        for (NSInteger i = 0; i < columns; i++) {
            [_columnHeights addObject:@(0.0f)];
        }
    }
    return self;
}
```

This is fairly straightforward – you initialize the class with the origin point of the section in the coordinate space of the collection view. You also pass in the width of the entire section, the number of columns and the item insets. The method then creates the various internal states and initializes the column heights array to 0 for the number of columns passed in.

Now, recall the header file you created for this class and the read-only properties. The layout is going to use these as it goes about its business. The methods are easy to implement, as they are just normal getter methods accessing the internal instance variables you defined in the class extension. So, add the following methods:

```
- (CGRect)frame {
    return _frame;
}

- (CGFloat)columnWidth {
    return _columnWidth;
}

- (NSInteger)numberOfItems {
    return _indexToFrameMap.count;
}
```

Next comes the bulk of the work the section has to do. This is the `addItemOfSize:forIndex:` method. The layout will call this method once for each item after it has determined the size of the item by asking its delegate. Add the following code:

```
- (void)addItemOfSize:(CGSize)size
                 forIndex:(NSInteger)index
{
    // 1
    __block CGFloat shortestColumnHeight = CGFLOAT_MAX;
    __block NSUInteger shortestColumnIndex = 0;

    // 2
    [_columnHeights enumerateObjectsUsingBlock:
        ^(NSNumber *height, NSUInteger idx, BOOL *stop)
    {
        CGFloat thisColumnHeight = [height floatValue];
        if (thisColumnHeight < shortestColumnHeight) {
            shortestColumnHeight = thisColumnHeight;
            shortestColumnIndex = idx;
        }
    }];
}
```

```
// 3
CGRect frame;
frame.origin.x = _frame.origin.x +
    (_columnWidth * shortestColumnIndex) +
    _itemInsets.left;
frame.origin.y = _frame.origin.y +
    shortestColumnHeight +
    _itemInsets.top;
frame.size = size;

// 4
_indexToFrameMap[@(index)] =
    [NSValue valueWithCGRect:frame];

// 5
if (CGRectGetMaxY(frame) > CGRectGetMaxY(_frame)) {
    _frame.size.height =
        (CGRectGetMaxY(frame) - _frame.origin.y) +
        _itemInsets.bottom;
}

// 6
[_columnHeights
    replaceObjectAtIndex:shortestColumnIndex
    withObject:@(shortestColumnHeight +
    size.height +
    _itemInsets.bottom)];
}
```

And here is what that does:

1. First of all, you need to calculate the column in which to add the item. You do this by enumerating the array of column heights and working out which is the shortest. I've implemented this with a block-based enumeration method, so a couple of variables are set up as `_block` so that they can be edited from within a block. These variables will eventually hold the shortest column height and its index.
2. You use a standard algorithm to calculate the height of the shortest column and its index.
3. You calculate the frame of the item using the origin of the section (i.e., the `_frame` instance variable), the column size, the cell insets and the item size.
4. You update the `indexToFrameMap` by creating an `NSValue` object to contain the frame. This will be read later when the layout wants to retrieve the frame of a given item.

5. If the y-value of this item's frame is greater than the section frame's y-value, then you update the section frame. This keeps the section frame instance variable up-to-date.
6. Finally, you update the column heights array to reflect the fact that the selected column will now be taller, given that there's a new item in it.

That really is the bulk of this object done! The remaining method is `frameForItemAtIndexPath:`, which returns the frame of a given item. The layout will make much use of this method, so implement it by adding the following code:

```
- (CGRect)frameForItemAtIndexPath:(NSInteger)index {
    return [_indexToFrameMap[@(index)] CGRectValue];
}
```

Back to the layout

Now that you've created the data model for storing the layout information, you can turn back to the layout itself. First, open **StackedGridLayout.m** and add the following import:

```
#import "StackedGridLayoutSection.h"
```

Next you need to define some instance variables that will be used to maintain state in the layout. Add the following class extension:

```
@interface StackedGridLayout () {
    __unsafe_unretained
    id <StackedGridLayoutDelegate> _myDelegate;
    NSMutableArray *_sectionData;
    CGFloat _height;
}
@end
```

Remember the protocol you created for the delegate? This is where you first use it. The `delegate` property of a collection view layout is just a standard `UICollectionViewDelegate`. But in the case of this layout, you're defining that it must be a `StackedGridLayoutDelegate` (remember, it inherits from `UICollectionViewDelegate`). So the `_myDelegate` variable will be used as a cache of the `delegate` property, but cast to an `id <StackedGridLayoutDelegate>` so that the compiler doesn't complain when you try to call methods defined in the `StackedGridLayoutDelegate` protocol.

The other instance variables are an array to hold the sections and a float for the height of the contents. There'll be more on both of those shortly.

Next up is the first of the layout methods, `prepareLayout`, which is the one that gets called by the collection view at the start of a layout pass. Add the following code:

```
- (void)prepareLayout {
    // 1
    [super prepareLayout];

    // 2
    _myDelegate = (id <StackedGridLayoutDelegate>)
        self.collectionView.delegate;
    _sectionData = [NSMutableArray new];
    _height = 0.0f;

    // 3
    CGPoint currentOrigin = CGPointMakeZero;
    NSInteger numberOfRowsInSection =
        self.collectionView.numberOfSections;

    // 4
    for (NSInteger i = 0; i < numberOfRowsSections; i++) {
        // 5
        _height += self.headerHeight;
        currentOrigin.y = _height;

        // 6
        NSInteger numberOfColumns =
            [_myDelegate collectionView:self.collectionView
                layout:self
                numberOfColumnsInSection:i];

        NSInteger numberOfRowsItems =
            [self.collectionView numberOfItemsInSection:i];

        UIEdgeInsets itemInsets =
            [_myDelegate collectionView:self.collectionView
                layout:self
                itemInsetsForSectionAtIndex:i];

        // 7
        StackedGridLayoutSection *section =
            [[StackedGridLayoutSection alloc]
                initWithOrigin:currentOrigin
                width:self.collectionView.bounds.size.width
                columns:numberOfColumns
                itemInsets:itemInsets];
    }

    // 8
```

```
for (NSInteger j = 0; j < numberOfRowsInSection; j++) {
    // 9
    CGFloat itemWidth = (section.columnWidth -
                          section.itemInsets.left -
                          section.itemInsets.right);
    NSIndexPath *itemIndexPath =
        [NSIndexPath indexPathForItem:j inSection:i];
    CGSize itemSize =
        [_myDelegate collectionView:self.collectionView
                           layout:self
                           sizeForItemAtWidth:itemWidth
                           atIndexPath:itemIndexPath];

    // 10
    [section addItemOfSize:itemSize forIndex:j];
}

// 11
[_sectionData addObject:section];

// 12
_height += section.frame.size.height;
currentOrigin.y = _height;
}
}
```

Here's what this method does:

1. First you must call the implementation of the super class, as it may want to do some setup itself.
2. You set up the internal state, first by caching the delegate to a variable of the required type, as explained earlier. Then you created the section data array and initialize the height.
3. You create a couple of variables that will be used frequently. The first is the current origin that will be used to track where in the collection view you currently are as the sections are being created. The second is the number of sections, which is determined by asking the collection view.

You may have thought that the best way to ask for the number of sections would be to ask the collection view's data source, but that would be wrong. Apple says to ask the collection view, since this assures that the layout and collection view both have the same value.

4. Loop through the number of sections.
5. The first thing to do for each section is to increase the layout's height by the section header height. This ensures that there's enough room left for the header

before the section starts. You also update the current origin to reflect this new height.

6. You determine the number of columns, items, and item insets for this section.
7. You create a new section object, passing in the various bits of required information. At this stage, you might want to glance back at the section object implementation to remind yourself why you're passing in each of these values.
8. Loop through the items for the section.
9. Calculate the item width by insetting the column width by the right amount, and then ask the delegate for the item size for the item at the required width.
10. Once the item size is known, you tell the section to add the item. Again, you may want to take a moment to remind yourself of the section's implementation.
11. Now that the section is fully populated with all of its items' data, you add it to the internal array.
12. Finally, you update the height and current origin values. Once this loop is completed for all sections, the height variable will equal the height of the collection view's content.

That's a large part of the layout done! The next method is easy. It's the one that tells the collection view the size of the content. Remember that this is used just like the content size of a scroll view – to indicate the size of the scrollable region and to set up the scroll bars properly. Add the following method:

```
- (CGSize)collectionViewContentSize {
    return CGSizeMake(self.collectionView.bounds.size.width,
                      _height);
}
```

This is straightforward. The width is the size of the collection view itself and the height is the one calculated in `prepareLayout`.

The next couple of methods return layout attributes. Just as you did in the pinch layout, you need to return attributes for every cell, supplementary view, and decoration view included in the layout.

First, implement `layoutAttributesForItemAtIndexPath:` by adding the following code:

```
- (UICollectionViewLayoutAttributes*) layoutAttributesForItemAtIndexPath:(NSIndexPath *)indexPath
{
    StackedGridLayoutSection *section =
        _sectionData[indexPath.section];
```

```
UICollectionViewLayoutAttributes *attributes =
    [UICollectionViewLayoutAttributes
        layoutAttributesForCellWithIndexPath:indexPath];
attributes.frame =
    [section frameForItemAtIndexPath:indexPath.item];

return attributes;
}
```

This method finds the section object for the item and then asks the section for the frame of the item. It creates attributes as appropriate, and returns them.

The next method is similar to the above, but returns layout attributes for supplementary views. You need this because this layout will support header views. So add the following method:

```
- (UICollectionViewLayoutAttributes *)
layoutAttributesForSupplementaryViewOfKind:(NSString *)kind
atIndexPath:(NSIndexPath *)indexPath
{
    StackedGridLayoutSection *section =
        _sectionData[indexPath.section];

    UICollectionViewLayoutAttributes *attributes =
        [UICollectionViewLayoutAttributes
            layoutAttributesForSupplementaryViewOfKind:kind
            indexPath:indexPath];

    CGRect sectionFrame = section.frame;
    CGRect headerFrame =
        CGRectMake(0.0f,
                  sectionFrame.origin.y - self.headerHeight,
                  sectionFrame.size.width,
                  self.headerHeight);
    attributes.frame = headerFrame;

    return attributes;
}
```

Just like the previous method, this one finds the right section and creates a layout attributes object to describe the section header. It calculates the frame by looking at the frame of the section and moving up by the height of the header.

The last method is the all-important `layoutAttributesForElementsInRect:` method:

```
- (NSArray*)layoutAttributesForElementsInRect:(CGRect)rect {
    // 1

    NSMutableArray *attributes = [NSMutableArray new];

    // 2
    [_sectionData enumerateObjectsUsingBlock:
     ^(StackedGridLayoutSection *section,
       NSUInteger sectionIndex,
       BOOL *stop)
    {
        // 3
        CGRect sectionFrame = section.frame;
        CGRect headerFrame =
            CGRectMake(0.0f,
                      sectionFrame.origin.y - self.headerHeight,
                      sectionFrame.size.width,
                      self.headerHeight);

        // 4
        if (CGRectIntersectsRect(headerFrame, rect)) {
            NSIndexPath *indexPath =
                [NSIndexPath indexPathForItem:0
                                      inSection:sectionIndex];
            UICollectionViewLayoutAttributes *la =
                [self layoutAttributesForSupplementaryViewOfKind:
                 UICollectionElementKindSectionHeader
                 atIndexPath:indexPath];
            [attributes addObject:la];
        }

        // 5
        if (CGRectIntersectsRect(sectionFrame, rect)) {
            // 6
            for (NSInteger index = 0;
                  index < section.numberOfItems;
                  index++)
            {
                // 7
                CGRect frame =
                    [section frameForItemAtIndexPath:index];

                // 8
                if (CGRectIntersectsRect(frame, rect)) {
```

```
        NSIndexPath *indexPath =
            [NSIndexPath indexPathForItem:index
                inSection:sectionIndex];

        UICollectionViewLayoutAttributes *la =
        [self layoutAttributesForItemAtIndexPath:indexPath];
        [attributes addObject:la];
    }
}
}];

// 9
return attributes;
}
```

This method is meant to return all the attributes for all elements (cells, supplementary views, and decoration views) within a given rectangle. Here's how it works:

1. It creates an array to hold all the attributes.
2. It enumerates each section item in turn.
3. It calculates the section frame and header frame and stores them in local variables for use later.
4. If the header frame intersects with the rectangle, then the header is visible and you need to return a layout attribute for it. So here you create one and add it to the attributes array.
5. If the section's frame intersects with the rectangle, then at least one of its items must be visible.
6. It loops through the items for the section.
7. For each item, the method obtains its frame by asking the section.
8. If the item's frame intersects with the rectangle, then the method creates layout attributes for it and adds them to the attributes array.
9. Finally, once the method has collected all the attributes of visible elements within the rectangle, it returns the array.

That wasn't so bad, was it?



Adding stacked grid layout

The final piece of the puzzle is using this new layout from the main view controller. So open **ViewController.m** and add the following import:

```
#import "StackedGridLayout.h"
```

Then add a new property to the class extension to hold an instance of the new layout:

```
@property (nonatomic, strong) StackedGridLayout *layout3;
```

Now add the following to the end of `viewDidLoad` to set up the layout:

```
self.layout3 = [[StackedGridLayout alloc] init];
self.layout3.headerHeight = 90.0f;
```

There's not much setup required, you just need to set the header height.

Now modify `case 2` in `layoutSelectionTapped:` to look like this:

```
case 2: {
    self.collectionView.collectionViewLayout = self.layout3;
    [self.collectionView
        removeGestureRecognizer:self.pinchOutGestureRecognizer];
    [self.collectionView
        removeGestureRecognizer:self.longPressGestureRecognizer];
}
break;
```

Now the stacked grid layout is set as the current layout when the user taps the third button in the layout selection segmented control. The gesture recognizers you added in the previous section need to be removed, as they would interfere with the layout if they were still active when this layout is in action.

The final task to do in the main view controller is to implement the various bits of the `StackedGridLayoutDelegate` protocol. First, add this method:

```
- (NSInteger)collectionView:(UICollectionView*)cv
    layout:(UICollectionViewLayout*)cvl
    numberOfColumnsInSection:(NSInteger)section
{
    return 3;
}
```

Recall that this is used to determine the number of columns in a section. I've chosen to return three, but feel free to play around with this number. Maybe you want to see what happens if you have two columns in odd sections and three columns in even sections, for instance!

Next, add the following method:

```
- (UIEdgeInsets)collectionView:(UICollectionView*)cv
    layout:(UICollectionViewLayout*)cvl
    itemInsetsForSectionAtIndex:(NSInteger)section
{
    return UIEdgeInsetsMake(10.0f, 10.0f, 10.0f, 10.0f);
}
```

This returns the item insets to use. I've gone with 10 pixels around each side, but other values may also look nice. Feel free to experiment. ☺

Finally, add the method to tell the layout the size of each item:

```
- (CGSize)collectionView:(UICollectionView*)cv
    layout:(UICollectionViewLayout*)cvl
    sizeForItemAtWidth:(CGFloat)width
    forIndexPath:(NSIndexPath *)indexPath
{
    NSString *searchTerm = self.searches[indexPath.section];
    FlickrPhoto *photo =
        self.searchResults[searchTerm][indexPath.item];

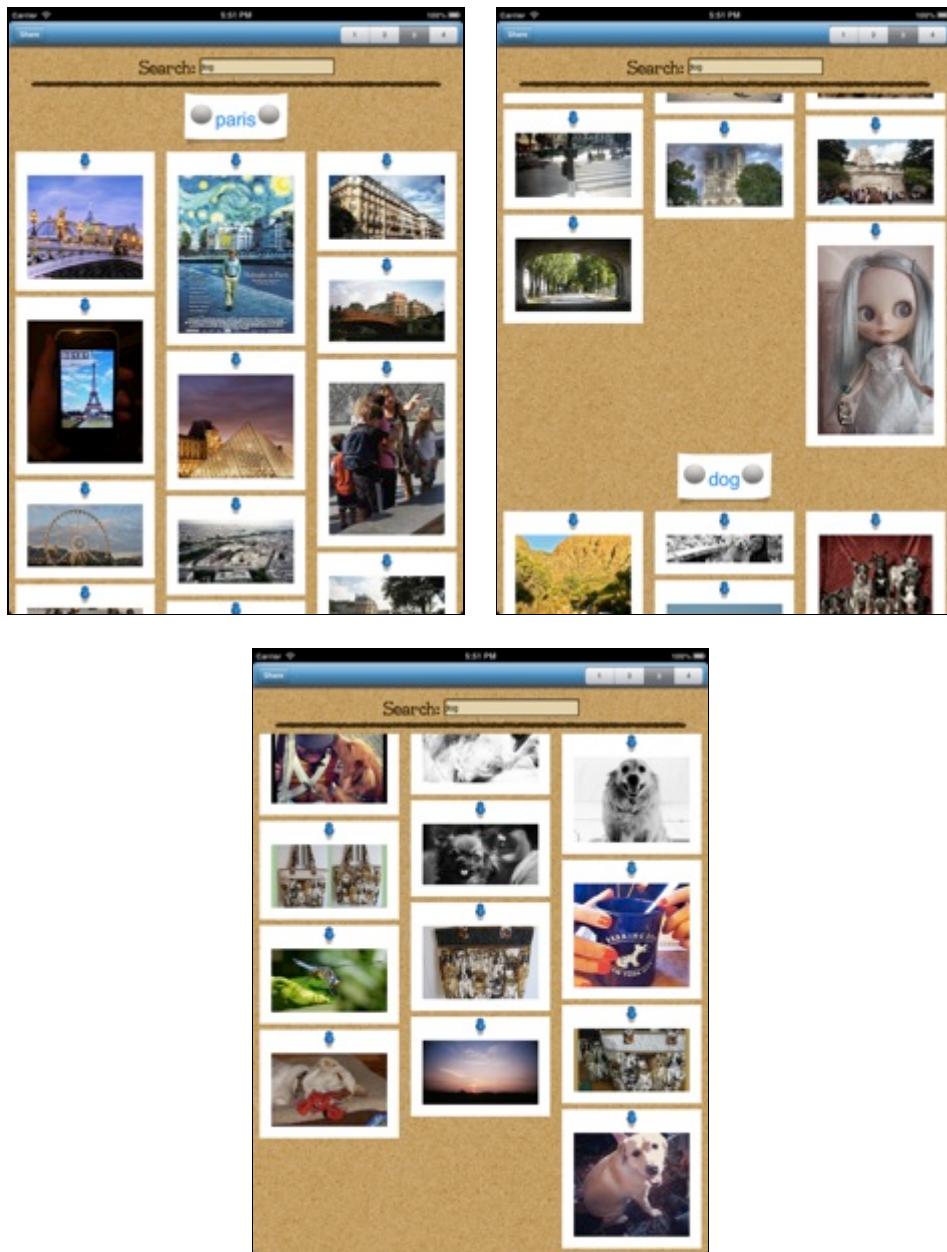
    CGSize picSize = photo.thumbnail.size.width > 0.0f ?
        photo.thumbnail.size : CGSizeMake(100.0f, 100.0f);
    picSize.height += 35.0f;
    picSize.width += 35.0f;

    CGSize retval =
        CGSizeMake(width,
                   picSize.height * width / picSize.width);
    return retval;
}
```

This is very similar to the `collectionView:layout:sizeForItemAtIndexPath:` method from `UICollectionViewDelegateFlowLayout`. The difference is that this method returns a size such that the cell will display the photo with the correct aspect ratio based on the width of the column.

Well done, you've made it!

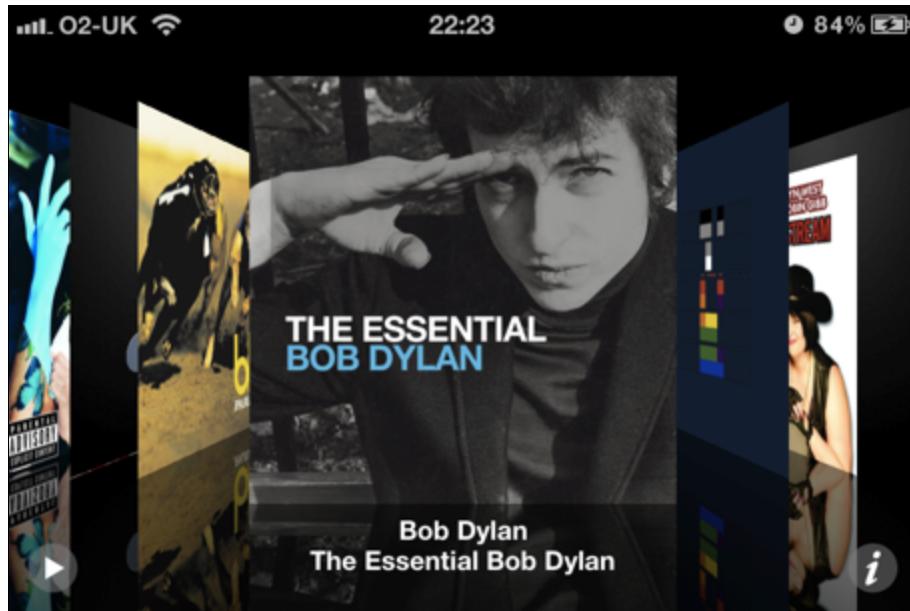
Build and run the app, select layout 3 and do some searches. You should see the stacked layout in action. It'll look something like this:



Not bad, eh? 😊

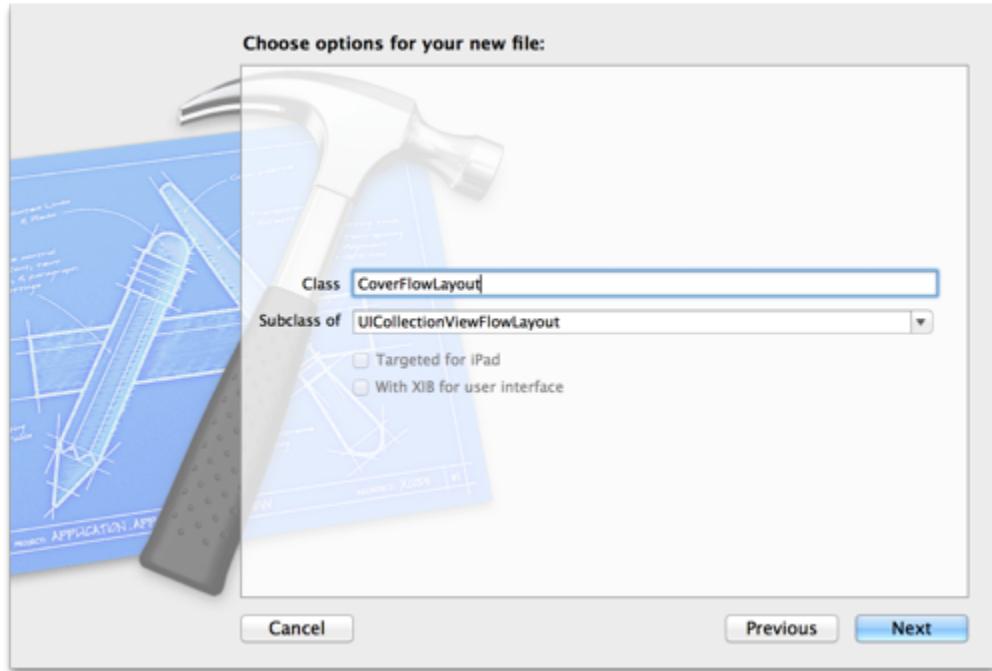
Cover Flow Layout

The final layout I'm going to show you is a "cover flow" style layout. You've likely seen cover flow before, but if not, then here is a screenshot from the Music app on iOS:



Cover flow is a great layout, because it looks really cool and is also quite functional. The user can focus on the item in the center – which also can act as a selection – while getting peeks of elements to either side without them getting in the way by taking up too much of the view.

So let's try this out! Create a new file within the **Layout** group using the **iOS\Cocoa Touch\Objective-C class** template. Name the class **CoverFlowLayout**, and make it a subclass of **UICollectionViewFlowLayout**.



Open **CoverFlowLayout.m** and add the following initializer method:

```
- (id)init {
    if ((self = [super init])) {
        self.scrollDirection =
            UICollectionViewScrollDirectionHorizontal;
        self.minimumLineSpacing = 10000.0f;
    }
    return self;
}
```

The scroll direction needs to be horizontal, and you set the minimum line spacing to a large value such that it forces the layout to have one item per line.

Switch to **ViewController.m** and import the new class by adding the following:

```
#import "CoverFlowLayout.h"
```

Then add a property to hold an instance of this new layout to the class extension:

```
@property (nonatomic, strong) CoverFlowLayout *layout4;
```

And setup this layout by adding the following to the bottom of viewDidLoad:

```
self.layout4 = [[CoverFlowLayout alloc] init];
```

Change the case 3 section for `layoutSelectionTapped:` so that the new layout appears when the fourth button in the layout selection segmented control is tapped:

```
case 3: {
    self.collectionView.collectionViewLayout = self.layout4;
    [self.collectionView
        removeGestureRecognizer:self.pinchOutGestureRecognizer];
    [self.collectionView
        removeGestureRecognizer:self.longPressGestureRecognizer];
}
break;
```

Since the cover flow layout is rather simple, no data source methods need editing, and there is only one flow layout delegate method that does. Modify `collectionView:layout:insetForSectionAtIndex:` to look like this:

```
- (UIEdgeInsets)collectionView:(UICollectionView *)cv
    layout:(UICollectionViewLayout*)cvl
    insetForSectionAtIndex:(NSInteger)section
{
    if (cvl == self.layout4) {
        NSString *searchTerm = self.searches[section];
        NSArray *results = self.searchResults[searchTerm];

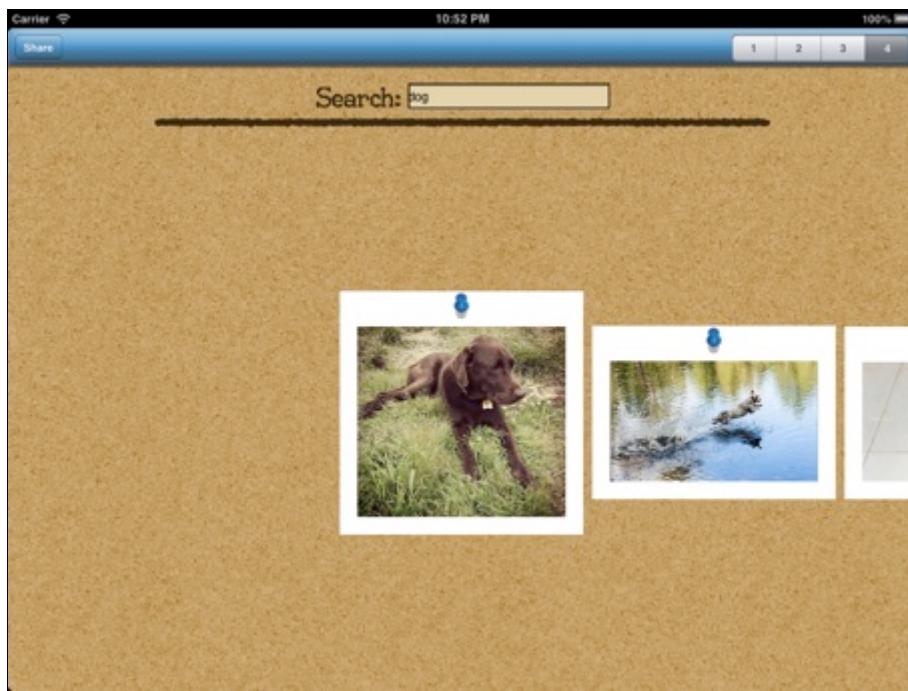
        FlickrPhoto *firstPhoto = results[0];
        CGSize firstItemSize = firstPhoto.thumbnail.size;
        firstItemSize.height += 35.0f;
        firstItemSize.width += 35.0f;

        FlickrPhoto *lastPhoto = results[results.count - 1];
        CGSize lastItemSize = lastPhoto.thumbnail.size;
        lastItemSize.height += 35.0f;
        lastItemSize.width += 35.0f;

        return UIEdgeInsetsMake(
            0.0f,
            (cv.bounds.size.width - firstItemSize.width) / 2.0f,
            0.0f,
            (cv.bounds.size.width - lastItemSize.width) / 2.0f);
    } else {
        return UIEdgeInsetsMake(50.0f, 20.0f, 50.0f, 20.0f);
    }
    return UIEdgeInsetsZero;
}
```

For the cover flow layout, the insets for the section need to be set with enough spacing on the left and the right such that if the view is scrolled all the way to the left or right, then the first or last photo is exactly in the middle of the view. If you didn't do this, then things would look very strange.

Build and run the app, select layout 4 and try a search. You should see something like this:



It's good, but it's no cover flow! There are a few things that need to be implemented in `CoverFlowLayout` to make it into the real thing. ☺

Open **CoverFlowLayout.m** and add a few constants above the `@implementation` line that you'll use later:

```
static const CGFloat kMaxDistancePercentage = 0.3f;
static const CGFloat kMaxRotation =
    (CGFloat)(50.0 * (M_PI / 180.0));
static const CGFloat kMaxZoom = 0.3f;
```

Next, add the following method:

```
- (NSArray*)layoutAttributesForElementsInRect:(CGRect)rect {
    // ...
    CGRect visibleRect =
        (CGRect){.origin = self.collectionView.contentOffset,
                 .size = self.collectionView.bounds.size};
    CGFloat maxDistance =
        visibleRect.size.width * kMaxDistancePercentage;
```

```
// 2
NSArray *array =
    [super layoutAttributesForElementsInRect:rect];
for (UICollectionViewLayoutAttributes *attributes in array)
{
    // 3
    CGFloat distance =
        CGRectGetMidX(visibleRect) - attributes.center.x;

    // 4
    CGFloat normalizedDistance = distance / maxDistance;
    normalizedDistance = MIN(normalizedDistance, 1.0f);
    normalizedDistance = MAX(normalizedDistance, -1.0f);

    // 5
    CGFloat rotation = normalizedDistance * kMaxRotation;
    CGFloat zoom = 1.0f +
        ((1.0f - ABS(normalizedDistance)) * kMaxZoom);

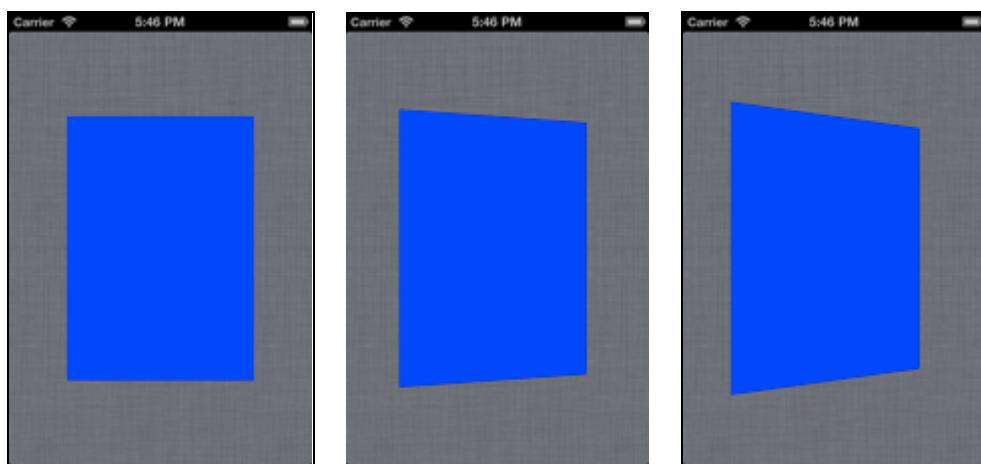
    // 6
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = 1.0 / -1000.0;
    transform = CATransform3DRotate(transform,
                                   rotation,
                                   0.0f,
                                   1.0f,
                                   0.0f);
    transform = CATransform3DScale(transform,
                                   zoom,
                                   zoom,
                                   0.0f);
    attributes.transform3D = transform;
}
// 7
return array;
}
```

As with the previous custom layouts, you need to override this method to provide the layout attributes for the elements in a given rectangle. Here's what the method does:

1. You set up a couple of temporary variables. The first is the visible rectangle of the collection view, calculated using the content offset of the view and the bounds size. The next is the maximum distance away from the center, which defines the distance from the center at which each cell is fully rotated.

2. Then you defer the list of cells in the rectangle to the super-class, since this is a subclass of `UICollectionViewFlowLayout`. The super-class does the hard calculation work.
3. You enumerate the attributes and first find the distance of the cell from the center of the current visible rectangle.
4. You then normalize this distance against the maximum distance to give a percentage of how far the view is along the line from the center to the maximum points in either direction. You then limit this to a value between 1 and -1.
5. You calculate the rotation and zoom.
6. Finally, you create the required transform by first setting `m34` so that when the rotation is done, skew is applied to make it have the appearance of coming out of and going into the screen. Then the transform is rotated and scaled appropriately.
7. You return the attributes.

If you're unsure about the math in this method, then please take some time to review it until you are certain that it is correct. Especially of interest is the `m34` property that refers directly to the transform matrix values. If you're happy to just believe me then all you need to know is that `m34` is the value within the transformation matrix used to give perspective when rotating into and out of the device's screen. Without it, such a rotation would end up just looking like the sides of the view had been squashed in a bit. The screenshots below show a view rotated by 45° about the y-axis. This is the axis running vertically and so such a rotation makes the view appear to go into and out of the screen. The screenshots show different values for `m34` with 0 on the left and then two different negative values on the right. Notice how when `m34` is 0, it doesn't show the effect you'd want at all. Also notice that changing the value kind of makes it look like the view is rotated more, when really all of them are actually rotated by the same amount. Getting this value correct depends on the situation and some trial and error is often the only way to get it spot on.



With this layout, the attributes need to be calculated every time the view is even slightly scrolled. You could ensure the layout gets invalidated by manually calling

`invalidateLayout` on the collection view every time the view scrolls, but fortunately, Apple already thought of this requirement, and added a method for it!

To get the collection view to automatically call `invalidateLayout` when the view scrolls, you simply need to override `shouldInvalidateLayoutForBoundsChange:` and return `YES`. So add the following method to **CoverFlowLayout.m**:

```
- (BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)newBounds
{
    return YES;
}
```

The last method to be added is rather interesting. As every iOS user knows, when you swipe the view and let go, the view continues to scroll and slows down to a stop. Apple added this feature to scroll views with iOS 2.x – and it's a nice one, because it makes you feel like you're interacting with a real-world object.

However, you cannot guarantee that the view will land *exactly* on a cell. This is why the `UICollectionViewLayout` API includes a method named `targetContentOffsetForProposedContentOffset:withScrollingVelocity:`.

This method is provided with the proposed offset of where the view will end up, and you can then decide where exactly you'd like the view to stop and return that value instead. The view will then alter the animation accordingly, such that it slowly lands on the desired location. In the case of the cover flow layout, which is always horizontal, the calculations can be done quite simply.

So, add the following method:

```
- (CGPoint)targetContentOffsetForProposedContentOffset:
    (CGPoint)proposedContentOffset
    withScrollingVelocity:(CGPoint)velocity
{
    // 1
    CGFloat offsetAdjustment = CGFLOAT_MAX;
    CGFloat horizontalCenter = proposedContentOffset.x +
        (CGRectGetWidth(self.collectionView.bounds) / 2.0f);

    // 2
    CGRect targetRect =
        CGRectMake(proposedContentOffset.x,
                  0.0f,
                  self.collectionView.bounds.size.width,
                  self.collectionView.bounds.size.height);
    NSArray *array =
        [super layoutAttributesForElementsInRect:targetRect];
```

```
for (UICollectionViewLayoutAttributes* layoutAttributes
    in array)
{
    // 3
    CGFloat distanceFromCenter =
        layoutAttributes.center.x - horizontalCenter;
    if (ABS(distanceFromCenter) < ABS(offsetAdjustment)) {
        offsetAdjustment = distanceFromCenter;
    }
}

// 4
return CGPointMake(
    proposedContentOffset.x + offsetAdjustment,
    proposedContentOffset.y);
}
```

And here's the explanation:

1. First you set up a couple of temporary variables. The first is an offset adjustment that will be added to the proposed content offset value. You initialize it to a very large value so that the algorithm below will work. The second variable is the horizontal center of the proposed offset, calculated from the proposed offset x-value, plus half the width of the view.
2. You calculate the target rectangle from the proposed offset and view size, and then you retrieve the layout attributes array for that rectangle. These attributes represent the cells that will be visible in the region where the view will end up.
3. Then you iterate through the array and find the cell whose center is nearest to the center of the proposed rectangle. The variable `offsetAdjustment` will then be equal to the distance the proposed rectangle has to be moved to make the view land exactly on that cell.
4. Finally, you change the proposed offset as required, and return it.

And that's it! Build and run the app, switch to layout 4 and do a search. Flick the view around to your heart's content, and there you have it, a nice cover flow layout! It'll look something like this:



Where to go from here?

In this chapter, I've shown you how to create three very different types of custom layouts. But this is just the tip of the iceberg when it comes to what you can achieve.

Hopefully, you have seen just how flexible the API is and you've been inspired with some ideas for custom layouts of your own. Let's hear it for stunning visual effects to display data in crazy new ways!

If you're looking to challenge your new knowledge, here are some ways you can experiment further with this project:

- Use the velocity of the pinch gesture recognizers in the pinch layout to make a tweak such that the user can change their mind about the pinch and go back to where they were before the pinch.
- Add more methods to the stacked grid layout delegate to support different header heights in each section.
- Add supplementary views to the cover flow layout to show a header for each section that stays stationary above the section as you slide through the cells.
- Add decoration view support to one or more of the custom layouts to put a pretty background behind it. If you're feeling brave, then attempt to make the background view move as the collection view is scrolled, giving a 3D effect.

I hope to see you use the power of custom collection view layouts in your own apps! 😊

Chapter 7: Beginning Passbook

By Marin Todorov

The passes implementation in iOS 6 is the hottest technology coming out from Apple right now, and it's groundbreaking for more reasons than one might imagine.

Passes are the result of some awesome fusion. Four separate technologies all combine together to deliver a completely new experience to iPhone users:

1. The new iOS framework *PassKit*
2. The new *Passbook* app bundled with iOS
3. Apple's push notification service (this time delivery is *guaranteed*)
4. Your own server code!

Passes are very different from any other Apple technology you've used before. What makes them unique is that passes are a concept wrapped around a file format. It's up to you to create a pass file, deliver and present it to the user in any way you like. And the choice of technology and programming language to do that is also up to you!

It hardly needs to be said that with this setup, you have much more control over how you implement things than you normally do with Apple technology.

And that is the very reason this chapter will be an epic trip! Since Apple does not handle the mechanisms behind passes, you need to employ a bunch of different technologies to make them work. In this chapter, you'll use Objective-C and iOS; write a PHP web application; use JSON; send multi-part emails with attachments; use OpenSSL to digitally sign files; and more.

The chapter is divided into three parts, with parts two and three building upon what came before:

1. First, you'll learn how to create a pass by hand. You'll start with an empty file and finish with a completely styled and digitally-signed pass.
2. In the second part, you'll develop a small iOS app in order to preview passes in the iPhone Simulator or on your device. You'll also learn about different kinds of passes, thus becoming a Pass Pro.

3. In the third and final part of this chapter, you'll develop a web application in PHP from scratch. This will serve a "real-life" purpose. In this chapter, you have a new employer – Free Hugs LLC! You will make them extremely happy with the amazing new pass-based product you'll develop for them. ☺

The trip through this chapter will be wild, but also very rewarding – by the end of it, you'll have mastered the very latest in Apple technology!

Part 1: Understanding and building Passes

Passes are everything in your pocket

Apple's simplified explanation of passes is that they are "everything in your pocket." I like that line very much, because it demonstrates so well how imaginative you can and should be about creating your pass applications and services. Passes can be anything.

"But what are they, anyway?", you might ask.

Well, here's how a pass looks on the iPhone:



You can easily spot a few different elements on the front of this pass that give you important information: a logo and a company name on the top, the words "Free hug!", which clearly denote what this pass is all about, and also information concerning its validity (start date, duration).

Then there's that barcode at the bottom. It's much like the barcodes you're already used to seeing on train tickets, airplane boarding passes, store cards, etc. If you

think about it, a pass on your iPhone can carry the same information as any of those paper passes you are used to stuffing in your pockets, right? Some text, a barcode – and that's it. Trust me, though: a digital pass on your iPhone can also do a lot more.

What makes a pass, a pass

Look again at what's on the pass below and consider the different sections it has:



All types of passes have these important areas in common:

- 1. Top header.** A top ribbon containing an easy-to-identify logo and a company name. This header area is the part of the pass a user will see when they open a full deck of passes in Passbook. It includes the most essential info you want to provide to the user, so they can easily spot the pass they need.
- 2. Main content area.** A section containing the substance of the pass. Each pass type has different styling for this section, and it usually shows the information that you want most prominently displayed once the pass is open. It's very useful for labels like "20% off," "One free coffee," "Balance: \$120" or anything else of the sort.
- 3. Additional info.** The third section is for additional information – material that's still very important (the validity of the promotion and the start date in the example), but definitely not as important as what the pass is all about.
- 4. Barcode.** The barcode contains encoded information that can be easily transferred to other systems by scanning it with a barcode reader. In the example above, the barcode contains a secret code. When decoded by a scanner, it entitles the bearer of the pass to one free hug (to be delivered immediately).

By now you're probably considering which of the paper passes, store cards, etc. that you use on a daily basis can be converted to a digital pass on your iPhone!

Do I board with this pass, or do I get a free coffee?

Before you dive into coding, have a look at the different types of passes.

Apple has defined four types of passes, each for a common use, as well as a fifth type for "generic" use. Why these different pass types, and how do you recognize them?

The pre-defined pass types are each styled to draw attention to different pieces of information that suit the purpose of the pass. There's a reason why the layout and content of your gym membership card is different from your concert ticket! The first needs your picture on it while the second doesn't, for example.

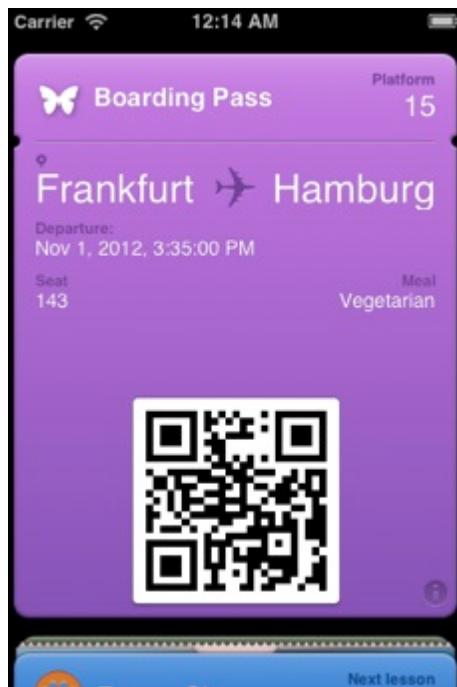
Layout is the main way to differentiate between passes, but UI features like rounded corners and paper cutouts also make passes easily distinguishable. The differences are small, but noticeable, and that's one of the reasons you have to make sure you use the correct type for your pass.

Here are the four pre-defined pass types:

- Coupon
- Boarding pass
- Store card
- Event ticket

Then of course, there is the generic type of pass. This chapter will cover all five types in detail in Part 2.

For now, have a look at the boarding pass example below. You can see it's pretty similar to the hug coupon from earlier, but has some important differences: it features a great two-column layout that makes it very easy to spot the departure and arrival cities; the departure platform number, time and seat number are also well-placed.



By now I am sure you are convinced that passes are a great thing for the iPhone! Now let's turn to how passes are set up, and how to make one!

The guts of the pass

This section will introduce the building blocks for a pass. A pass comes to the user as a file with a .pkpass extension. ("pkpass" standing for PassKit Pass – totally makes sense, right?)

The .pkpass file is just an ordinary .zip file with a custom extension. If you rename one to ZIP and extract it, you'll find several files. Here are the contents of the Free Hug Pass you saw earlier:

- **pass.json** – the description of the pass information fields, their content, and meta information.
- **manifest.json** – the file that describes the list of files inside the pass and the SHA1 checksums of each of those files.
- **signature** – a detached DER signature of manifest.json, generated using an Apple-provided certificate.
- **logo.png** – the logo to show in the pass header. Apple recommends a solid one-color logo. The image size is up to you, but the height shouldn't be more than 50px in order to fit inside the header.
- **logo@2x.png** – the retina counterpart of logo.png.
- **icon.png** – a small icon for the pass, used when the pass comes as an attachment in Mail. As of the time of writing, there is still no documentation about the icon's size, but it looks to be 29x29 pixels (bigger size icons are scaled down, so they look fine as well).

- **icon@2x.png** - the retina icon file.
- **strip.png** and **strip@2x.png** - the image strip, used as background behind the primary fields (only the coupon and store card pass types have this image).

And that's all!

As you can see, the main file of the pass is the JSON file `pass.json`. Inside it you declare all the contents of the front and back of the pass. Along with this JSON file, you provide all images the pass should display (in the case of a coupon, you can supply only `strip.png` and its retina counterpart). Finally, you need a manifest file, which states the original SHA1 checksums of all those files above, and a detached signature signed by you, so that Passbook can verify that the pass hasn't been amended since you created it.

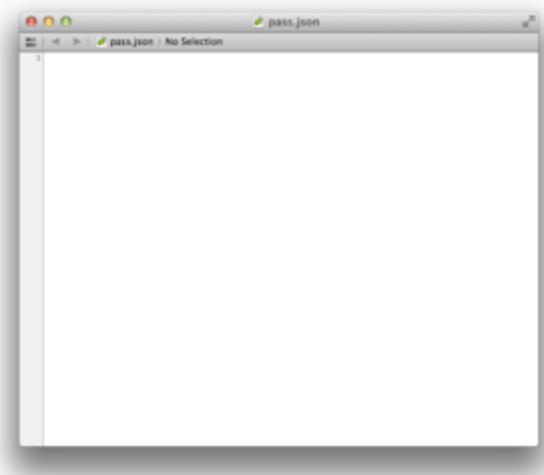
Let's write some JSON!

You're at the point where you can write some code. Let's see if you can reproduce the Free Hug coupon!

But wait! JSON isn't Objective-C. It's not even a programming language of any sort. It's just a markup language used to describe data structures... So, how do you write JSON? Actually, you can use any text editor – Xcode, TextMate, Coda or evenTextEdit. In this chapter, you're going to use Xcode.

From Xcode's menu choose File/New/File... and then choose iOS/Other/Empty for the file type. Name the new file **pass.json** and save it in a folder of your choice. Note you probably want to create a new folder to store the pass in, because you'll be putting a lot of pass-related files in the same folder and it will be nice to keep it all together.

You should now be looking at a tragically empty window, like this one:



But's that's OK – no worries!

For those of you not familiar with JSON notation – it's pretty easy. You can use numbers, strings, arrays, and dictionaries. The information is written out like this:

14.37457 – for numbers

"Some text here!" – for strings

[object1, object2, object3, ...] – for arrays

And:

{"key1": object1, "key2": object2, "key3": object3, ...} – for dictionaries

Object1, object2, object3 and so forth can be any of the four types of objects above – i.e. you can have arrays of dictionaries, dictionaries that hold arrays, strings, numbers, and so on.

You can read more on JSON here: <http://en.wikipedia.org/wiki/JSON>

Note: You might notice that Apple's proprietary **plist** file format stores the same types of primitive data that JSON does – arrays, dictionaries, numbers, and the like. You might wonder why Apple chose to use JSON instead of the plist file format we all know and love.

Well, passes are most likely to be generated in an environment that is not iOS or OS X – most likely, passes will be generated on a web server in response to a request from a user. Therefore Apple wanted to make it easier for you to generate passes in all possible web-scripting languages, and since JSON is well-supported in PHP, .NET, Ruby, Python and so forth, in the end JSON probably seemed like the best choice.

Let's start with the barebones of the Free Hug coupon! Copy this code into your **pass.json** file:

```
{  
    "formatVersion" : 1,  
    "passTypeIdentifier" : "pass.com.yourdomain.couponfreehug",  
    "serialNumber" : "001",  
    "teamIdentifier" : "<YOUR TEAM IDENTIFIER>",  
    "organizationName" : "Free Hugs LLC",  
    "description" : "Coupon for 1 Free Hug"  
}
```

This is the bare minimum of meta-information you need to provide:

- **formatVersion** – the file format version, and since this is a brand-new file format, you're using 1 (note that 1 is a number; if you provide a string for the value, the file won't validate).

- `passTypeIdentifier` – this is the identifier of the Pass. It's pretty similar to the bundle identifier in an iOS app. More will be said about this identifier in a minute.
- `serialNumber` – this is the serial number of the pass. You generate this number any way you like – it can be numeric, like "00193" (note that it still needs to be written out as a string value), or a combination of letters and numbers, like the serial numbers you've seen on airplane boarding passes (for example, "XS83A").
- `teamIdentifier` – this is the unique 10-character identifier Apple assigns to you as an iOS developer. If you've been creating your own iOS apps, you should already be familiar with it. To find your team identifier, log onto the iOS Member Center and click the name of your organization. You will find it there next to a label titled "Company/Organization ID". I'll show you another way to find it later on as well.
- `organizationName` – the name of the issuing entity.
- `description` – a short description of the pass.

This is a lot of information, so let's see how is it useful to Apple.

Since the passes are not necessarily connected to an iOS app, there's no automatic way for Apple to connect a pass to a given iOS developer account (which it needs to do to validate the pass contents). Passes might arrive to the user's device independent from an app, via mail or web download (or another way). This is why the `teamIdentifier` is included in the meta information inside the pass – to connect the pass instance to an iOS developer.

Once Apple knows the identity of the developer who created the pass, they take the `passTypeIdentifier` to figure out which type of pass it is (of all the ones defined by a given developer account). Every pass type has its own certificate, so Apple can use this certificate and validate the signature included in the pass bundle – i.e., make sure nobody tampered with the pass contents.

Finally, the serial number is used to identify the unique instance issued for the given pass type.

To recap, consider an example:

- Joe's Café has an iOS developer account with Apple. Thus, they use their team identifier on all their passes.
- They have store cards with preloaded store credit, which users can use to order yummy coffees in the shop. Store cards have `passTypeIdentifier` "pass.com.joescafe.storeCard".
- They also have discount coupons that have a different `passTypeIdentifier` – "pass.com.joescafe.discountCoupon".
- A given user can own more than one store card (for example, they bought one and a friend gave them one as a present), so the serial number is used to distinguish between two store cards of the same pass type (for example, "0134" and "0274").

- Passes from the same `passTypeIdentifier` will be grouped together inside Passbook. When the user is at Joe's Café, they'll tap the stack of Joe's Café store cards in Passbook, and choose the one they want to use – probably the one that still has credit on it!

By now you should understand pretty well how pass identification works. So "pass" this section and move on to the next, where you'll create your first pass type.

Get me the certificate!

Head to the iOS Developer Portal (<https://developer.apple.com/devcenter/ios/index.action>), and after you log in, open up the iOS Provisioning Portal from the menu on the right-hand side.

If you've already been checking out the new stuff for iOS 6, you might have noticed the new item in the left menu called **Pass Type IDs**. Click on that link and on the next page, you'll see a list of all the pass types you've already created (probably empty at this point).

Click the **New Pass Type ID** button and you'll go to a page where you can create a new type. In the *description* field enter "Free Hug Pass" and in the *Identifier* field enter "pass.com.yourdomain.couponfreehug". For the Pass Type ID, Apple recommends the reversed domain notation with a prefix of "pass." So that's the format you're using.

Note: For the purposes of this chapter you can certainly use "com.yourdomain", but in production applications remember that you should replace com.yourdomain with the actual reverse notation for your domain. ☺

Also note that if you decide to change the identifier here to something other than "pass.com.yourdomain.couponfreehug", be sure to update the `passTypeIdentifier` value in `pass.json` accordingly.

Next, simply click Submit to create your pass.



By the very way that light indicator is not lit up in green, you might already guess there's something missing. Yes – you are right. You still need to generate the pass certificates. Click on Configure to do that. The next page that comes up is a good place to spot the full identifier of your pass.



Here you can see your *teamIdentifier* (it's the first 10 characters beginning with ABC) and update **pass.json** with it – make sure to replace only the <YOUR TEAM IDENTIFIER> placeholder and leave the quotes around it!

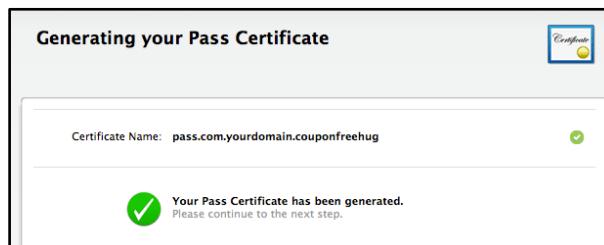
All right! Your pass.json metadata is now updated. However, there's still an extra step to get your certificate imported into your development environment.

Back in the iOS Provisioning Portal, click the Configure button:



This will launch the Pass Certificate Assistant (what a fancy name!), which will guide you through the process of generating your certificate. Pay attention to the dialog and follow the steps.

After you upload your signing request, your certificate will be generated in a timely manner and you should see the success dialogue:



Click Continue one more time and then click Download to get the certificate file. After the file downloads, find and double click it to import it into the Keychain. Note that it's most probably in the Downloads folder inside your user folder, unless you've set another default download location in your browser.

If Keychain asks you to confirm the import, click the Add button:



And then you should see your certificate in Keychain Access:



w00t – you’re done making your certificate! Now you have real metadata in your JSON file and the proper certificate for when the time comes to sign the pass.

Passes come in style this year!

Let’s continue building the pass.json file – next up is styling its looks.

Open up **pass.json** and at the end of the line defining the “description” key, add a comma – you need this because you’re going to be adding more keys to that dictionary. Just before the closing curly bracket, add:

```
"logoText" : "Free Hugs LLC",
"foregroundColor" : "rgb(255, 255, 255)",
"backgroundColor" : "rgb(135, 129, 189)",
"labelColor" : "rgb(45, 54, 129)"
```

Here’s what those keys do:

- `logoText` – this is the text that shows up in the pass header next to your logo image.
- `foregroundColor` – the color of the text.
- `backgroundColor` – the background color of the pass. Passbook will take the background color code and create a nice gradient based on it that gives the front of the pass a very nice visual appearance when presented to the user. Apple recommends using vibrant colors.
- `labelColor` – depending on your background color, the labels (or hints) on the pass will be rendered in a color that is nicely recognizable over the background. If you want, though, you can specify your own color code for those labels.

Now your pass has some style! Heidi Klum would be proud.

You’re going to keep adding elements until you have the bare minimum to proceed to building and testing in Passbook. Next, you’ll add a barcode.

Sounds complicated, right? Luckily, it’s incredibly easy to do this. PassKit supports three different types of barcodes (all of them being 2D barcode formats): QR, PDF417 and Aztec. To illustrate what I mean:



In other words, passes don't support those old-fashioned barcodes you usually see on the packaging of items in the grocery store and (anticipating your question) support for them is not planned. Instead, PassKit uses fancy 2D formats, which are easier for image-based scanning devices (like the iPhone) to read.

You will use the PDF417 standard for the Free Hug coupon (the middle option in the picture above), so add a comma after the last line of the code (again before the closing bracket) and paste in the barcode JSON:

```
"barcode" : {
    "message" : "All you need is love",
    "format" : "PKBarcodeFormatPDF417",
    "messageEncoding" : "iso-8859-1"
}
```

Believe it or not, this is all you need to do – Apple will take care of generating the barcode for you! The `barcode` dictionary has three keys:

- `message` – this is the data you'd like the barcode to contain. It can be a unique code, the bearer's client ID, or anything else. In the case of the Free Hug pass, it's just a text message that authorizes the bearer to a promotional hug.
- `format` – the barcode format name. To generate a 2D barcode, use one of the constants : `PKBarcodeFormatQR`, `PKBarcodeFormatPDF417` or `PKBarcodeFormatAztec`.
- `messageEncoding` – the name of the text encoding used to decode the message. If you use a non-“latin1” language, you will want to use `UTF8` or any other text encoding that suits you (if you use standard English letters and numbers, the default to go with is “`iso-8859-1`”).

So far, the pass source code should look something like this:

```
{  
    "formatVersion" : 1,  
    "passTypeIdentifier" : "pass.com.yourdomain.couponfreehug",  
    "serialNumber" : "001",  
    "teamIdentifier" : "ABC1230000",  
    "organizationName" : "Free Hugs LLC",  
    "description" : "Coupon for 1 Free Hug",  
    "logoText" : "Free Hugs LLC",  
    "foregroundColor" : "rgb(255, 255, 255)",  
    "backgroundColor" : "rgb(135, 129, 189)",  
    "labelColor" : "rgb(45, 54, 129)",  
    "barcode" : {  
        "message" : "All you need is love",  
        "format" : "PKBarcodeFormatPDF417",  
        "messageEncoding" : "iso-8859-1"  
    }  
}
```

Next you'll add some valuable information to the front side of the pass.

Once again, after the closing bracket of the `barcode` dictionary add a comma. (You keep doing this because I'd like for you to stick to having a valid JSON file at all times. Once you're used to writing valid JSON, you can save yourself lots of time while debugging down the road.)

Go ahead and add the key denoting a coupon type of pass (again, paste this just before the final closing bracket):

```
"coupon" : {  
}
```

Inside this empty dictionary you will be adding all the definitions and data to show information on the coupon pass. In between the enclosing brackets of "coupon" add:

```
"primaryFields" : [  
    {  
        "key" : "offer",  
        "label" : "for you",  
        "value" : "Free hug"  
    }  
]
```

This structure looks a bit weird at first glance, but believe me – it all makes sense. First you add a new key called "primaryFields", the value of which is an array. Why an array? Because all sections can display one or more fields, so you need a list to specify them one after another.

Each dictionary in the list describes one field. You've probably already figured it all out, but let me spell it out just in case. For each field you've got the following basic keys:

- **key** – the field name. In this case, the "offer" field is the one that shows up in the main content area of the pass.
- **label** – the accompanying label to the field. In this case, "for you" will show up in small letters below the text.
- **value** – the text the field shows on the pass. In this case, "Free hug!" will show up very large in the majority of the content area.

There are more keys you can use for each field, but you'll have a look at those later on.

The important thing is – you now have a complete pass.json file! Sweet!

Time to make it beautiful, baby!

They say an image is worth a thousand words – lucky for you, there's no space for thousand words on the front side of the pass, so when in need you might as well use an image.

In the resources folder for this chapter you will find a zip file called **Beginning_Passbook_files.zip** – extract it to find a sub-folder called **FreeHugCoupon**. Copy all files in this folder with a .png extension into your own Pass folder. These are the images for the pass's icon, logo and strip image.

And that's it – you're done! Phew, that was easy.

Wait, what? You don't have to add any code inside pass.json to tell Passbook which image files to load. That's right!

Passbook will automatically load image files following standard naming conventions. This means images named icon.png; icon@2x.png; logo.png; logo@2x.png; strip.png and strip@2x.png will be displayed on the pass, if provided. No need to do anything but include these files in the pass bundle. (There's also one more image that this chapter will cover later on.)

Note: There's one nasty gotcha about the pass images, which took me a while to figure out – the image files need to be exported as PNG24 format. For some reason the much smaller-in-size PNG8 format just won't show up in Passbook.

"Let's give that pass a try! I want to see it already!", for sure you are crying in despair.

Unfortunately, until the pass bundle is complete, signed and zipped you cannot preview the pass. Passbook won't display any invalid (incomplete) passes. So muster a little more patience and keep going.

The pass manifest

The pass manifest is another JSON file you need to create, and it describes all the files included in the pass bundle and their SHA1 checksums.

You can make this yourself by generating the SHA1 checksums for each file yourself (as I'll show you in a minute), but so that you can move faster with building your first pass I've included an already-generated `manifest.json` file in the chapter's assets folder. Copy **manifest.json** to your working Pass folder. It's contents are as follows:

```
{  
    "strip.png": "25b4c9ff2bafe056f3e28379db0ef3fb460c718b",  
    "strip@2x.png": "dee775ed6fb3c7278b84c65853401e760caabc92",  
    "icon.png": "8eaa0896db93f2165fa417df3d002ce9c61fc92",  
    "icon@2x.png": "555ce7f70f2f44fb7ac9d9f46df5738ec6250f37",  
    "logo.png": "e8c4edfbcae41d9d88fad7137d8ed30ae5f73e67",  
    "logo@2x.png": "1f9b1cc4c75b380ade07e9f2b7f37f988d9d14c3",  
    "pass.json": "<INSERT YOUR PASS SHA1 HERE>"  
}
```

The SHA1 checksums for the images are already pre-filled, but the final checksum – the one for your `pass.json` file – is not. You are going to generate its SHA1 yourself. It's quite easy – open up Terminal and navigate to your pass folder.

Note: If you aren't familiar with navigating directories in the Terminal, do this: move your pass files into a folder on your Desktop and call it "FreeHugCoupon", then launch Terminal and enter this command:

```
cd ~/Desktop/FreeHugCoupon
```

There you go.

Enter this command at the Terminal prompt:

```
openssl sha1 pass.json
```

The output of the command should look something like this (the actual checksum will be different for you):

```
SHA1(pass.json)= c24766ef5aa92197eace640fcc4fb584a505a733
```

Select the alphanumeric checksum value and in your **manifest.json** file replace "<INSERT YOUR PASS SHA1 HERE>" with the checksum string. Save the file and you're done! (Make sure you leave the quotes alone.)

NB! It's important that you do not edit pass.json anymore until told to do so. If you add even one character to pass.json, the SHA1 checksum will change and your manifest.json will become invalid, since the checksum given there will no longer match the checksum of the modified pass.json file.

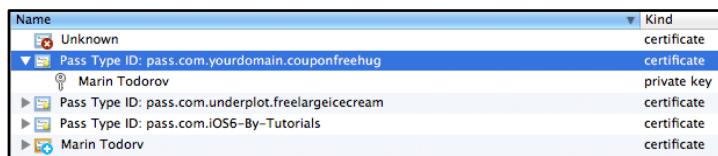
This is the last source file you need for your pass. Awesome!

Can I have your signature, please?

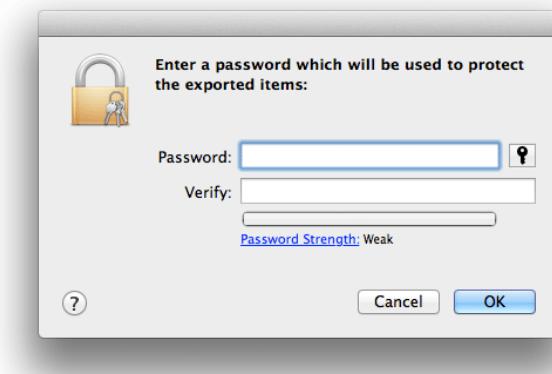
Now comes the most interesting part of creating a pass.

Remember the certificate you got from Apple for your pass type? You've imported it to your Keychain and you haven't touched it since. Now you're going to export the certificate and the key in PEM format, so that you can use them with OpenSSL.

Open up Keychain Access, select Certificates from the left menu (under Categories) and find the certificate called "Pass Type ID: pass.com.yourdomain.couponfreehug". Make sure you've selected the item itself and *not* the private key underneath:



Next right-click on the certificate and choose from the popup menu **Export "Pass Type ID: pass.com.yourdomain.couponfreehug"**. Save the exported file as **Certificates.p12** inside your working pass folder. You'll be presented with a dialog to choose a password:



To make the process a bit easier, just click OK – the certificate will be exported with no password protection.

Note: At this point you might be prompted to enter the password for the login keychain. If you are, then simply providing your user password should suffice.

The Certificates.p12 file now contains both the pass certificate and your private key. OpenSSL needs those in separate files, so now you have to extract them from the .p12 file.

Switch back to Terminal – it's time for OpenSSL magic!

After making sure the current directory is the correct one (to double check, enter "ls -al" and hit Enter – you should see the file listing of the folder, and it should contain your Certificates.p12 file), enter the following command:

```
openssl pkcs12 -in Certificates.p12 -clcerts -nokeys -out  
passcertificate.pem -passin pass:
```

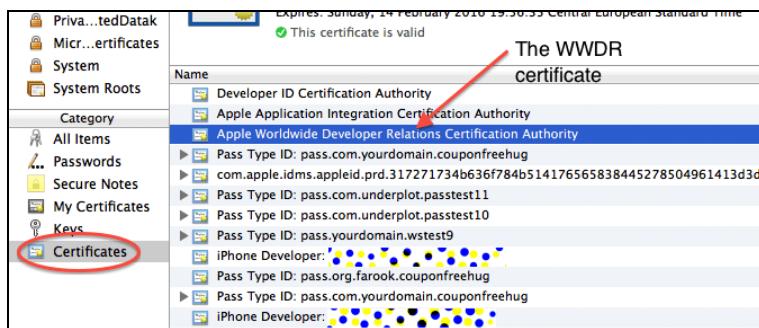
That'll export only the pass certificate in PEM format and save it as "passcertificate.pem" inside the same folder. (OpenSSL will spit out the message "MAC verified OK" if the operation is successful.)

Next, export the key as a separate file with this command:

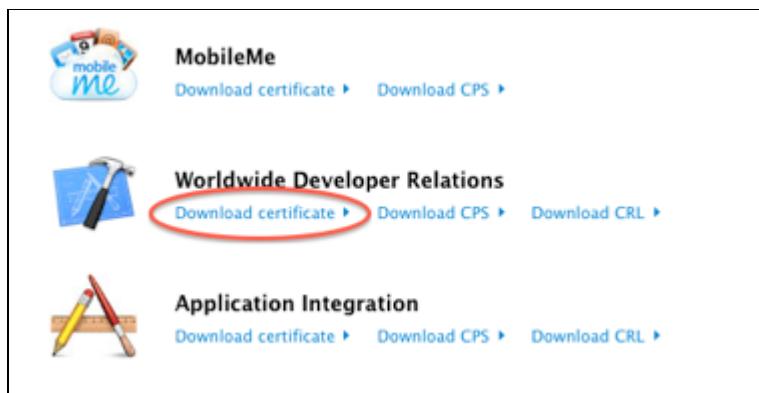
```
openssl pkcs12 -in Certificates.p12 -nocerts -out passkey.pem -  
passin pass: -passout pass:12345
```

Note that this time, you need to provide a password in order to export the key file. In this case, just use "12345" – obviously for production environments it's advisable to use a strong password (doh!) – so, nothing of the "password1" or "passw00t" sort, please.

To sign your pass bundle you will need one more certificate - the WWDR Intermediate certificate, which authorizes the issuer of your own certificate – Apple. Chances are you already have it installed in your Keychain. Open up Keychain Access, select the "Certificates" category and look for a certificate called "Apple Worldwide Developer Relations Certification Authority" (yes, this is a quite long name indeed):



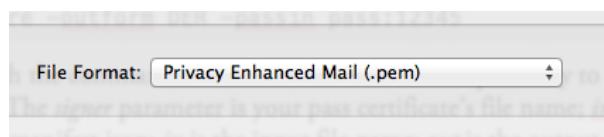
If by any chance you don't have this certificate already imported, then open up in your browser this web page: <http://www.apple.com/certificateauthority/>. Here you can freely download the most important Apple certificates you might need. Scroll down, find the WWDR certificate, download the .cer file and import it in Keychain.



You're ready to go on with exporting the certificate. Back in Keychain Access right click on the certificate name and choose the **Export** option in the popup menu:



In the "Save as..." dialogue find the File format drop box and choose **Privacy Enhanced Mail (.pem)** option:



In the text field at the top of the dialogue give the file the name "**WWDR.pem**", select your working pass directory as destination, and hit the Save button to finish the export.

You're finally ready to create the signature. Enter this command:

```
openssl smime -binary -sign -certfile WWDR.pem -signer  
passcertificate.pem -inkey passkey.pem -in manifest.json -out  
signature -outform DER -passin pass:12345
```

Read through the command line above once more – it's quite easy to understand all the parameters. The *certfile* parameter is the intermediate certificate file, the *signer* parameter is your pass certificate's file name; *inkey* is the key file used to sign *manifest.json*; *in* is the input file name; *out* is the output file name; *outform* is the output format (you need "DER" to create a detached signature); and finally, *passin* is the password for the key file.

Now you have your signature and the production of the pass is almost complete.

The last remaining step is to combine the various files for the pass into a .pkpass file. Enter this command in the Terminal:

```
zip -r freehugcoupon.pkpass manifest.json pass.json signature  
logo.png logo@2x.png icon.png icon@2x.png strip.png strip@2x.png
```

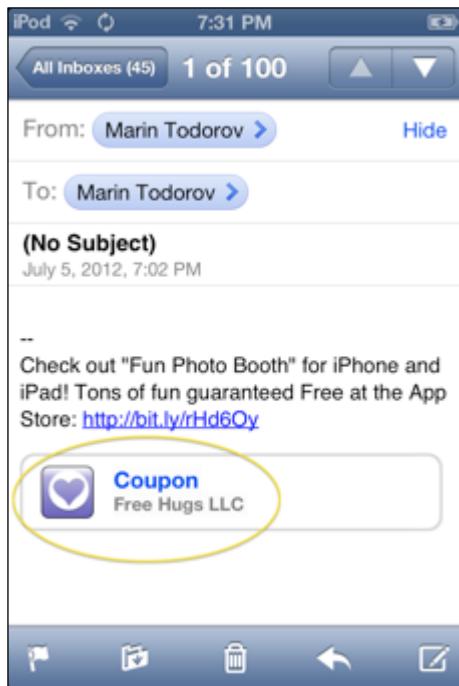
By using the shell command "zip" you create a ZIP file called *freehugcoupon.pkpass* and the archive will contain all the files in the list that follows.

Believe it or not... that's it! High fives all around! You did it!

Show me or I don't believe it!

Yes, you've finally reached the point where you possess a complete and valid pass that you can see on your iOS 6-powered device.

Create an email message to yourself (or to the email account you have set up on your iOS 6 device) and attach the .pkpass file you just created. Send it over, open the mail in Mail.app and voila! You should see the attachment turn into a pass, like so:

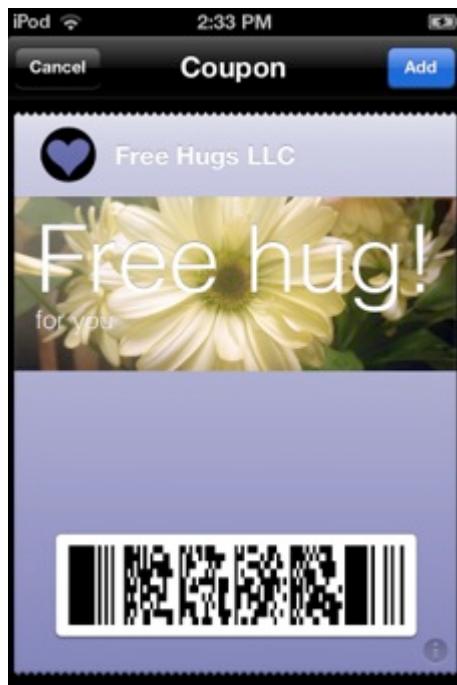


If you see the pass show up – congratulations! You've made it!

If not, don't be disappointed – it's a long and error-prone process, so you'll have to go back to the very beginning and check that you performed all the necessary steps. Make sure your JSON files are valid and that you've exported your certificate and key correctly.

Tip: If you want to check the validity of your JSON files, use an online tool like <http://jsonlint.com/> to quickly proof your code.

Time to see the pass! Tap on it inside the email message and Passbook should pop up showing you some greatness!



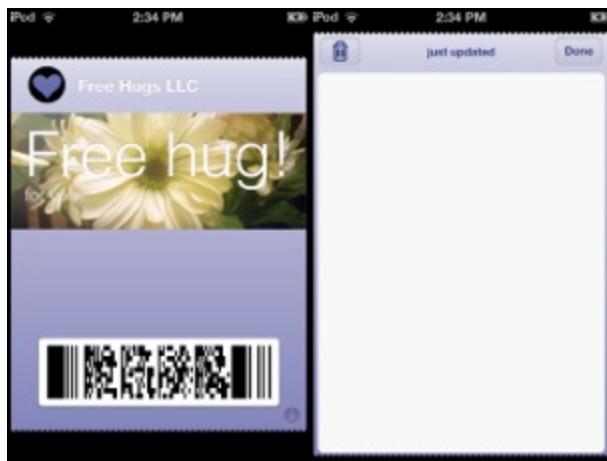
This is pretty awesome! Tap the Add button at the top-right corner and you'll see a neat animation of the pass sliding down (into the deck of passes, but yours is probably empty at this point) and then you will return to Mail.

Note: If you want a few additional passes in your Passbook to see how it looks when it's less empty, try visiting <http://passk.it/samples> using your mobile browser. The site has samples for a bunch of passes that should prove to be interesting. ☺

Now exit Mail and find and launch the Passbook app. Aha! It's your first very own pass. Congratulations!

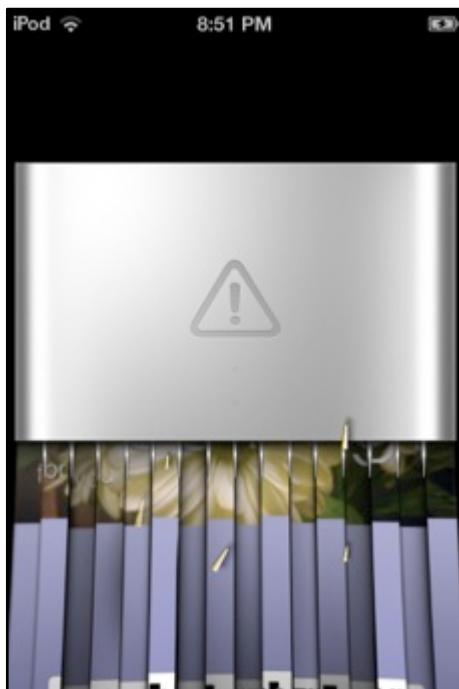
If on the other hand you don't find your pass showing up in Passbook it means it was invalid for some reason and was not imported. Try reviewing your source code; if you get really stuck you can have a look at the pass debugging section in the next chapter "Intermediate Passbook".

Preview the pass in Passbook, front and back (in case you're wondering, you can access the back of the pass by tapping the small italic "i" on the bottom right-hand corner of the pass):



You must admit, the pass does look beautiful, but it's not very informative. Yes – you're going to add more fields to it in a moment.

I bet you are itching right now to try shredding the pass – go ahead! After all, you can now create as many as you want, so – shred some. ☺ Tap the trash bin button and...



Additional pass information

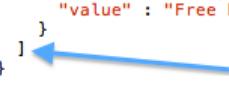
The pass has two sides: on the front you can add text information for several predefined areas; on the back you can show as many fields as you want. Let's first do some front-end maintenance. ☺

Open up **pass.json** and find the `primaryFields` array declaration. On the same level, inside the `coupon` dictionary, you are going to add more sections to show more fields.

As you will be adding another key to the `coupon` dictionary, you need to first add a comma, and then define the new key. Add that comma after the closing square bracket at the bottom of the file (have a look at the picture below for the exact location).

```

12 "barcode" : {
13   "message" : "All you need is love",
14   "format" : "PKBarcodeFormatPDF417",
15   "messageEncoding" : "iso-8859-1"
16 },
17 "coupon" : {
18   "primaryFields" : [
19     {
20       "key" : "offer",
21       "label" : "for you",
22       "value" : "Free hug!"
23     }
24   ]
25 }
26 }
```



On the next line after the comma, paste in this JSON code:

```

"secondaryFields" : [
  {
    "key" : "expires",
    "label" : "Valid for",
    "value" : "Lifetime"
  }
],
"auxiliaryFields" : [
  {
    "key" : "date",
    "label" : "Promo starts on",
    "dateStyle" : "PKDateStyleMedium",
    "timeStyle" : "PKDateStyleNone",
    "value" : "2013-01-11T00:00Z"
  }
]
```

Now in addition to the `primaryFields` dictionary, you have `secondaryFields` and `auxiliaryFields` – these are the extra fields on the front side of the pass. They are not as visually prominent as the primary fields, but are still important enough to be on the front.

The auxiliary fields section should be of particular interest to you – it has some new properties that you haven't used before. This field, unlike the rest of the ones you have, is a date field. Passbook takes care to convert the date/time values to the local time of the user, and also to format and localize the final text.

Have a look at the date field properties:

- **value** – it's a W3C datetime formatted value (for boring documentation on it, check <http://www.w3.org/TR/NOTE-datetime>).
- **dateStyle** – the style to format the date part; must be one of: `PKDateStyleNone`, `PKDateStyleShort`, `PKDateStyleMedium`, `PKDateStyleLong` or `PKDateStyleFull`.
- **timeStyle** – the same as the date style field; use the same constant names to define the time style.

That's done! It's time for you to do another build and see how the pass looks this time! You're having fun already, right?

The build steps to perform:

1. Make sure the JSON code looks OK (if not, be sure to use <http://jsonlint.com/> to check).
2. In Terminal, navigate to your working folder and get the SHA1 checksum of `pass.json` by running:

```
openssl sha1 pass.json
```

3. Place the `pass.json` SHA1 checksum in the **manifest.json** file.
4. Generate the detached signature for the pass by running:

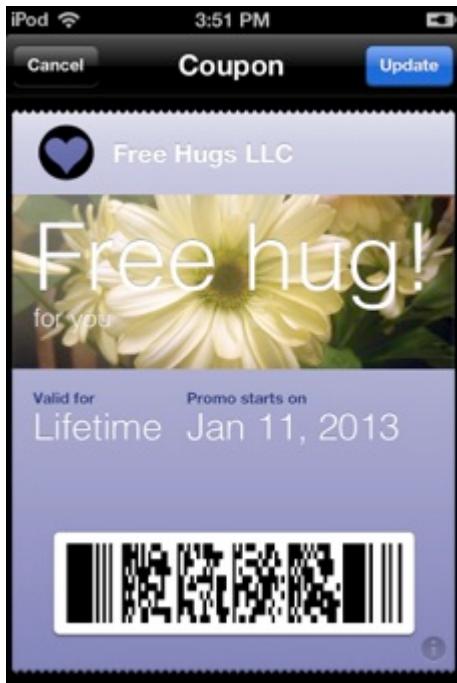
```
openssl smime -binary -sign -certfile WWDR.pem -signer
passcertificate.pem -inkey passkey.pem -in manifest.json -out
signature -outform DER -passin pass:12345
```

5. And finally, zip up the files into a `.pkpass` package:

```
zip -r freehugcoupon.pkpass manifest.json pass.json signature
logo.png logo@2x.png icon.png icon@2x.png strip.png strip@2x.png
```

6. Send the file via email to your iOS 6 powered device and have a look!

Mission accomplished! Your pass should look like the one in the picture below:



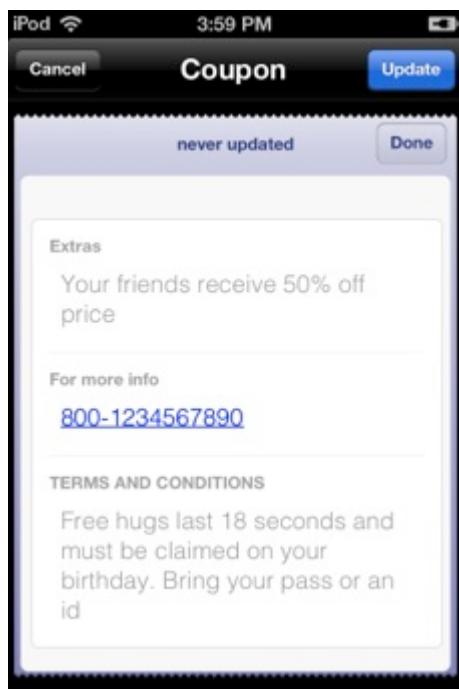
The front is now almost done, so do something for the back!

Open up **pass.json** and add a comma after the last closing square bracket (you should be proficient in adding the comma to the right spot by now). After the comma, add a fields dictionary for the back of the card:

```
"backFields" : [
  {
    "key" : "extras",
    "label" : "Extras",
    "value" : "Your friends receive 50% off price"
  },
  {
    "key" : "phone",
    "label" : "For more info",
    "value" : "800-1234567890"
  },
  {
    "key" : "terms",
    "label" : "TERMS AND CONDITIONS",
    "value" : "Free hugs last 18 seconds and must be claimed on your
birthday. Bring your pass or an id"
  }
]
```

Fortunately the back of the pass is scrollable, so you can add as much info as you need. (Yes, sometimes those Terms and Conditions sections get pretty lengthy, so scrolling is absolutely necessary.)

Now – it's time to see what the fields you just added look like on the pass. Build again following the instructions from last time and send the pass via email in order to see it on your device. Tap on the "i" button and the back will flip into view – it should look like the picture below:



So far so good! Also check out the phone number – it's automatically converted to a link, which you can tap on to be taken directly to the Phone app. You can include other interactive information on the back too – like addresses (which open up in Maps) and email addresses (which of course open up in Mail).

Part 2: The Pass Preview application

You probably already feel that it would be great if the process of building and previewing passes could be a little bit easier and faster. I mean, come on, do we really have to email the pass to ourselves every time we want to preview it?

Well – good news! Since I know you are going to want to try out many color combinations, pictures and texts before settling on the perfect pass, in this part of the chapter you're going to build an iOS app that will allow you to preview your passes a lot faster inside the iPhone simulator on your Mac. Yay!

There's one way to install passes in Passbook that I haven't mentioned yet. Besides sending a pass file over email and downloading it straight from a web server, you

can also use the new PassKit framework in iOS 6 to install a pass from within an iOS app.

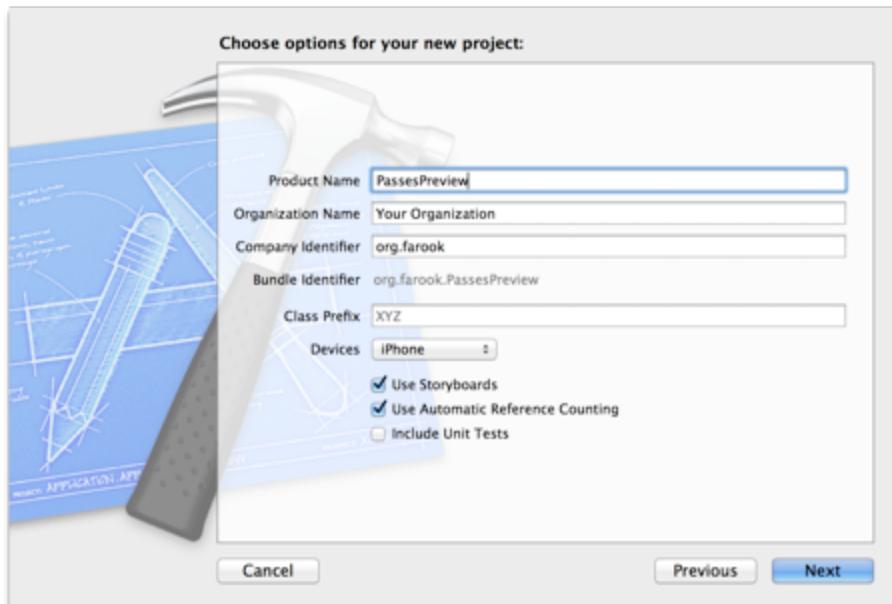
PassKit is a very simple framework, as it contains only three classes. To build a simple pass preview application, you are going to use the `PKAddPassesViewController`. This class takes in a `PKPass` instance and displays it on the screen.

In real life, your app will most probably fetch a pass from your server, or get it via iCloud, but in this example you're just going to use passes that are bundled with the app.

Note: OSX 10.8.2 has now support for pass previewing straight on your mac. If you double click on a `.pkpass` file you will see how the pass looks like. However, the preview is not 100% accurate with what the rendering is on an iOS device, and the only way to check whether the pass is actually valid still is to open it up on your device or simulator and try to import it.

Let's go!

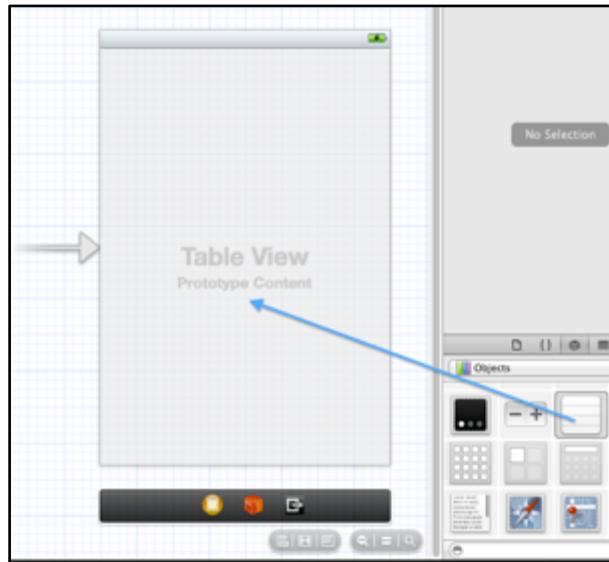
From Xcode's menu, select File/New/Project..., and then select iOS/Application/Single View Application and click Next. Name the project **PassesPreview** and make sure it's an iPhone app. Also ensure that **Use Storyboards** and **Use Automatic Reference Counting** are checked. Save the project in a location of your choice.



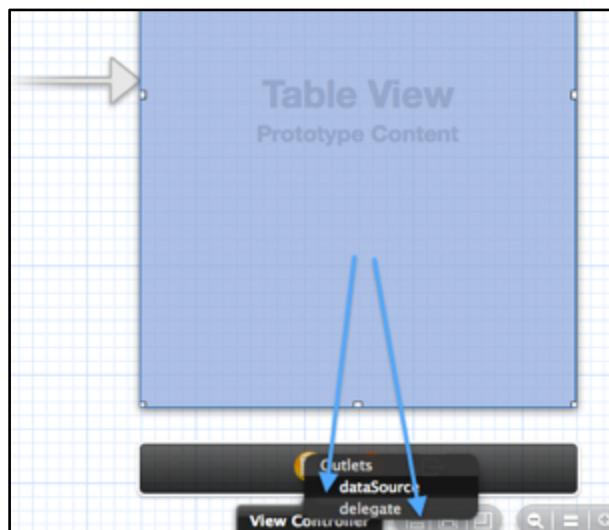
In the Project Navigator, select the Xcode project (the top item in the project tree) and in the strip on the right make sure the "PassesPreview" target is selected. Next click on the Build Phases tab and open up the Link Binary With Libraries strip.

Click on the (+) button, and in the list of frameworks double-click on "PassKit.framework" to include it in your Xcode project (or simply select PassKit.framework and click the Add button).

Now select **MainStoryboard.storyboard** – the app interface pops up. You need a full screen table, so grab a table view from the Objects palette and drop it on the main view:



Next, hold the Ctrl key and drag to the view controller object on the bottom palette. A popup menu will appear. Click on "dataSource", then repeat, and the second time choose "delegate". The table is now connected properly to your view controller.



And now, finally some Objective-C goodness!

Open up **ViewController.m** and replace its contents with this:

```
#import "ViewController.h"
#import <PassKit/PassKit.h> //1

@interface ViewController () //2
<UITableViewDelegate, UITableViewDataSource,
PKAddPassesViewControllerDelegate>
{
    NSMutableArray *_passes; //3
}
@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

//4
if (![PKPassLibrary isPassLibraryAvailable]) {
    [[[UIAlertView alloc] initWithTitle:@"Error"
                                message:@"PassKit not available"
                                delegate:nil
                                cancelButtonTitle:@"Pitty"
                                otherButtonTitles: nil] show];
    return;
}
}

@end
```

The code is pretty basic and there are just four things to note:

1. First you import the PassKit framework headers.
2. In the class extension of the `ViewController` class, you make it conform to the data source and delegate protocols of `UITableView`, and finally to the `PKAddPassesViewControllerDelegate` protocol.
3. You declare an array instance variable to hold the list of all the bundled pass files.
4. Finally in `viewDidLoad`, if the result of `[PKPassLibrary isPassLibraryAvailable]` is NO then you show a message to let the user know they don't have Passbook available. (The user in this case is you, but it was a nice opportunity to show you how to check for Passbook availability, so I snuck it in.)

The goal will be to check if there are passes bundled with the app and if so, to show a list of them. The app will also show nice previews of the passes when the user taps on a given pass file name.

Add the following at the end of `viewDidLoad`:

```
// 1 initialize objects
_passes = [[NSMutableArray alloc] init];

//2 load the passes from the resource folder
NSString* resourcePath =
[[NSBundle mainBundle] resourcePath];

NSArray* passFiles = [[NSFileManager defaultManager]
contentsOfDirectoryAtPath:resourcePath
error:nil];

//3 loop over the resource files
for (NSString* passFile in passFiles) {
    if ( [passFile hasSuffix:@".pkpass"] ) {
        [_passes addObject: passFile];
    }
}
```

What this simple code does is:

1. First initializes the `_passes` array.
2. Gets the list of all app resource files into the `passFiles` array.
3. Checks all file names in `passFiles` and adds the ones having extension `.pkpass` to the `_passes` array.

Pretty simple! Now you have the names of all passes bundled with your app. Next, let's show them in the table!

Add these three methods to **ViewController.m** in order to get the table view running:

```
#pragma mark - Table View
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return _passes.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

```
{  
    UITableViewCell *cell = [tableView  
dequeueReusableCellWithIdentifier:@"Cell"];  
    if (!cell) cell = [[UITableViewCell alloc] initWithStyle:  
UITableViewCellStyleDefault reuseIdentifier:@"Cell"];  
  
    NSString *object = _passes[indexPath.row];  
    cell.textLabel.text = object;  
    return cell;  
}
```

More simple code – you've probably done this many times already. However, do note some of the new code changes: you can now get the count of elements in an array by accessing the `count` method (it's not a property) with `return _passes.count`. Getting elements out of an array (or a dictionary) is now simpler, too:

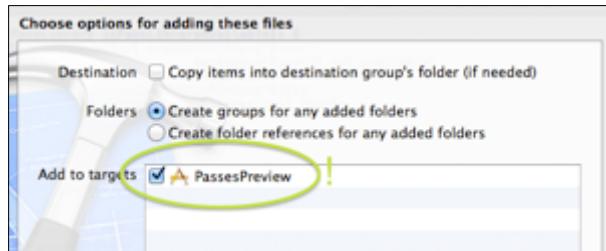
```
NSString *object = _passes[indexPath.row];
```

Some of these changes, like the dot-notation for methods, were around before iOS 6, but they are still fairly new. Whereas other stuff like the new `NSArray` notation is brand-spanking new, as you can read about in Chapter 2, "Programming in Modern Objective-C."

This should be enough – you can run the project and see no warnings or errors, and get an empty table view on screen.

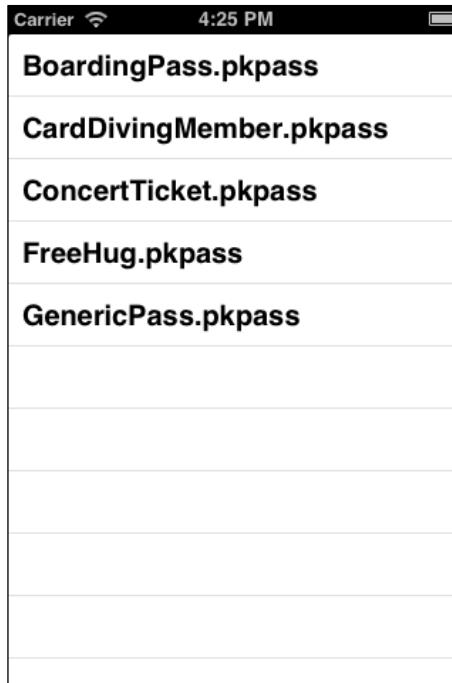
OK, time to include some passes with the app. Open up the folder with the assets for this chapter and copy all the `.pkpass` files from the **Passes** folder into your Xcode project by dragging the `.pkpass` files and dropping them on your Xcode project root.

Note: Make sure to check that those files are added to the `PassesPreview` target. By default, Xcode won't include them, as they are not code files or other known iOS resources.



Note: When you are working and designing your own passes do **NOT** check the “Copy items into ...” checkbox. It’s much simpler to just link the .pkpass files from your Pass folder into the Xcode project. When you do code changes or replace an image, then you just rebuild the pass and restart the Xcode project, and the changed pass pops up on the screen.

OK, now that you’ve got some passes in the app, hit run and you’ll see the list of passes:



Finally, you need to add the code to show the selected pass on the screen. Add it to **ViewController.m**:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    //1
    NSString* passName = _passes[indexPath.row];
    [self openPassWithName:passName];
}

-(void)openPassWithName:(NSString*)name
{
    //2
    NSString* passFile = [[[NSBundle mainBundle] resourcePath]
        stringByAppendingPathComponent: name];
```

```
//3
NSData *passData = [NSData dataWithContentsOfFile:passFile];

//4
NSError* error = nil;
PKPass *newPass = [[PKPass alloc] initWithData:passData
    error:&error];

//5
if (error!=nil) {
    [[[UIAlertView alloc] initWithTitle:@"Passes error"
        message:[error
        localizedDescription]
        delegate:nil
        cancelButtonTitle:@"Ooops"
        otherButtonTitles: nil] show];
    return;
}

//6
PKAddPassesViewController *addController =
    [[PKAddPassesViewController alloc] initWithPass:newPass];

addController.delegate = self;
[self presentViewController:addController
    animated:YES
    completion:nil];
}
```

All right – there's a fair bit of code, but it's easy to go over it:

1. First, when a table row is tapped you call `openPassWithName`.
2. Inside `openPassWithName` you recreate the full path to the `.pkpass` file.
3. Then you read the contents of the `.pkpass` file into an `NSData` instance.
4. Then you create a new `PKPass` object by calling its initializer `initWithData:error:`.
5. You check the `error` variable for any load issues.
6. In the end, you create a new `PKAddPassesViewController` instance – this is a special view controller, which takes in a `PKPass` object and displays it. Assign the `delegate` property to `self` and present the controller modally on the screen.

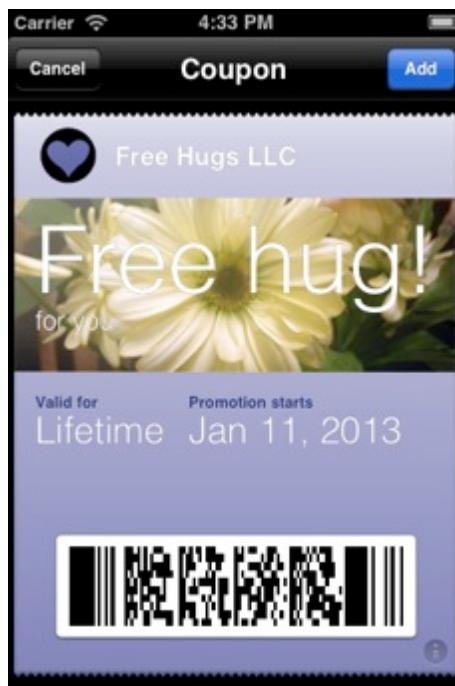
You need one more method – the one called when the user closes the pass dialogue. (You're not going to do anything special here, but it's good to know your options for when you develop something more complicated.)

Add this method required by the `PKAddPassesViewController` delegate protocol:

```
#pragma mark - Pass controller delegate

-(void)addPassesViewControllerAnimated:YES completion:nil;
{
    //pass added
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

Now you can run the project again and tap on the row saying **FreeHug.pkpass** – so cool! You can preview the passes and also import them into the Passbook app in your iPhone Simulator.



This is exactly the same dialogue that popped up inside your Mail app when you opened a pass attachment. Note that if you have an identical pass in your Passbook already, the Add button at the top right corner will be disabled.

Success!

One last touch – add the following at the end of `viewDidLoad`:

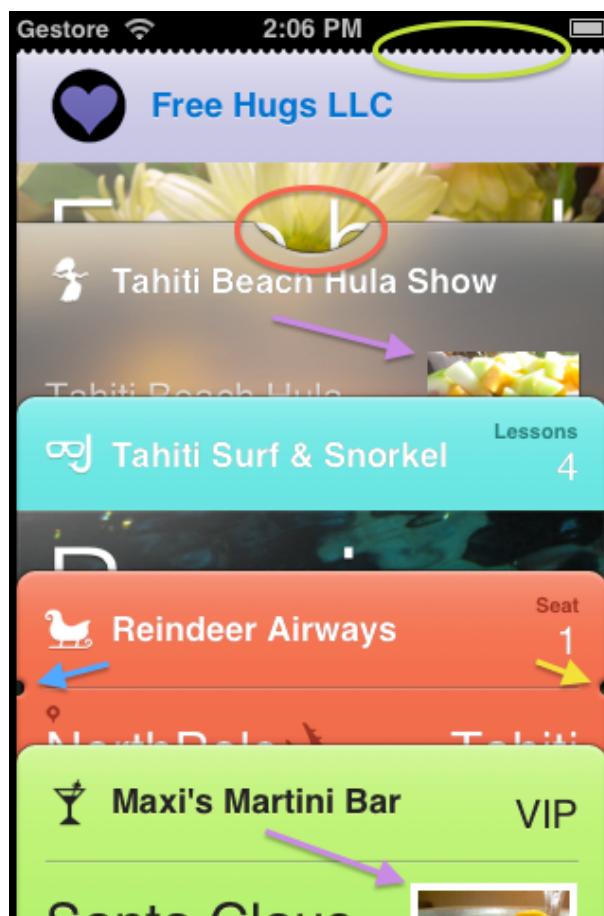
```
if ([_passes count]==1) {
    [self openPassWithName:[_passes objectAtIndex:0]];
}
```

When you are working on refining a single pass, you can save yourself the extra tap!

So far, you know everything you need to know about how to build a coupon, and you know how to easily test and preview your passes. This is the perfect combination of skills to approach the next topic in this part of the chapter: getting acquainted with the different styles of passes.

Dissecting pass styles...

Next you are going to look in more detail at the different styles of passes. Here is a preview of how they all look when imported in Passbook – you can see each type has a small style touch applied to the card:



So, how to create different passes? Go back to the source code of the Free Hug coupon you were working on previously and open up **pass.json**. In the top-level dictionary you've added a key called “coupon” – the value of this key is a dictionary with field definitions. All passes work the same way – the only thing you need to do is change the key name from `coupon` to the style you'd like to have!

Santa's well deserved vacation trip

To make the examples in this part a bit more fun, Vicki Wenderlich planned Santa's vacation trip and prepared all the passes he's going to need throughout his well-deserved vacation. With his iPhone's Passbook loaded with coupons, VIP cards, and plane tickets bought in advance, Santa can have some time off after the Christmas rush without carrying around extra papers in his sack of toys. The first example is of course Santa's plane boarding card for his flight to Tahiti.

Here's the content of pass.json from the BoardingPass.pkpass example included in this chapter's resources:

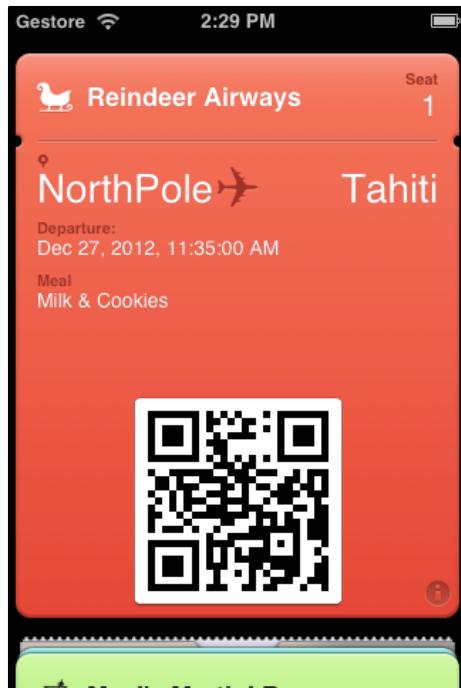
```
{
  "formatVersion" : 1,
  "passTypeIdentifier" : "pass.com.iOS6-By-Tutorials",
  "serialNumber" : "AX6184HJDG",
  "teamIdentifier" : "ABC1230000",
  "barcode" : {
    "message" : "AXB739-StClaus-A280",
    "format" : "PKBarcodeFormatQR",
    "messageEncoding" : "iso-8859-1"
  },
  "organizationName" : "Reindeer Airways LLC",
  "description" : "Train Boarding Pass",
  "logoText" : "Reindeer Airways",
  "foregroundColor" : "rgb(255, 255, 255)",
  "backgroundColor" : "rgb(230, 72, 56)",
  "boardingPass" : {
    "headerFields" : [
      {
        "key" : "header",
        "value" : "1",
        "label" : "Seat"
      }
    ],
    "primaryFields" : [
      {
        "key" : "from",
        "value" : "NorthPole"
      },
      {
        "key" : "to",
        "value" : "Tahiti"
      }
    ],
    "secondaryFields" : [
      ...
    ]
  }
}
```

```
{  
    "key" : "meal",  
    "label" : "Meal",  
    "value" : "Milk & Cookies"  
}  
,  
"auxiliaryFields" : [  
    {  
        "key" : "departure",  
        "label" : "Departure:",  
        "dateStyle" : "PKDateFormatMedium",  
        "timeStyle" : "PKDateFormatMedium",  
        "value" : "2012-12-27T10:35Z"  
    }  
,  
    "backFields" : [  
        {  
            "label" : "terms & conditions",  
            "key" : "terms",  
            "value" : "Ticket is non-refundable. Please specify which  
type of cookie you want upon check-in: chocolate chip, peanut  
butter, or M&M sugar cookie."  
        }  
,  
        "transitType" : "PKTransitTypeAir"  
    }  
}
```

As you can see, the structure of the file is almost the same as what you had before, with the difference that instead of `coupon` you have a `boardingPass` key. There's one other thing of interest: inside the `boardingPass` dictionary there's an extra field called "`transitType`". This field is specific for the boarding pass and is what sets the little transit icon shown on the front side of the pass.

For `transitType`, depending on what kind of boarding pass are you building, you can use one of the predefined type constants: `PKTransitTypeAir`, `PKTransitTypeTrain`, `PKTransitTypeBus`, `PKTransitTypeBoat`, and `PKTransitTypeGeneric`.

And that's the difference between building a coupon and a boarding pass! The source code above results in this nifty boarding pass:



Can you spot Santa's meal preferences on the boarding card? So can the flight attendant – he's up for a relaxing flight!

Boarding Pass style considerations

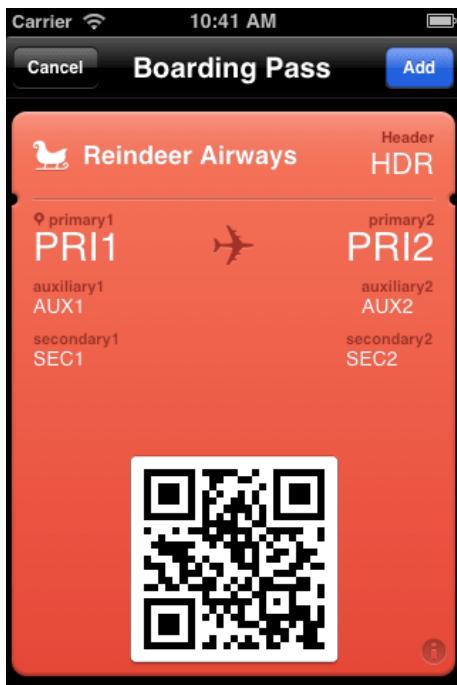
- Note that the primary fields are using quite a large font so that they can't show more than a few characters of text. Longer destination names like "San Francisco" are going to be cut off, so consider using destination codes instead of names. It would be best to show the full name of the city as the field label, and the destination code as the value.
- The boarding pass doesn't show any pictures, so use only the icon and logo image files.
- It's a great idea to use the header field for things like airport departure gate, departure platform or a seat number.
- Make sure you are using the correct transit type for your boarding pass! Here are the different icons shown, depending on type (all the types are listed on the previous page):



And here's also a preview of the positions of all available fields you can use (of course you don't need to use all of the fields; this is to give you an overview of the styling and the positions of the available fields):



When designing a new pass there's one major point you have to always pay attention to: the different styles of barcodes take up different amount of space on the front side of the pass. The PDF417 barcode has a horizontal layout, so it kind of fits nicely at the bottom of the card – leaving more space for the pass fields (like in the example above). On the other hand the QR barcode has a square shape and takes up more space vertically. Observe the same pass from above when it features a QR barcode:



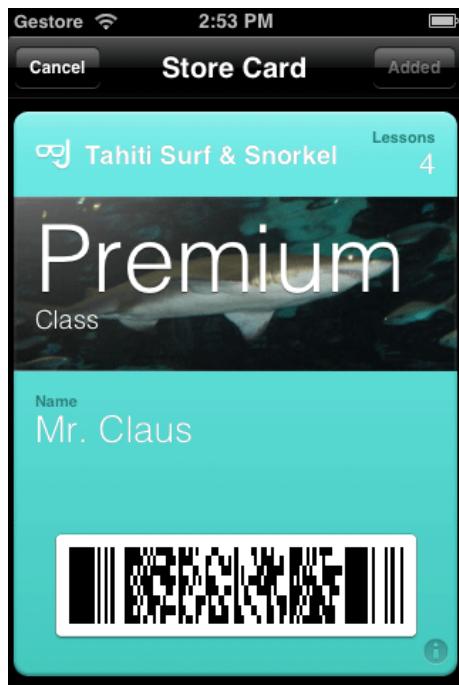
You can clearly see there's less vertical space between the fields so that the barcode can also fit on the card; therefore: always start designing a pass with the correct type of barcode, so that you can have the correct field layout from the start!

Note: If you reuse any of the code from the example passes in this part of the chapter, don't forget to change the team and pass type identifiers to your own – otherwise, your passes won't import to Passbook.

Are you a premium member?

Next, let's look into store cards. These are very popular and already widely used, so I bet many merchants will want to use the iPhone pass format to digitalize their store cards.

There is a store card example included with this chapter. Santa is up for some active recreation so he bought a premium package from the Tahiti Surf & Snorkel school. His pass also conveniently shows him how many pre-paid lessons he has left on his account with the school.



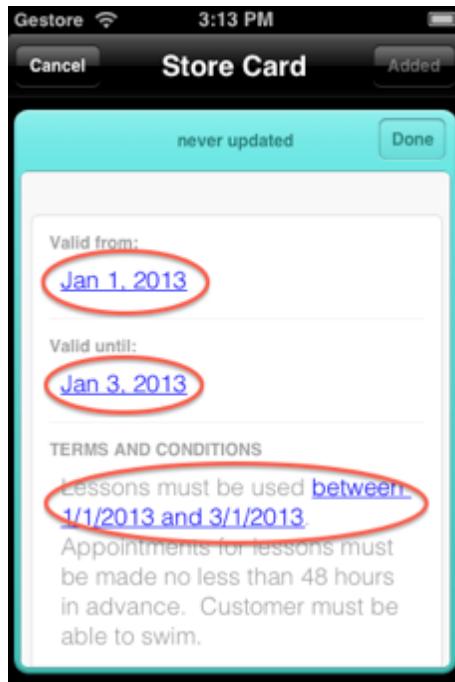
As you can see, the layout of the store card is somewhat similar to the coupon, but the header styling is different. Have a quick look at the source code for the pass above:

```
{
  "formatVersion" : 1,
  "passTypeIdentifier" : "pass.com.iOS6-By-Tutorials",
  "serialNumber" : "00012",
  "teamIdentifier" : "ABC1230000",
  "barcode" : {
    "message" : "q2eq3aaa",
    "format" : "PKBarcodeFormatPDF417",
    "messageEncoding" : "iso-8859-1"
  },
  "organizationName" : "Tahiti Surf & Snorkel",
  "description" : "Diving Lessons",
  "logoText" : "Tahiti Surf & Snorkel",
  "foregroundColor" : "rgb(255, 255, 255)",
  "backgroundColor" : "rgb(68, 200, 190)",
  "storeCard" : {
    "headerFields" : [
      {
        "key" : "lessons",
        "label" : "Lessons",
        "value" : "4"
      }
    ]
  }
}
```

```
],
  "primaryFields" : [
    {
      "key" : "balance",
      "label" : "Class",
      "value" : "Premium"
    }
  ],
  "secondaryFields" : [
    {
      "key" : "memberName",
      "label" : "Name",
      "value" : "Mr. Claus"
    }
  ],
  "backFields" : [
    {
      "key" : "date",
      "label" : "Valid from:",
      "dateStyle" : "PKDateFormatMedium",
      "timeStyle" : "PKDateFormatNone",
      "value" : "2013-01-01T00:00Z"
    },
    {
      "key" : "date",
      "label" : "Valid until:",
      "dateStyle" : "PKDateFormatMedium",
      "timeStyle" : "PKDateFormatNone",
      "value" : "2013-01-03T00:00Z"
    },
    {
      "key" : "terms",
      "label" : "TERMS AND CONDITIONS",
      "value" : "Lessons must be used between 1/1/2013 and  
3/1/2013. Appointments for lessons must be made no less than 48  
hours in advance. Customer must be able to swim."
    }
  ]
}
```

The basics, which you already well know and love, are all there, so I'm not going to cover them again. However, there are a few things to discuss inside the `backFields` dictionary. You have two date fields and a text field for the terms and conditions, and all three fields contain dates. This is a great opportunity to connect the pass

with the Calendar on the iPhone! Tap the “i” button at the bottom-right corner and have a look:



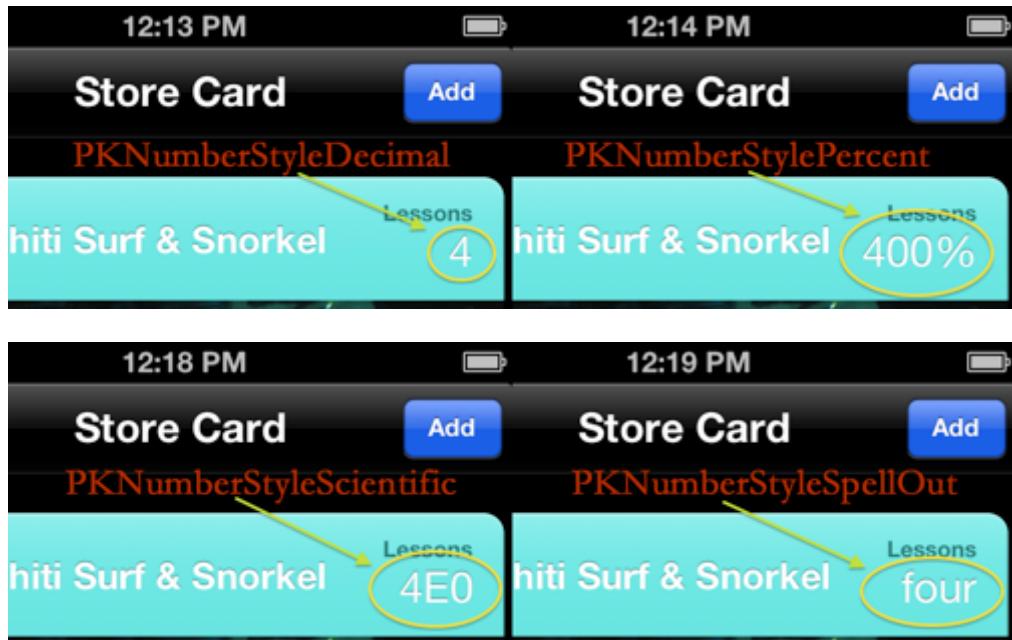
Passbook auto-detects the dates and turns them into links. When they are tapped, the user is presented with an action sheet, which can take them to the specific day in the Calendar, or help them create an event in the Calendar for the specific day or period. Outstanding!

Have a look at the `headerFields` section. This field always shows a number, but you have its value as a string in the example above. Let's quickly see how can you use an actual number as value and then change the number formatting.

If you supply a number value for any of the pass field, you can also provide an extra key (besides “key”, “label” and “value”) called `numberStyle` to format how the number would look like on the pass; the field definition would look like this:

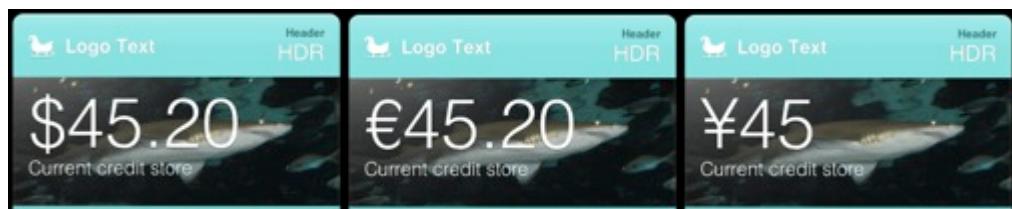
```
"headerFields" : [
  {
    "key" : "lessons",
    "label" : "Lessons",
    "value" : 4,
    "numberStyle": "PKNumberStyleDecimal"
  }
]
```

Note that there are no quotes around the value and the extra key is present as discussed above. Here's the list of possible number formatting values for the same value – the number 4:



Store card style considerations

- Just like the coupon type of pass, the store card checks for **strip.png** and **strip@2x.png** files in the pass bundle and if they are present they will be used as a background image to the primary fields on the upper part of the pass card.
- Usually a store card holds the customer's credit with the store. It's a good idea to put that information in the primary area of the pass. Passbook can render different currency styles as well, so if you are showing money amounts, consider showing them in the user's currency, like so:



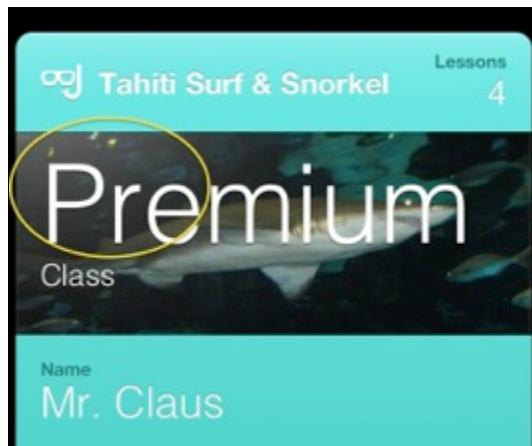
Passbook even automatically removed the digits after the floating point for the amount in Japanese Yen, because the Yen doesn't use a decimal point! (1 is the lowest possible positive value.)

To style a field as a currency amount, you need a numeric value (so, no quotes around the number) and there has to be an extra key to define which currency to use. For an amount in Japanese Yen, that would result in the following code:

```
{
  "key" : "amount",
  "label" : "Current credit store",
  "value" : 45.20,
  "currencyCode" : "JPY"
```

{

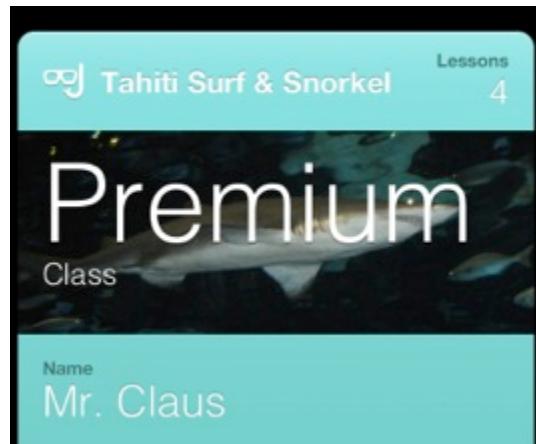
- Passbook adds automatically a nice glossy shine over the strip image (the shark image in the example above, which makes the pass look really cool):



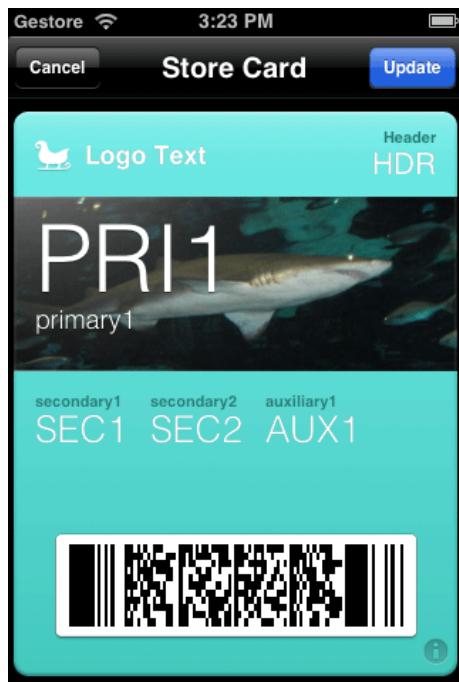
Sometimes however, I am sure you will want to have only your own visual goodness in the strip area, without the additionally added shine. In that case add a new key in your pass.json file to instruct Passbook that you want to disable the strip shine, like so:

```
{  
    ...  
    "logoText" : "Tahiti Surf & Snorkel",  
    "foregroundColor" : "rgb(255, 255, 255)",  
    "backgroundColor" : "rgb(68, 200, 190)",  
    "suppressStripShine" : true,  
    "storeCard" : {  
        ...  
    }  
}
```

If you set `suppressStripShine` to `true` on the same store card pass from above the strip will look like that:



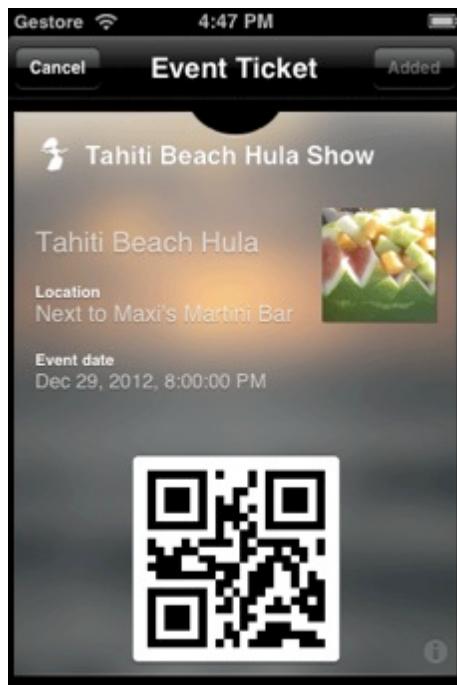
And finally, here are the kinds of fields a store card can show on its front:



It's a ticket, it's a pass... it's an iPhone!

The next pass type on the list is the event ticket – it can be a concert ticket, cinema ticket or any other kind of ticket you can come up with!

Switch back to the PassPreview app and select ConcertTicket.pkpass. A stylish ticket card pops up:



As you can see, Santa is not going to bed early – instead he plans to enjoy a hula show and a nice evening on the beach. It's easy when you plan ahead and have all the tickets in your pocket – conveniently stored on your iPhone!

The event ticket pass has a couple of unique features. If you provide background image files (background.png and background@2x.png inside the .pkpass bundle), it gets stretched, blurred, and is set as the background of the whole card. If you provide a thumbnail image (thumbnail.png and thumbnail@2x.png), the image shows up on the right-hand side.

Note that since the background image is rendered over the whole front side of the pass, you don't need to specify "backgroundColor" in your pass.json file.

Besides those specifics, the rest you already know well:

```
{
  "formatVersion" : 1,
  "passTypeIdentifier" : "pass.com.iOS6-By-Tutorials",
  "serialNumber" : "000024567",
  "teamIdentifier" : "ABC1230000",
  "barcode" : {
    "message" : "12946390566",
    "format" : "PKBarcodeFormatQR",
    "messageEncoding" : "iso-8859-1"
  },
  "organizationName" : "Tahiti Beach Hula Show",
  "description" : "Entrance ticket to Tahiti Beach Hula Show",
  "logoText" : "Tahiti Beach Hula Show",
```

```
"foregroundColor" : "rgb(255, 255, 255)",
"backgroundColor" : "rgb(242, 121, 55)",
"eventTicket" : {
    "primaryFields" : [
        {
            "key" : "name",
            "value" : "Tahiti Beach Hula"
        }
    ],
    "secondaryFields" : [
        {
            "key" : "location",
            "label" : "Location",
            "value" : "Next to Maxi's Martini Bar"
        }
    ],
    "auxiliaryFields" : [
        {
            "key" : "date",
            "label" : "Event date",
            "dateStyle" : "PKDateStyleMedium",
            "timeStyle" : "PKDateStyleMedium",
            "value" : "2012-12-29T19:00Z"
        }
    ],
    "backFields" : [
        {
            "key" : "terms",
            "label" : "Terms & conditions",
            "value" : "This ticket may cause happiness and good cheer!
Free lei with ticket."
        }
    ]
}
```

After seeing a few different pass.json files, it becomes pretty easy to understand the pass source code! Again, you can include many kinds of information on the front and back of the pass in the manner already covered.

Event Ticket styling considerations

- If you provide a background image, but no thumbnail image, then Passbook will also show the background image in the thumbnail spot on the right-hand side.
- You cannot disable the blur effect on the background – that's how Apple makes sure the text information stands out.

- For event tickets like conferences, the thumbnail image is a great way to include the attendee's photo straight on the front of the pass.

Finally, here's also a preview of all the possible fields you can use on the front of an event ticket:



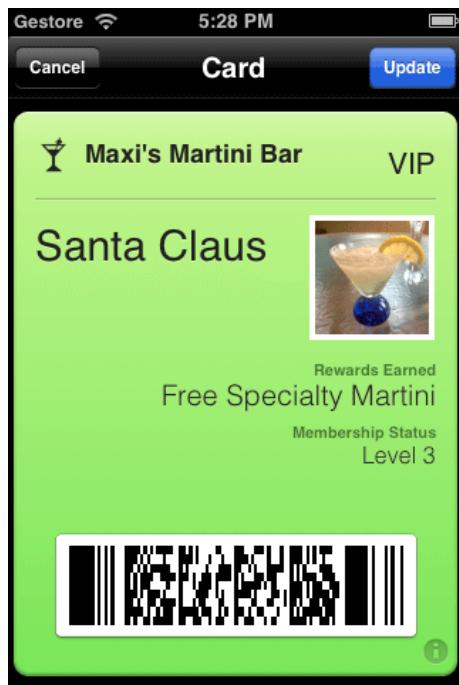
It's the pass you always wanted to have!

Finally, we get to the pass you always wanted to have! Or to create. Or to create and then have!

The last type of pass is the generic pass. It has basic styling, can feature many different text fields, can display a picture on the front side, and you can use it for all kinds of passes that don't fit well in the other four categories – coupon, boarding pass, store card, or event ticket.

And now we discover Santa's drink of choice: after all the hard work delivering billions of presents around the world in a night, he's up for a cold martini on the beach. Enjoy responsibly, Santa!

Open up PassPreview and tap on GenericPass.pkpass:



As you can see, this card doesn't fit neatly into any of the other four pass styles. It gives the bearer access to the VIP lounge of Maxi's Martini Bar and a free martini each day. It's more like a club membership, for which there exists no pre-defined style.

But this "generic" pass actually looks pretty good! Why yes – you still have all the possible ways to customize it that were already discussed! There's nothing special inside `pass.json`, except of course for the `passType` key being `generic`:

```
{
  "formatVersion" : 1,
  "passTypeIdentifier" : "pass.com.iOS6-By-Tutorials",
  "serialNumber" : "123456",
  "teamIdentifier" : "ABC1230000",
  "barcode" : {
    "message" : "7465-454-2234-1",
    "format" : "PKBarcodeFormatPDF417",
    "messageEncoding" : "iso-8859-1"
  },
  "organizationName" : "Maxi 's Martini Bar",
  "description" : "Generic Pass Example",
  "logoText" : "Maxi 's Martini Bar",
  "foregroundColor" : "rgb(33, 33, 33)",
  "backgroundColor" : "rgb(122, 230, 88)",
  "generic" : {
    "headerFields" : [
      {
        "key" : "name",
        "value" : "Santa Claus"
      }
    ],
    "bodyFields" : [
      {
        "key" : "image",
        "value" : "https://example.com/martini.jpg"
      },
      {
        "key" : "text",
        "value" : "Free Specialty Martini"
      },
      {
        "key" : "text",
        "value" : "Membership Status Level 3"
      }
    ],
    "footerFields" : [
      {
        "key" : "barcode",
        "value" : "7465-454-2234-1"
      }
    ]
  }
}
```

```
        "key" : "header",
        "value" : "VIP"
    }
],
"primaryFields" : [
{
    "key" : "member",
    "value" : "Santa Claus"
}
],
"secondaryFields" : [
{
    "key" : "faux1",
    "value" : ""
},
{
    "key" : "secondary1",
    "label" : "Rewards Earned",
    "value" : "Free Specialty Martini",
    "textAlignment" : "PKTextAlignmentRight"
}
],
"auxiliaryFields" : [
{
    "key" : "faux2",
    "value" : ""
},
{
    "key" : "status",
    "value" : "Level 3",
    "label" : "Membership Status",
    "textAlignment" : "PKTextAlignmentRight"
}
],
"backFields" : [
{
    "label" : "terms & conditions",
    "key" : "terms",
    "value" : "Must show proof of ID to claim rewards."
}
]
}
```

Wait... what? (If you didn't notice anything strange, maybe you need to look over the code once more?)

What is this field supposed to do?

```
{  
    "key" : "faux1",  
    "value" : ""  
},
```

Yes – it's time for some *trickstery*! The position of the fields on the front of the pass are predefined, and so... it goes like this: the first secondary field shows up on the left-hand side, the second secondary field shows up on the right-hand side, the first auxiliary field shows up... etc., etc.

So what if you want to show a piece of text where normally the second secondary field would be, but you don't need to show anything in the first secondary field?

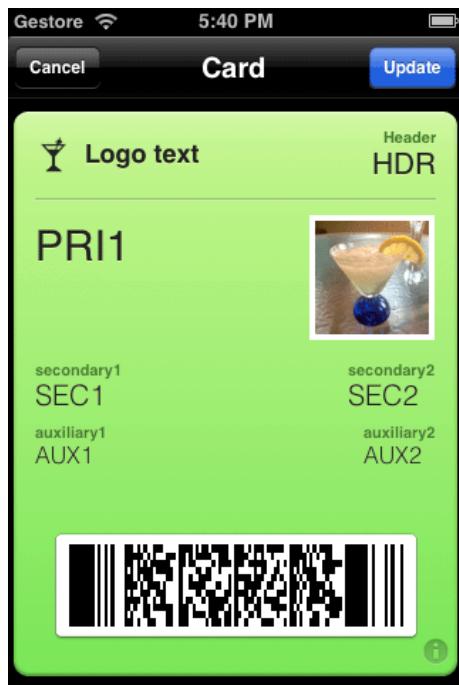
Aha! Define a faux first secondary field with an empty value and no label (like in the code example above). Neat!

Another thing to note is the “`textAlignment`” key used in the code above. Using this key, you can tell Passbook which way to align the text in a given field. You can choose between the following list of predefined alignment constants:

`PKTextAlignmentLeft`, `PKTextAlignmentCenter`, `PKTextAlignmentRight`, `PKTextAlignmentJustified`, and `PKTextAlignmentNatural`.

The image files you can use for the generic pass type are only **thumbnail.png** and **thumbnail@2x.png** - as on the screenshots above, if present, they will pop on the upper right hand side of the store card.

Here's an overview of all the possible fields you can use on the front of the generic type of pass:



OK! You've made it this far and I hope the trip was rewarding! You know everything you need about designing, coding and previewing passes. You've learned a whole lot, but wait – there's more! And it gets more interesting by the page!

Part 3: Getting your passes out there

You are now ready to create passes and send them to customers. It looks like a hard job to create those by hand and send them out via email, right? Thankfully, there's an easier way. In this part of the chapter, you are going to create a simple website that takes in the visitor's information, generates a personalized pass, and sends it straight to the user's mailbox! Woot!

First of all you need a web server, where you'll host the web site.

I will assume you have at your disposal an Apache web server with PHP support. If you don't, then have a look at this short tutorial online:

<http://www.raywenderlich.com/2965/how-to-write-an-ios-app-that-uses-a-web-service>

It will guide you through setting up a web server on your own computer. Of course, if you want to use this in a production environment, be sure to have a real hosting package.

You'll create the website in an Apache/PHP environment, as this is probably the most popular web development setup nowadays. However, once you understand the process of dynamically creating and signing a pass (and you know much about

it already), you'll be able to recreate it in your favorite web development language (if other than PHP).

Note: You can use a local web server on your Mac to complete the demo project, but in the last stage of development you will need to send out email messages from PHP, and this you (usually) can't do from your local machine. To test and complete the project, you will need a real web server with functioning email services where you can send mails via PHP.

Free Hugs online promotion website

Free Hugs LLC is looking to expand its services and to attract more customers. They have more hugs than they know what to do with! They decide to build a small promotional website, where visitors have the option of filling out a simple questionnaire. If a visitor correctly answers the promotional question about the history of Free Hugs LLC, they will receive a coupon for one free hug.

Your task is to build all of this from the ground up!

Fire up your favorite web-coding editor and create an **index.php** file in the root folder of your webserver.

Note: I've used Coda for years now – it's an awesome all-in-one IDE and it's also utterly beautiful. If you don't already have a favorite web IDE, give it a try – there's a free 30-day trial download here: <http://panic.com/coda/>

Paste the initial HTML of your questionnaire page inside **index.php**:

```
<html>
<head>
<style>
    form {border: 1px solid #611111; width: 400px;
          padding: 10px; background: #ccefff;
          border-radius: 15px;}
    h1 {font: italic bold 20px/30px Georgia, serif;}
    h2 {font: normal bold 16px/30px Georgia, serif;}
</style>
</head>
<body>

</body>
</html>
```

This just defines some CSS code you are going to need to style your page contents.

Now add the form with the questionnaire. Insert the following after <body>:

```
<h1>Free Hugs bonus program:</h1><br/>
<form action="getpass.php" method="post">
<h2>Bonus game question:</h2>
In which year was "Free Hugs" incorporated?
<select size="1" name="year">
  <option>1999</option>
  <option>2003</option>
  <option>2009</option>
</select><br/>

<h2>Your data:</h2>
Your name:
<input name="name" value="" /><br/>

Your email address:
<input name="email" value="" /><br/>

Your favorite "Free hugs" store:
<select size="1" name="location">
<option value="0">City square</option>
<option value="1">Kiosk at the beach</option>
<option value="2">Big City Mall floor 1</option>
<option value="3">Big City Mall floor 4</option>
</select>

<br/><br/>
<input type="submit" value="Participate in the bonus game"/>

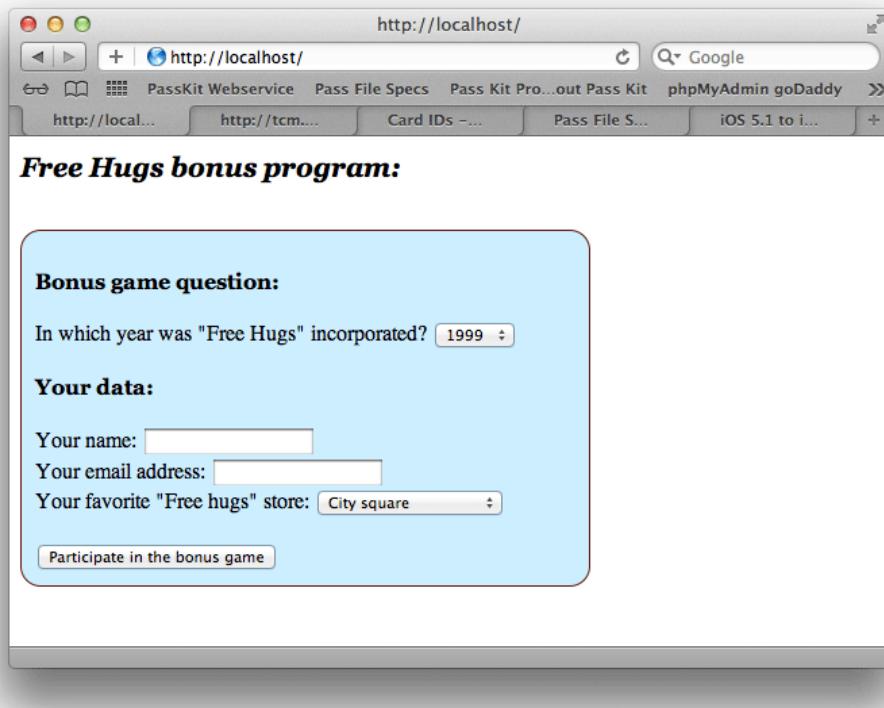
</form>
```

If you know HTML, you'll be able to make your way around, but just in case you don't, here's a short description of what you're doing with this code:

- The H1 tag renders the title of the page.
- Then you declare that the form will submit its contents to a file called getpass.php. There are two ways to main ways to send data over the web – GET and POST – this specifies POST.
- Next comes the promotion question and a drop box with three possible answers for the visitor to choose from.
- In the second half of the form, the visitor fills in their personal data – there are fields for their name and email address.
- Finally, the visitor can choose their favorite Free Hugs location in town.

- At the bottom of the form, there's a button that submits the data to the `getpass.php` script.

If you're running the web server on your own Mac, open up <http://localhost/> (or the appropriate URL if you're running on a web server) in your web browser and have a look at the page so far:



Looking good! Well, at least “good” in the sense that it’s working; as for how to create aesthetically-pleasing pages, there are many other books. ☺

You can select items from the drop boxes and fill in values. Submitting the form leads to a rather unpleasant *404* page. That’s because the `getpass.php` script has not been created yet. That’s the script that will check the visitor’s answer to the promo question and generate a pass if the visitor answered correctly.

Create a new file in the web server’s root web folder called **getpass.php**. (If you use Coda, it’s File/New/PHP-HTML and then Cmd+S to choose location and save the file.)

OK! Start with checking the visitor’s input. You’re not going to implement a thorough check, but you should always add at least the bare minimum of input validation. Paste in the following code:

```
<?php  
  
//1 VERY simple input validation  
if (strlen($_POST['year'])<4 || strlen($_POST['email'])<6 ||
```

```
strlen($_POST['location'])<1) {  
    header("Location: index.php?message=Enter valid data");  
    exit();  
}  
  
//2 check for correct answer  
if ($_POST['year']!='2003') {  
    header("Location: index.php?message=That was not the correct  
answer, try again");  
    exit();  
}  
  
?>
```

This validation is indeed very simple:

1. First you check the length of the submitted answers – this checks whether there was something submitted at all, and whether the values are of sufficient length.
2. Then you check whether the user submitted the correct answer to the promo question. (Surely everyone knows Free Hugs LLC was founded in 2003, but an extra check is always good.)

Ah! The beauty of user validation on the web – what to do if the submitted data was not correct or valid? In this example, you are just redirecting the visitor to the form page with an added message parameter to the URL (a parameter named “message”, sent via GET).

Note to keep things simple, it doesn’t pass back any other data (such as what the user entered previously) – in a real world situation, you would probably want to send back the user’s data and populate in the form with it, so the visitor doesn’t have to fill in the form again.

Switch back to **index.php** and just after <style> paste in some more CSS:

```
div {border: 1px solid #611111; width:400px;  
background: #efffef; padding:10px; border-radius: 15px;}
```

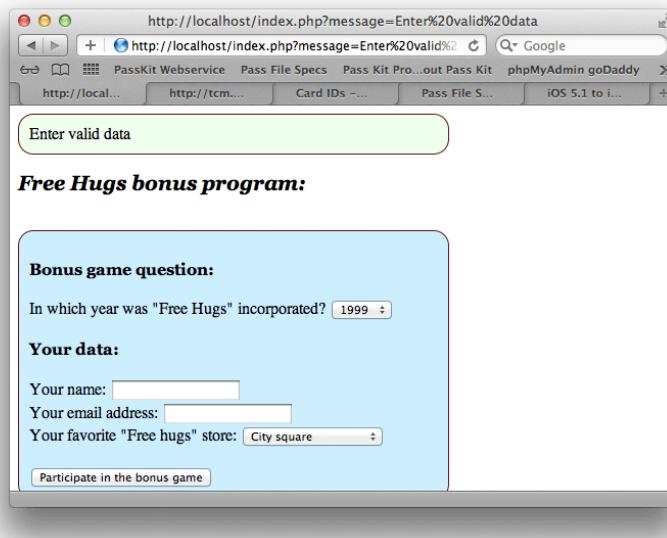
This is the CSS styling of the message you will show if the visitor entered invalid data. And you also need the code to show the message on the page, so directly after <body> paste in:

```
<?php  
if ($_GET['message']!='') {  
    print "<div>".$_GET['message']. "</div>";  
}  
?>
```

This code checks if there was a message parameter passed in the URL, and if so, prints it on the page.

Your form validation should work now! Open up <http://localhost/> again in your browser, and without entering any information, click the “Participate in the bonus game” button.

Cool! Instead of the *404* page, now you see the form again, but with a nifty message at the top telling you that you need to be more careful when filling in your data:



Now you need to provide the skeleton of the pass. Most of the data in your `pass.json` file won't change per visitor – the images and the meta-information will remain the same for every pass you issue. So these can remain static values, and you only need to change (via PHP) the fields that will have unique values (the pass serial number, the owner's name, etc.).

Since you already developed a Free Hug coupon, I am not going to go into much detail about all the files used to create it. Open up the resources folder for this chapter and copy the folder called “`pass`” into the folder on your web server where `getpass.php` is.

If you look inside this folder, you'll find all the images required and also a `pass.json` starter file (inside the “`source`” sub-folder):

```
{
  "formatVersion" : 1,
  "passTypeIdentifier" : "pass.com.yourdomain.couponfreehug",
  "serialNumber" : "<INSERT SERIAL NUMBER>",
  "teamIdentifier" : "<INSERT YOUR TEAM IDENTIFIER>",
  "barcode" : {
    "message" : "All you need is love",
  }
}
```

```
        "format" : "PKBarcodeFormatPDF417",
        "messageEncoding" : "iso-8859-1"
    },
    "locations" : [],
    "organizationName" : "Free Hugs LLC",
    "description" : "Coupon for 1 Free Hug",
    "logoText" : "Free Hugs LLC",
    "foregroundColor" : "rgb(255, 255, 255)",
    "backgroundColor" : "rgb(135, 129, 189)",
    "labelColor" : "rgb(45, 54, 129)",
    "coupon" : {
        "primaryFields" : [
            {
                "key" : "offer",
                "label" : "for you",
                "value" : "Free hug"
            }
        ],
        "secondaryFields" : [
            {
                "key" : "name",
                "label" : "Name:",
                "value" : "<INSERT NAME>"
            }
        ],
        "auxiliaryFields" : [
            {
                "key" : "date",
                "dateStyle" : "PKDateFormatMedium",
                "timeStyle" : "PKDateFormatNone",
                "label" : "Promotion starts",
                "value" : "2013-01-11T00:00Z"
            }
        ],
        "backFields" : [
            {
                "key" : "extras",
                "label" : "Extras",
                "value" : "Your friends receive 50% off price"
            },
            {
                "key" : "phone",
                "label" : "For more info",
                "value" : "800-1234567890"
            }
        ],
        "barcode" : {
            "format" : "PKBarcodeFormatPDF417",
            "messageEncoding" : "iso-8859-1"
        }
    }
}
```

```
{  
    "key" : "terms",  
    "label" : "TERMS AND CONDITIONS",  
    "value" : "Free hugs last 18 seconds and must be claimed  
on your birthday. Bring your pass or an id"  
}  
]  
}  
}
```

This is almost the same pass.json file you had for the Free Hug coupon. Can you spot the key difference? There's one new key in the top-level dictionary that you didn't have before – "locations". For the time being, leave it be – it will come up again later.

Before continuing, make sure you change the value for "teamIdentifier" to your own team identifier. Also, be sure to change the value for "passTypeIdentifier" to your own pass type identifier that you set up in iTunes Connect earlier.

Next you are going to develop a PHP class to handle dynamic generation of passes. This class will be robust and reusable, so you can use it in your future projects if you wish.

Create a new file called **Pass.php** (in the same folder where index.php and getpass.php are) and paste in some initial code:

```
<?php  
class Pass  
{  
    private $workFolder = null;  
    private $ID = null;  
  
    var $content = null;  
    var $passBundleFile = null;  
  
}  
?>
```

You define a new class called `Pass` and several class members:

- The private variable `workFolder` will hold the path to a temporary folder where you will store the pass files during build time.
- In `ID` you will store a unique ID for the pass.
- Inside `content` you will store the pass.json contents.
- Finally, `passBundleFile` will hold the file path to the bundled .pkpass file.

First of all, you need the class to copy the source files into a temporary location – that way you can handle the generation of more than one pass at the same time, when, for example, you have multiple simultaneous visitors to your site.

Add a method that will read files from a given folder and copy them to a given temporary location. (The path to the temp folder will be stored in `workFolder` for you in advance. You'll implement that in a moment.) Just before the closing curly bracket, at the end of the file, add this method:

```
private function copySourceFolderFilesToWorkFolder($path) {

    //recurse over contents and copy files
    $files = new RecursiveIteratorIterator(
        new RecursiveDirectoryIterator($path),
        RecursiveIteratorIterator::SELF_FIRST);

    foreach($files as $name => $fileObject) {
        if (is_file($name) &&
            substr($fileObject->getFileName(), 0, 1)!=".") {

            copy($name,
                  $this->workFolder."/".str_replace($path."/","", $name));
        } else if (is_dir($name)) {
            mkdir($this->workFolder."/".
                  str_replace($path."/","", $name));
        }
    }
}
```

This code looks a bit confusing at first sight I'm sure, but it is the correct way to do this task. You need to copy the source files not only from the pass folder, but also from all the subfolders inside it. Wait... what? Yes, the pass can also contain subfolders. So far you haven't needed subfolders, but as you'll see later in this chapter, you will.

This first line of code gets you the recursive list of files inside your source folder:

```
$files = new RecursiveIteratorIterator(
    new RecursiveDirectoryIterator($path),
    RecursiveIteratorIterator::SELF_FIRST);
```

After you get the list of files, you loop over them and check for a few things:

- In the case that the filename starts with a “.” you're not interested in the file – it's probably just a file automatically-created by OSX like the `.DS_Store` file.
- If it's any other type of file, you copy it over to the temporary location stored in `workFolder`.

- If the current item is a folder, you create a corresponding subfolder in the target location.

OK! After you add this handy helper method, go on and add the class constructor, which takes in a source folder, copies the files to the temp location and initializes the `Pass` instance. It's not so difficult – again, before the closing curly bracket of the class, add the constructor's body:

```
//make new instance from a source folder
function __construct($path) {
    assert(file_exists($path."/pass.json"));

    $this->ID = uniqid();

    $this->workFolder = sys_get_temp_dir()."/".$this->ID;
    mkdir($this->workFolder);
    assert(file_exists($this->workFolder));

    $this->copySourceFolderFilesToWorkFolder($path);

    $this->readPassFromJSONFile($this->workFolder."/pass.json");
}
```

All right, let's see what's going on in this piece of code.

The constructor takes in a path to the source files folder. You, being a very bright chap, immediately call `assert` and check whether a `pass.json` file exists in the given location. Good for you!

Next you create a temporary folder to store the files while you work with them. PHP has a handy function for generating unique IDs – `uniqid()`. You use it above to generate the name for the temporary folder:

```
$this->ID = uniqid();
```

The format of this unique ID can vary, but in general it looks something like this: `4ff89f2c0532e`.

`sys_get_temp_dir()` is yet another handy PHP function that gives you the path to the system temporary folder. By using `mkdir()`, you create your temporary subfolder inside the system temp folder, and just to double check you call `assert` to verify whether the operation was successful.

You copy recursively all the pass source files by calling the method you added previously, `$this->copySourceFolderFilesToWorkFolder()`, and then you load `pass.json` by calling `$this->readPassFromJSONFile()`. This last method you still don't have in the class implementation, so add it!

```
//import a json file into the object
function readPassFromJSONFile($filePath)
{
    //read the json file and decode to an object
    $this->content =
        json_decode(file_get_contents($filePath),true);
}
```

Loading JSON is easy in PHP. The new method just takes in the path to the pass.json file, reads the file contents (by using `file_get_contents`), and finally `json_decode` takes in the JSON text and returns actual variables. So after you call this method, you have the data structure that describes your pass in the class member `$this->content`.

Since the constructor creates the temporary folder automatically, make sure you don't leave trash files behind – add the cleanup process to the class destructor. The cleanup works almost the same way as the copy method: it recursively goes through the files inside the temp folder, but instead of copying, it deletes them. Paste these two methods inside the class:

```
//delete all auto-generated files in the temp folder
function cleanup()
{
    //recurse over contents and delete files
    $files = new RecursiveIteratorIterator(
        new RecursiveDirectoryIterator($this->workFolder),
        RecursiveIteratorIterator::CHILD_FIRST);

    foreach($files as $name => $fileObject){
        if (is_file($name)) {
            unlink($name);
        } else if (is_dir($name)) {
            rmdir($name);
        }
    }

    rmdir($this->workFolder);
}

//cleanup the temp folder on object destruction
function __destruct() {
    $this->cleanup();
}
```

- You again use the `RecursiveIteratorIterator` class to gather all the file names inside the temp folder. (Yes, I am asking the same thing as you, namely: "Who comes up with these kinds of class names?")
- You loop over all the file names stored in `$files`. If the given item is a file, you call `unlink()`; if it's a folder you call `rmdir()`.
- After all files inside the folder are removed, you call `rmdir()` on the temp folder you created in the constructor.

If all goes well, the `cleanup()` method should remove everything the class copies into the system temp location.

You also declare a simple class destructor whose only task is to call the `cleanup()` method. This way you don't need to call `cleanup()` yourself, plus you make sure you will never forget to remove the temp files.

So far so good – these are great plans for how the class will work, so let's see it in action!

Switch to the **getpass.php** file and at the end, just before the closing PHP tag, add the code to create a new `Pass` instance:

```
//create new pass instance
require_once("Pass.php");
$coupon = new Pass("pass/source");
```

OK, time to test the webpage! Open up the form, enter valid data and submit it. You should see a white empty page – success! Or is it? It's difficult to say right now, so let's add some debug output to check whether the code so far works as advertised.

Add these two lines at the end of **getpass.php**:

```
header("Content-type: text/plain");
print_r($coupon->content);
```

Refresh the page in the web browser (it will ask you whether you want to resubmit the form data, confirm you want this to happen), and this time you should see the contents of the pass:

```

Array
(
    [formatVersion] => 1
    [passTypeIdentifier] => pass.com.yourdomain.freehug
    [serialNumber] => <INSERT SERIAL NUMBER>
    [teamIdentifier] => ABC1230000
    [barcode] => Array
        (
            [message] => All you need is love
            [format] => PKBarcodeFormatPDF417
            [messageEncoding] => iso-8859-1
        )
    [locations] => Array
        (
        )
    [organizationName] => Free Hugs LLC
    [description] => Coupon for 1 Free Hug
    [logoText] => Free Hugs LLC
    [foregroundColor] => rgb(9, 118, 203)
    [backgroundColor] => rgb(135, 129, 189)
    [labelColor] => rgb(69, 20, 20)
    [coupon] => Array
        (
            [primaryFields] => Array
                (
                    [0] => Array
                        (
                            [key] => offer
                            [label] => for you
                            [value] => Free hug!
                        )
                )
        )
)

```

The Pass object is ready for the changes you want to make per user. You can access `$coupon->content` and adjust it any way you like.

Before the lines you just added, do add a new bit of code that fills in the dynamic pass data:

```

//fill in dynamic data
$coupon->content['serialNumber'] = (string)uniqid();
$coupon->content['coupon'][ 'secondaryFields'][0][ 'value'] =
(string) $_POST[ 'name'];

```

On the first line, you store in the top-level key 'serialNumber' a freshly-generated unique ID. For the coupon example, the serial number won't be of any importance, so you just make sure it's a unique identifier.

On the second line, you drill down the pass.json's contents to the first secondary field, and you set the visitor's name as the value. Now the coupon is personalized – when opened in Passbook, it will show the name of the bearer on the front side! How awesome is that? Very awesome. ☺

Tip: In this step you learned how to dynamically customize passes. Think for a moment about the possibilities – you can dynamically adjust all the information stored in the pass! You can include the owner's photo dynamically, you can change the information on the front and back, and you can change the way the information is displayed... You can change text, images, colors, and more!

Let's move on to the next step. After you've adjusted the contents according to your needs, it's time you saved pass.json back to the disc. Add this simple method at the end of **Pass.php**:

```
//export a json file from the object
function writePassJSONFile()
{
    file_put_contents($this->workFolder."/pass.json",
                      json_encode($this->content));
}
```

The method is pretty basic: you pass a filename and JSON-encoded text to `file_put_contents()` and that's all. The file is overwritten with the modified data.

Now you can switch back to **getpass.php** and again before the `header(...)` line add:

```
$coupon->writePassJSONFile();
```

Once again, refresh the page in your web browser. Now you can see the dynamically modified contents of the pass: a random serial number and the name you entered as a value of the first secondary field. Also, this modified pass is saved to `pass.json` file in the background.

Note: In your real life applications, take care to filter the user input – you will have to remove or escape any quotes and new lines, because otherwise these will break the JSON format.

Now for some heavy lifting! It's time to generate the `manifest.json` file.

Open up **Pass.php** and add one more method at the end of the class body:

```
//generate the manifest file
function writeRecursiveManifest()
{
    //create empty manifest
    $manifest = new ArrayObject();

    //recurse over contents and build the manifest
    $files = new RecursiveIteratorIterator(
        new RecursiveDirectoryIterator($this->workFolder),
        RecursiveIteratorIterator::SELF_FIRST);

    foreach($files as $name => $fileObject) {
        if (is_file($name) &&
            substr($fileObject->getFileName(), 0, 1)!=".") {

            $relativeName = str_replace($this->workFolder.
                                         "/" , "" , $name);
        }
    }
}
```

```

    $sha1 = sha1(file_get_contents(
        $fileObject->getRealPath()
    ));
    $manifest[$relativeName] = $sha1;
}

//write the manifest file
file_put_contents($this->workFolder."/manifest.json",
    json_encode($manifest));
}

```

I admit that looks like a lot of code, but hey – considering that you loop over the pass folder contents, build the file list containing SHA1 checksums, and save the result as a JSON file, I think it's actually pretty short! Let's see what's happening step-by-step:

- First you create a new `ArrayObject` called `$manifest` – `ArrayObject` is just a simple dictionary object (think of it as `NSDictionary` without all the bells and whistles).
- Then, using the now-familiar method, you gather the list of all files inside the pass folder and store it in `$files`.
- Next you loop over the file list and process all files whose names do not start with a `"."`
- In `$relativeName` you get the file path to each file, relative to the pass folder (this is the kind of file path you need for `manifest.json`).
- You get the content of each file using your old friend `file_get_contents()` and you pass it to the built-in `sha1()` function to get the file's SHA1 checksum.
- At the end of the loop, you store the file path and the SHA1 checksum in the `$manifest` dictionary.
- Finally, you save the contents of `$manifest` as JSON-encoded data in `manifest.json`.

Pretty easy!

You just need to call this method on your `$coupon` object, after you're finished editing the contents, and you will have your manifest created.

Add at the end of `getpass.php`, add:

```
$coupon->writeRecursiveManifest();
```

To make sure all the code is working properly, have a look at the generated manifest. Switch back to `Pass.php` and at the end of `writeRecursiveManifest()` add a debug print line, like so:

```
print_r($manifest);
```

OK – time for another refresh of the web page. Confirm that you want to resubmit the form, and at the end of all the debug output you will see your freshly-generated manifest:



Sweet! That was actually pretty easy to do. (You might need to scroll down the page to see the manifest printout.)

To keep things clean, go ahead and remove the `print_r($manifest);` line since you now know it's working correctly.

The next step, as you already suspect, is to generate the detached signature – I am sure you are excited about it, so let's do it!

At the end of the **Pass.php** class body, add the method to create the signature:

```

//generate the bundle signature
function writeSignatureWithKeysPathAndPassword($keyPath, $pass)
{
    $keyPath = realpath($keyPath);

    if (!file_exists($keyPath . '/WWDR.pem'))
        die("Save the WWDR certificate as
            $keyPath/WWDR.pem");

    if (!file_exists($keyPath . '/passcertificate.pem'))
        die("Save the pass certificate as
            $keyPath/passcertificate.pem");

    if (!file_exists($keyPath . '/passkey.pem'))
        die("Save the pass certificate key as
            $keyPath/passkey.pem");

    $output = shell_exec("openssl smime -binary -sign".
        " -certfile '$keyPath.'/WWDR.pem'.
        " -signer '$keyPath./passcertificate.pem'.
        " -inkey '$keyPath./passkey.pem'.
        " -in '$this->workFolder./manifest.json'").

```

```
" -out '".$this->workFolder."/signature"'.
" -outform DER -passin pass:'$pass'");
}
```

This method takes two parameters. `$keyPath` is the path to the folder where you have your pass certificate and key files. The second parameter is the password for the key file.

First of all, you make sure `$keyPath` contains an absolute path by running it through the `realpath()` function.

Then you check whether the WWDR.pem, passcertificate.pem and passkey.pem files exist in the `$keyPath` folder. (All these checks are not really necessary right now, but they can save you lots of time when you use this class in future projects.)

At the end of the method body, you call the exact same OpenSSL command you used to execute from within Terminal earlier. It's just that this time, you construct the command dynamically from PHP and you pass it to the `shell_exec()` function.

Note: It is possible to have the `shell_exec()` function disabled on some shared hosting providers. You can call your hosting provider and ask them whether that's the case. I personally checked three different hosting accounts of mine on different popular shared hosting services, and they all had `shell_exec()` enabled, so I believe this should not be a problem.

There are other ways to create the detached signature file besides calling OpenSSL directly. Just for your information, I'll summarize them below.

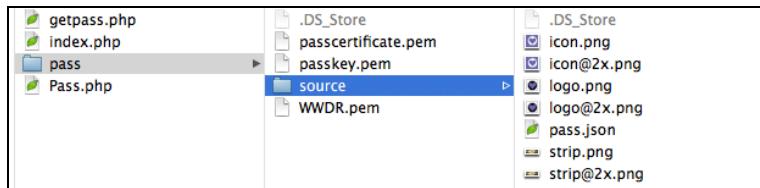
One way is to use the built-in OpenSSL module of PHP. This sounds like a great idea, but turns out not to be in practice. Numerous people have had problems using the built-in solution, and in many cases broken or invalid signatures have been reported.

You can also use a pure PHP encryption library. I tried two libraries that I found on the Internet, but I couldn't generate a valid signature with either of them. (Or didn't have the patience to try hard enough.)

So, in the end, I highly recommend that you invoke OpenSSL through `shell_exec()` and generate your signature that way.

Well, the code to get the manifest signature wasn't complicated, but you still need the certificate and the key, so let's copy them in a convenient place.

Find the WWDR.pem, passcertificate.pem and passkey.pem files you used earlier for the Free Hug coupon. For your example web site, copy them inside the pass folder inside your web server. Don't worry, the contents of that folder are protected from direct downloads via an .htaccess file.



Note: In the “pass” folder there’s a hidden file called “.htaccess”, containing the following Apache directive : deny from all, which denies the access to the folder’s contents. Usually you can’t see this file in Finder.

Advanced text editors like Coda will show the hidden files in their Project navigator, but if you don't use one like these you can still check out the contents of ".htaccess". Download TinkerTool:

<http://www.bresink.com/osx/TinkerTool.html>, fire it up and check the first checkbox “Show hidden and system files” then click “Relaunch Finder”.

Now that you have the .pem files in place, switch back to **getpass.php** and add at the end of the script:

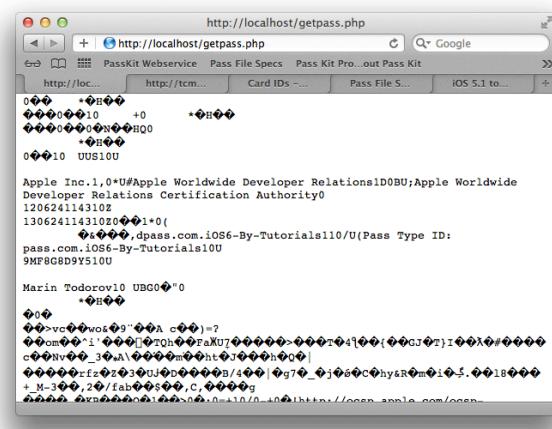
```
$coupon->writeSignatureWithKeysPathAndPassword("pass", '12345');
```

Let's quickly check, just as for the manifest file, if the signature is properly generated.

Switch back to **Pass.php** and at the end of
writeSignatureWithKeysPathAndPassword() add:

```
print(file_get_contents($this->workFolder . "/signature"));
```

Now resubmit the web form and have a look at the debug output. Aha! There's the signature file's content:



I can't really say myself whether the signature is valid or not, but OpenSSL has certainly generated *something!* That should do for the moment as validation, so go back and remove the print line you just added.

You are almost there – if you remember, there's just one more step to completing the pass: zipping up the files in a nice .pkpass package. Fortunately, PHP features an easy-to-use ZIP class called `zipArchive` that will do the job.

At the end of the `Pass` class body add this awesome method that will zip up all the files and save the .pkpass file:

```
//create the zip bundle from the pass files
function writePassBundle()
{
    //1 generate the name for the .pkpass file
    $passFile = $this->workFolder."/".$this->ID.".pkpass";

    //2 create Zip class instance
    $zip = new ZipArchive();
    $success = $zip->open($passFile, ZIPARCHIVE::OVERWRITE);
    if ($success!==TRUE) die("Can't create file $passFile");

    //3 recurse over contents and build the list
    $files = new RecursiveIteratorIterator(
        new RecursiveDirectoryIterator($this->workFolder),
        RecursiveIteratorIterator::SELF_FIRST);

    //4 add files to the archive
    foreach($files as $name => $fileObject){
        if (is_file($name) &&
            substr($fileObject->getFileName(), 0, 1)!=".") {

            $relativeName = str_replace($this->workFolder."/",
                                         "", $name);
            $zip->addFile($fileObject->getRealPath(), $relativeName);
        }
    }

    //5 close the zip file
    $zip->close();

    //6 save the .pkpass file path and return it too
    $this->passBundleFile = $passFile;
    return $passFile;
}
```

A big chunk of this code should already look painfully familiar. Let us go bit by bit and see what's happening:

1. You create the file path to the .pkpass file inside the temporary folder. As the actual name of the file doesn't really make a difference, you use the same unique ID you used earlier to create the temp folder.
2. You create a new `zipArchive` instance simply called `$zip`. On the next line, you call the `$zip->open()` method, and this creates the file itself in the file system. Again – having one more check if the operation was successful is never too much (unless of course if you are a big fan of blindly debugging a problem the day before releasing your product).
3. Step 3 is barely worth mentioning, as you've created recursive directory iterators a million times this chapter already, so you are a pro at this.
4. There's also a lot of familiar code here: you loop over the file list and add files to the archive by using `$zip->addFile`. The two parameters you supply are the full file path to the file to be included in the archive, and the relative path that the file should have *within* the archive (i.e., after the .pkpass is unarchived).
5. Close and save the archive file to disc.
6. Finally, you store the file path to the .pkpass file in `$this->passBundleFile` and return it as the result of the function.

Wow! That was a long trip, but in the end the satisfaction is great! Your class now dynamically generates a pass, signs it and also creates the final .pkpass file! And the process is broken up nicely into small steps, so you can have greater control over the workflow.

Now of course you need to call your final method in **getpass.php**. Add this line at the end of the file:

```
$fileName = $coupon->writePassBundle();
```

Congratulations! You've made it! But wait... how would the user import the pass into their Passbook? Since they fill in their email address in the form, let's just send them an email message with an attachment!

Ah! Sending email attachments with PHP – always a barrelful of fun! (Insert a note of bitterness before the exclamation mark.) PHP has no native support for sending attachments, so you need a small helper function.

Create a new file called **lib.php** on the web server's root folder and paste in this function:

```
<?php

function sendFileToEmailWithTitleAndMessage($to, $subject,
$message, $from, $file)
{
```

```

$baseName = basename($file);
$base64Content =
chunk_split(base64_encode(file_get_contents($file)));
$boundary = md5(time());

$headers = "From: ".$from."\n"
    . "MIME-Version: 1.0\n"
    . "Content-Type: multipart/mixed;
boundary=\"".$boundary."\";

$content = ""
    ."This is a multi-part message in MIME format.\r\n"
    ."--".$boundary."\r\n"
    ."Content-type:text/plain; charset=iso-8859-1\r\n"
    ."Content-Transfer-Encoding: 7bit\r\n\r\n"
    .$message."\r\n\r\n"
    ."--".$boundary."\r\n"
    ."Content-Type: application/octet-stream;
name=\"".$baseName."\r\n"
    ."Content-Transfer-Encoding: base64\r\n"
    ."Content-Disposition: attachment;
filename=\"".$baseName."\r\n"
    .$base64Content."\r\n\r\n"
    ."--".$boundary.--";

    return mail($to, $subject, $content, $headers);
}
?>
```

If you have trouble copying or pasting the code (wrecked new lines, etc.), have a look in the resources folder for this chapter – you can just copy the lib.php file from there.

This method generates a multipart email message and includes the base64-encoded contents of the pass file. If you are interested in the mechanics, read carefully through the source, otherwise just assume it'll all work fine.

The final step of your **getpass.php** script is to send out the pass file. Add at the end of the script:

```

//send over the pass file
require_once("lib.php");

$success = sendFileToEmailWithTitleAndMessage(
    $_POST['email'],
    "Your coupon has arrived",
```

```
$_POST['name'] . ", Thank you for participating!",  
"noreply@yourdomain.com",  
$fileName);
```

When you have a function like this one with a number of parameters, you really start to appreciate the Objective-C style messaging, right?

Let's go over the parameters:

1. The email address of the receiver of the message.
2. The title of the email message.
3. The body of the email message. If the user's name is John, the message will simply be, "John, Thank you for participating!"
4. The email address the message will be sent from. Note you should use an existing email address on your server. Using non-existing senders will likely land your server's IP in spam lists, and will cause you much unnecessary trouble.
5. The last parameter is the file path of the file you want to attach to the message.

Note: I need to warn you that this is a very simple helper function – it will most probably only handle latin1 encoded texts properly. If in your web application you'd like to use Cyrillic, French or Turkish texts, HTML and other advanced features, consider using a more complete library for sending emails, such as PHPMailer:

<http://code.google.com/a/apache-extras.org/p/phpmailer/wiki/UsefulTutorial>

You're almost done! A little bit of cleanup and you'll be crossing the finish line. ☺

Find these two lines in **getpass.php** and delete them (you don't need this debug output any longer):

```
header("Content-type: text/plain");  
print_r($coupon->content);
```

Also add this line at the end of the file to get the user back to the page where they came from:

```
//show thank you message  
header("Location: index.php?message=Thank you for participating in  
our bonus program. <br/>Your coupon has been sent to  
".$_POST['email']);
```

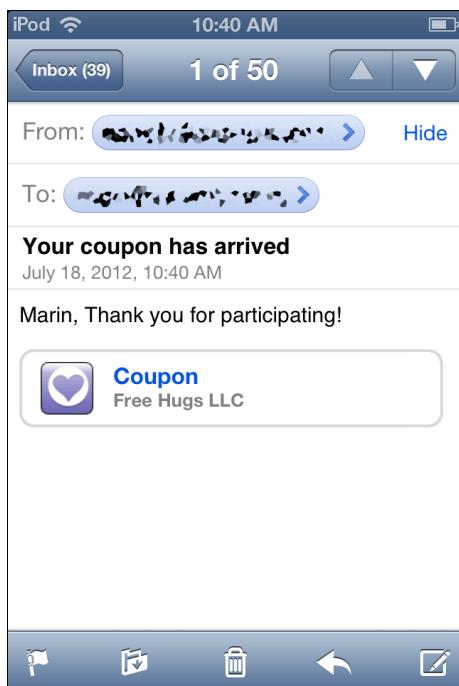
This will redirect the user's browser to index.php and will show a thank you message just above the form.

Phew! That's about it.

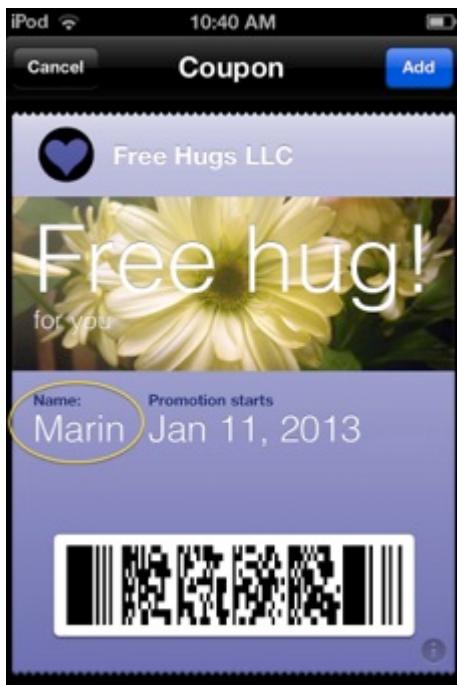
Note: If you've been working on your local Mac machine so far, please note that you won't be able to send email messages from it. To test this final step of developing the website, you should upload the PHP files you created (including the pass sub-folder) to a web server that also has an SMTP server for sending mail.

So, open up your server's URL and fill in the form as you did before, submit the data, and then check your email account.

If all went well, you should see a new message with a Free Hug coupon attached:



Tap the attachment and check if it states your name on the front of the pass:



Awesome!

You successfully developed the promotional website for Free Hugs LLC! You go to their office and present it proudly; there's a round of applause. Then there's a round of hugs. All is well!

However, management has decided to extend the project further. They ask you to add two important features:

1. There are a lot of foreigners in town, and they would really appreciate it if their passes displayed the information in the user's own language.
2. Since the website visitors already provide information about their favorite Free Hugs location in town, it would be nice if the pass would pop up automatically on the lock screen of the device when the user is at their favorite store.

Not a problem! Let's get going!

The right pass at the right location!

First you'll add location awareness to the pass. Each pass can include up to ten locations in its pass.json file, so when the user wakes up the iPhone at one of those locations, the pass pops up on the lock screen.

For the Free Hug coupon, you could easily include all four available locations, but in order to demonstrate more dynamic pass building you'll include only the user's favorite store. (Also, by having fewer locations to check against, you're helping save battery life).

Open up **getpass.php** and find the line `$coupon = new Pass("pass/source");`. Just before this line add an array of store locations:

```
//predefine store locations
$locations = array(
    array("latitude"=>52.402419, "longitude"=>12.970734),
    array("latitude"=>52.491231, "longitude"=>13.430362),
    array("latitude"=>52.481921, "longitude"=>13.433458),
    array("latitude"=>52.481967, "longitude"=>13.431993,
        "altitude"=>30.0)
);
```

Note that the latitude and longitude are numeric values – there are no quotes around them. Also have a look at the last two items – they seem very close to each other (when you look at just latitude and longitude), but the second one also has an “altitude” key. This is because these two locations are both inside the *Big City Mall* in town – one of them is on the 1st floor and the second is up on the 4th floor.

To add the user’s favorite location to the generated pass, open the pass.json file and have a look at the top part of the code. You can spot the “locations” key with an empty array as the value. You’ll add the user’s location of choice to this empty array at run-time.

Switch back to **getpass.php** and just before `$coupon->writePassJSONFile()` add this code to insert a location in the pass:

```
$coupon->content['locations'][0] =
    $locations[(int)$_POST['location']];
```

The user selects a location from the drop box in the form of a store name, and when the form is submitted it sends back the index of the store in the list. You just grab the relevant location value from the `$locations` array using that index.

That’s all, folks! Now the pass also includes location information!

In order to test locations, you can replace the latitude and longitude of one of the stores with your own location. To get the coordinates of the address where you currently are, use this simple and super-fast website: <http://www.getlatlon.com>

After you import the new pass, simply put your device to sleep and press the Home button to wake it back up. In a moment, the pass notification should appear on the lock screen:



If you swipe the “Free hugs LLC” notification to the right, it will open in PassBook immediately.

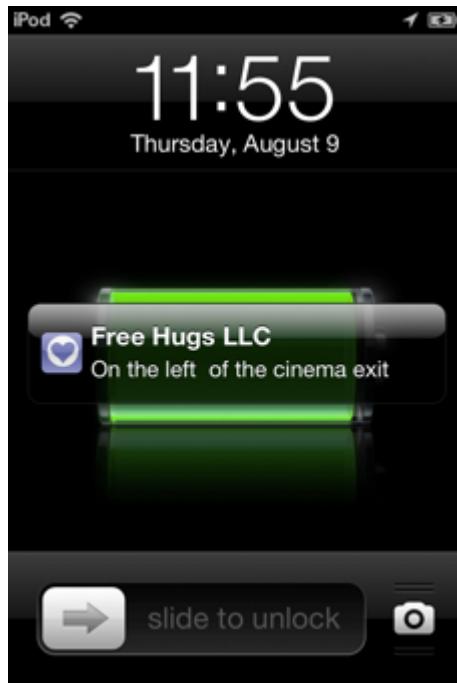
Having Passbook letting know the user that a pass locations is “Nearby” is pretty nifty, right?

Imagine this case: The Free Hugs store is in a big mall. The user is presented with the notification, but “Nearby” doesn’t tell them much about how to reach the store! No fear! You can easily provide more meaningful information for that lock screen notification.

Go back to the code, where you defined the list of locations, and modify the one location with your own coordinates to include also one extra key called `relevantText` like so:

```
//predefine store locations
$locations = array(
    array("latitude"=><your latitude>, "longitude"=><your longitude>,
        "relevantText"=>"On the left of the cinema exit"),
    ...
);
```

Now the user has a pointer where to look for the store!



Localized content

Now for adding localization files to the pass. Pass localization more or less follows the rules of localizing iOS or OS X apps. You create subfolders in the pass directory called en.lproj, it.lproj, etc. to create a localized version for a given language. Inside each of these folders, you create a file called pass.strings and define your translations.

Now you finally understand why you needed all the `Pass` class methods to be directory recursive, right?

Let's add an Italian translation. In your pass/source folder, create a new subfolder called `it.lproj`. Inside the folder, create a file called `pass.strings` and paste in the following definitions:

```
"for you" = "per te";
"Free hug!" = "Free Hug";
"Name:" = "Nome:";
"Promotion starts" = "La promozione inizia";
```

Also, at the time of writing this chapter you need to include an English localization as well, or else the Italian strings will show up even if you're in English. So in your pass/source folder, create a new subfolder called `en.lproj`. Inside the folder, create a file called `pass.strings` and paste in the following definitions:

```
"for you" = "for you";
```

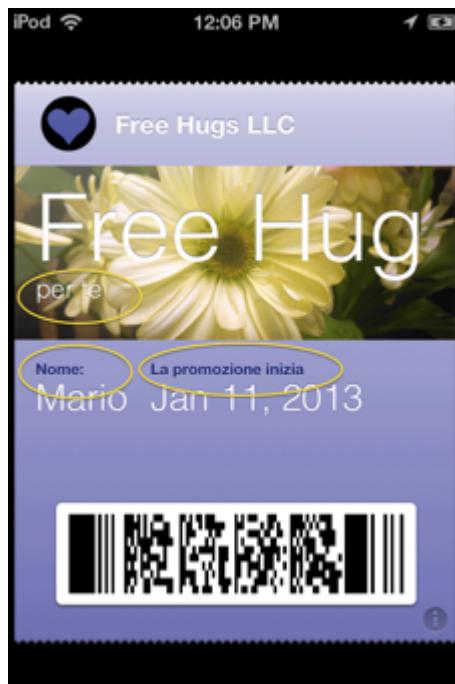
```
"Free hug!" = "Free hug!";
"Name:" = "Nome:";
"Promotion starts" = "Promozione inizia";
```

Believe it or not, that's all you need to do! You now have Italian (and English) translations for the front of your pass.

You can go ahead and update your web server's files, then fill in the form again and get your new localized pass in your mail.

Open up Passbook with your device set to English language and have a look at the latest version of the Free Hug coupon. It looks the same as before – exactly as it should.

Now set your device to Italian language by going to the Settings app, selecting General/International/Language and from the languages list and choosing Italiano. Tap Done to apply the changes. Go back to the Home screen and launch Passbook. The Free Hug coupon now looks like this:



Success!

If you want to provide additional localized content, you can also include customized versions of background.png, background@2x.png, thumbnail.png, etc. in the localization folder.

Note: To switch back from Italian to English, open the Impostazioni app, and choose Generali/Internazionale/Lingua, and from the list choose English. Tap

Fine to apply the changes. Then take a vacation to Italy to celebrate your new Italian skills! ☺

This time, the owners of Free Hugs LLC are more than satisfied and give you a big bonus. They run a successful promotion on the web and give away tons of free hugs to people using iPhone passes!

Where to go from here?

Do you remember back when you were first starting on this chapter? You didn't have any idea of the wonderful and powerful technology behind passes. But now you are a real pro, with a deep understanding of how to create passes, how they are distributed, imported, and most importantly, generated on-the-fly.

You now have a very valuable new skillset – and instead of the superficial understanding you might get from quickly scoping some incomplete examples on the web, now you **really** know and understand iOS 6 passes.

In addition, in this chapter you've developed a couple of tools that you can reuse in your future projects and extend to suit your needs.

When you start designing and developing your own passes, you will definitely appreciate the PassPreview app and the way it speeds up the process of testing passes.

You can also extend the Free Hugs LLC promo website to:

- Generate other types of passes. Try including a photo provided by the user on an event ticket. Or, instead of a coupon, send them a store card with preloaded store credit.
- Try to alter the colors and the layout of the coupon. The more you know about how to tweak the pass, the better prepared you are for real-life pass-design.
- Add a field on the back of the pass with traffic directions to the user's favorite store (depending on the store location chosen by the user).
- Add the user's favorite store address to the back of the pass, so they can easily open it up in Maps.

You covered tons of theory and you also applied it in practice, but... there's more!

Believe it or not, there's much more behind generating and maintaining passes on iOS! The next chapter will dive straight into the really cool stuff that makes cutting edge passes. You will:

- Develop a server-based application for maintaining your pass database.
- Learn how to update users' passes with current information.

- Update already-installed passes with current information coming in real-time from your server!
- Create a web service on your server so that Apple can talk to you about the passes you create and distribute.
- Create a barcode-scanning app that can read your pass information.
- Make your passes date- and time-aware.
- Put all these together into an awesome end-to-end solution.

I'll see you in the next chapter!

Chapter 8: Intermediate Passbook

By Marin Todorov

After successfully making it through the last chapter, you are already an iOS Pass Pro. You know how to create new pass types, how to generate passes on-the-fly on your web server, and of course, how to distribute them via email.

But as I promised at the end of the last chapter, there's still a lot more to learn!

This chapter will focus on how to keep track of passes sent to users and update them over time. Up to now, you created passes and sent them out into the wild, but you didn't really "stay in touch." In this chapter, you'll keep track of passes using a database on your server, and when necessary you will update the passes dynamically.

Just imagine the possibilities! Here are a few examples of how iOS passes are better and smarter than their paper equivalents:

- **When a ticket's boarding-platform changes** – it's no problem! You can update users' train tickets remotely from your server, and notify them of the change.
- **A concert is postponed** – also not a problem! You change the date on the ticket, and notify the fans.
- Even **if a store has relocated** – you can just update the address on the back of the store cards, modify the driving directions, and you're done.

Besides implementing a two-way communication between your server and your customers' devices, you are also going to cover few Passbook aspects that didn't fit in the previous chapter, like:

- Making your passes date- and time-relevant.
- Distributing your passes as a direct file download in Safari.
- Implementing a complementary app to read your pass information.

I'm sure all of this sounds exciting, so let's dive in!

Prerequisites for this chapter

Before you start implementing the project discussed in this chapter, I need to mention a few things. Developing an end-to-end system for distribution and management of passes is an intermediate-to-advanced task (as the chapter title suggests), so I will assume you have a good understanding of the following:

- First of all to successfully follow through this chapter you will need to test on a device (not the iPhone Simulator) as some of PassKit's functionality is available only on an iPhone or an iPod Touch.
- Setting up a website, either on your local computer or on a hosted domain. You will need an Apache web server, as you are going to use the Apache `mod_rewrite` module to handle dynamic URLs.
- If you're working on your local machine, you will need to be able to access the website running on it from your iOS 6 device (usually through your local WiFi connection).
- If using a hosted server, one of those all-in-one, no-limits hosting packages probably won't cut the cheese. Look below for the server requirements.
- You need at least a basic-level of understanding of HTML; you are going to develop simple pages and I won't go in to detail about what HTML is and how to use it.

I know the prerequisites are high, but you are looking at developing a complex service with the hottest, latest technology out there from Apple, so some preexisting knowledge is required.

Server requirements

On your web server, you will need:

1. Apache web server with PHP 5.2+ installed.
2. PHP modules installed – ZipArchive.
3. `mod_rewrite` Apache module installed (and access to rewrite rules via `.htaccess`).
4. You will need to create socket connections to port 2195 on a remote server from PHP (this is where a number of shared hosting packages will fail you).
5. A MySQL database.

These are the only requirements. If you can, in general, import some SQL and update files on your website, everything else will be explained step-by-step!

The big Easter Egg Hunt

In this chapter you are going to develop an end-to-end software solution for a hypothetical client. The client is the Big City Mall in town and they are planning an

awesome promotional game for Easter. They decided to call it the “Easter Egg Hunt.”

The project specification requires you to develop several separate small projects and connect them to work together. Namely you will, by the end of this chapter, create:

1. A website where the Mall visitors can download Easter Egg Hunt iOS passes.
2. A web admin panel where the Mall animators can start the game and control it in real time.
3. An iOS app to scan passes and communicate with a web service via JSON.
4. A PassKit web service to communicate with the devices that have your passes installed, and also directly with Apple.

These elements will all work together and communicate with each other in different ways – in other words, this is going to be very educational and a ton of fun!

To illustrate the goals of the project, and how the Easter Egg Hunt game works, the creative agency behind it has come up with the following comic:



(Artwork: Nikolay Petkov, www.nikolaypetkov.com)

And here is a detailed description to go along with the comic:

1. In preparation for the Easter Egg Hunt promotion, mall visitors can download iOS store card passes with \$0.00 value from a website. These passes are also stored on the server's database.

2. On Easter Sunday, the Easter Bunny starts the game and transmits information about its location inside the Mall (for example "I am hiding somewhere in the Ice-cream Princess store!").
3. All the passes receive an update with the latest location of the Easter Bunny, so people can search for it.
4. The first person who finds the Easter Bunny has their pass scanned by the Bunny, and the pass receives a \$25 increase on its current store credit amount.
5. The Easter Bunny then hides in another store, and the next person who finds it receives a \$25 increase on their pass. This repeats 10 times and the bunny gives away \$250 total in store credit.
6. After the last bonus is distributed, the server marks the game as finished.
7. After the game has finished, everyone can use the store credit they earned during the game to buy goods in the Mall, using, of course, their passes.

Sounds interesting? Then let's get started!

Building the pass distribution website

Getting help from an old friend

Let's start by building upon material you are already familiar with. In this part of the chapter you will develop a website that will provide the visitors with \$0.00 value store cards. The task is somewhat similar to the website project you already developed in the previous chapter. In fact, you are going to use the `Pass` PHP class you created there.

In your web server's root folder, create a new subfolder called **egghunt**. Inside it you will place all the different parts of the Easter Egg Hunt project. (The code you are going to write later on will assume you did create the egghunt folder in your root folder.)

Next, inside the "egghunt" folder, create another folder called **class**. You will put all your shared PHP classes in there. In fact, in the last chapter you created a very well designed class for creating passes called **Pass.php**, so you are going to use it almost unmodified for this new project.

So go ahead and copy the **Pass.php** that you developed in the previous chapter into this folder. If you do not have this file, you can find it in the resources for this previous chapter inside **webfiles.zip**.

Let's add one new feature to the `Pass` class and then you'll be done making changes to this file. On the new website, you are going to provide passes as file downloads directly via the browser, rather than having to ask for the visitor's email address and delivering the passes via email.

At the very bottom of **Pass.php** (but still inside the class body, i.e., before the final closing curly bracket) add this method:

```
function outputPassBundleAsWebDownload() {
    //dump the generated pass to the browser
    header("Content-Type: application/vnd.apple.pkpass");
    header("Content-Disposition: attachment; ".
        "filename=".basename($this->passBundleFile));
    header("Content-Transfer-Encoding: binary");
    header("Content-Length: ". filesize($this->passBundleFile));
    flush();
    readfile($this->passBundleFile);
}
```

The new method, as its name suggests, reads the content of the last generated .pkpass file and dumps into the web browser. To instruct the browser that you are providing a file for download, you first send several HTTP headers. Here's what they do:

- **Content-type:** this is the most important one. It informs the browser of the MIME type of the file being served. When Safari encounters a file download with the content type of "application/vnd.apple.pkpass", it knows to expect a pass file from the server, and when the file is downloaded, it tries to open and display the file inside Passbook.
- **Content-Disposition:** this header tells the browser the default name for saving the file.
- **Content-Transfer-Encoding:** instructs the browser that the content will be in binary format.
- **Content-Length:** provides the size of the file to the browser.

After all headers are set, you call `flush()` to send them over to the browser, and then you output the content of the .pkpass file. `readfile()` is a special binary safe function, made exactly for the purpose – it takes in a file name, reads the content of the file, and outputs it to the browser.

Now you're done with the `Pass` class. Sweet!

Creating a custom Easter Egg Hunt pass class

Up until now, you've worked with the rather abstract `Pass` class, and it's been good! The implementation is not too specific so you can be very flexible about what passes you generate.

Now you are going to build upon the code you have so far and create a specialized PHP pass class that will be a concrete pass implementation, specifically serving the purpose of the current project.

Inside the “class” folder, create a new file and call it **EggHuntPass.php**. Paste in the basic code for the new class:

```
<?php

require_once("Pass.php");

//EggHuntPass is a more concrete
//pass class implementation
class EggHuntPass extends Pass
{

    // 1) the pass configuration is
// bundled in the class
    private $keyPath = "pass";
    private $sourcePath = "pass/source";
    private $keyPassword = "12345";

    static $passTypeID = "pass.com.yourdomain.couponfreehug";

    // 2) autoconfigures the object and
// calls the original Pass constructor
    function __construct() {
        $this->keyPath = realpath(dirname(__FILE__)).  

            "/../../$this->keyPath";
        $this->sourcePath = realpath(dirname(__FILE__)).  

            "/../../$this->sourcePath";

        parent::__construct($this->sourcePath);
    }
}
```

Note: A quick note on PHP syntax. If your file contains only PHP code, you need to have the PHP opening tag `<?php` on top, but you don't need to use the closing tag. When the interpreter reaches the end of the file, it knows that the script is over.

But of course, if you want, you can add the closing `>>` tag. ☺

Let's have a look at the code, shall we?

Since this class facilitates the creation of only one specific type of pass, you bundle its entire configuration within the class body. You don't need to supply the key path

and key password to the constructor anymore – they are the same for all possible uses of this concrete class.

You declare four class members containing the entire configuration the parent class needs:

1. **\$keyPath** – the path to the folder where your certificate and key files are located (relative to the root of the project, i.e., to the egghunt folder). You will create this folder soon.
2. **\$sourcePath** – as before, this is the relative path to the source files for your passes, which you will also create soon.
3. **\$keyPassword** – the password for your key file. You exported your key in the last chapter using **12345**. If you used a different password, put that value here. Without the right password, the pass will not work!
4. **\$passTypeID** – this is, of course, the pass type, i.e., "pass.com.yourdomain.couponfreehug". You should change this to match the value you entered for your own pass type ID.

You don't need the class constructor from the parent class anymore. So you provide a new constructor, which takes no arguments – the class already knows where to find the pass source files.

Let's have a detailed look at what happens on this line:

```
$this->keyPath = realpath(dirname(__FILE__)).  
    "/.../".$this->keyPath);
```

`__FILE__` is a special PHP constant, which holds the absolute path to the file currently being interpreted. The `dirname()` function returns the parent folder path when given a path to a file, so now you have the full path to the class folder. Next, you append to this path the relative path to the certificate and key folder.

Then `realpath()` converts the path again to an absolute path. Now you have an absolute path in `$keyPath`, instead of the relative path to the desired folder.

The next line does exactly the same for the source folder.

Finally, you call the parent class constructor and provide it with the pass source files' path. Sweet!

If you think about it, you can now automate the pass generation process as well, because the configuration is already there!

Let's do it. Add this new method inside the class body (i.e., before the final closing curly bracket):

```
//calls all Pass methods one after another  
//because all the configuration is bundled with the class  
function writeAllFiles()
```

```
{  
    $this->writePassJSONFile();  
    $this->writeRecursiveManifest();  
    $this->writeSignatureWithKeysPathAndPassword(  
        $this->keyPath, $this->keyPassword);  
    $this->writePassBundle();  
}
```

This is exactly the same sequence of method calls you had in your `getpass.php` file in the previous chapter. You know pretty well what each of the method calls do. The new method `writeAllFiles()` saves the `pass.json` file of the pass, generates the manifest, and prepares the final pass `.pkpass` bundle in one shot.

Remember, designing a good abstract base class pays out big time when you then develop further concrete implementations.

Now that you've made good progress on the PHP code, it's time to start working on the HTML pages for the website.

Get me a pass to the game!

Inside the **egghunt** folder, create a new file and call it **index.php**. This will be a simple page with a bit of information about the game, and a button to download a new game pass.

Paste the following into **index.php**:

```
<html>  
<head>  
    <title>Big Easter Egg Hunt!</title>  
    <meta name="viewport"  
          content="initial-scale = 0.75,maximum-scale = 0.75" />  
    <style>  
        form {border: 1px solid #611111; width: 380px; padding: 10px;  
              background: #ffcccc; border-radius: 15px;}  
        h1 {font:italic bold 20px/30px Georgia, serif;}  
        li {padding-top: 8px;}  
    </style>  
</head>  
<body>  
  
<form action="geteasterpass.php">  
    <p>Come this Easter to the Big City Mall  
        and participate in the Easter Egg Hunt!</p>  
    <ul>  
        <li>Download your own unique store card now</li>  
        <li>On Easter Sunday you will receive a Passbook
```

```
notification on your iPhone</li>
<li>Whoever first finds the Easter Bunny receives a $25
credit on their pass</li>
<li>The Easter Bunny will appear in 10 different stores in
the mall, so keep looking for it</li>
<li>When the hunt is over you can spend your gained credit
in the mall by showing your Easter Hunt card at the cash
desk</li>
</ul>

<input type="submit" value="Get an Easter Egg Hunt Pass"/>

</form>
</body>
</html>
```

The page contains only basic HTML, but in order not to leave anyone behind, here comes a short walkthrough:

- The `<title>` tag specifies the page title, which is usually displayed at the top of the browser.
- The `<meta name="viewport" ...>` tag specifies the scale of the page. You set it to be slightly scaled down in order to fit the page contents on the iPhone screen.
- Inside the `<style>` tag, there are a few CSS-style declarations to make the page look beautiful - at least to a programmer's eyes ☺
- Next, inside the `<body>` tag you declare a `<form>` tag. This just instructs the browser to go to `geteasterpass.php` when the user presses the submit button.
- Then there's a bulleted list with the game details.
- Finally, at the bottom of the page an `<input type="submit" ...>` tag creates a button that will get the user their pass.

Pretty easy, right?

I won't go into the details of how to publish the website to a server, because you should have already managed it in the previous chapter. Now you should do it again, by copying the whole `egghunt` folder to your server's root web folder.

Tip: You can actually build and test the entire project on your local machine's web server. The only thing to watch out for is that you will need to open the website from your iPhone. If your Mac and your device are on the same WiFi network, you can browse to the computer's IP address to see if you can see the web server in Mobile Safari. It should normally work.

If you aren't familiar with setting up a local web environment, just publish everything to your website and access it from there – but don't forget to update the server files every time you make changes to your local source!

Note: In the coming pages I will refer to a generic site URL; for example: <http://yourdomain.com/egghunt>. **Please replace “yourdomain.com” with your actual server URL** (or the IP address, if you're working locally) every time you need to type in a URL.

So, just to make sure if all your initial set up is working properly, open up <http://yourdomain.com/egghunt/> and check if you can see the promo page in Mobile Safari on your device, like so:



Pretty cool... if you like pink!

As the script to generate the passes is super simple, let's go on and add it to the website as well. Create a new file, call it **geteasterpass.php**, and paste the following code into it:

```
<?php
require_once("class/EggHuntPass.php");

//generate a new pass
$coupon = EggHuntPass::createPassWithUniqueSerialNr($error);

if ($coupon!=null) {
    //send it over to the client
    $coupon->outputPassBundleAsWebDownload();
```

```
    } else {
        //there was an error
        die("Error: ".$error);
    }
```

First you import the new and shiny `EggHuntPass` class.

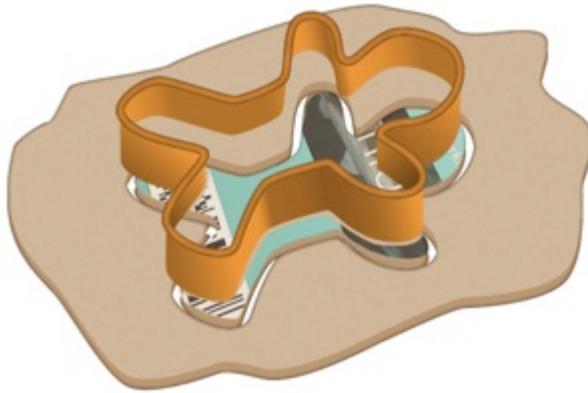
Then you call the factory method `createPassWithUniqueSerialNr()` and it returns an instance of the `EggHuntPass` class.

If `$coupon` is not `null`, it means you've got a valid instance of a pass generated successfully. So you dump it in the browser. (As mentioned previously, that will open the pass in Passbook on the user's device.)

If, on the other hand, `$coupon` is `null`, then you just print the returned error in the browser.

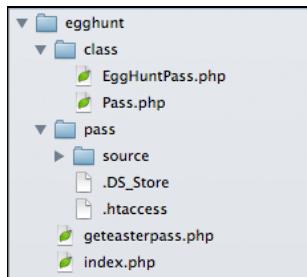
Although... you don't have this magical `createPassWithUniqueSerialNr()` method, do you? Next you are going to add it to `EggHuntPass`.

The Easter Egg Hunt Pass cookie cutter



Let's create a factory method for `EggHuntPass` that will load a pass template and fill it in with data – your very own pass cookie cutter of sorts!

Copy and extract **Resources.zip** found in the resources for this chapter. It should contain a subfolder named **pass** – copy it to the **egghunt** folder on your web server. The pass folder contains a **source** subfolder, similar to your pass source folder from the previous chapter. Your folder structure should now look like this:



Go ahead and open up **pass/source/pass.json**. This is material you've already covered, so you should be feeling pretty comfortable with the source code. Let's have a quick look:

```
{
  "formatVersion" : 1,
  "passTypeIdentifier" : "<INSERT PASS TYPE ID>",
  "serialNumber" : "<INSERT SERIAL NUMBER>",
  "teamIdentifier" : "<HERE YOUR TEAM ID>",

  "webServiceURL" : "http://yourdomain.com/eggplant/ws/",
  "authenticationToken" : "<INSERT TOKEN>",

  "barcode" : {
    "message" : "<INSERT BARCODE>",
    "format" : "PKBarcodeFormatQR",
    "messageEncoding" : "iso-8859-1"
  },
  "locations" : [],
  "organizationName" : "Big City Mall",
  "description" : "Easter Egg Hunt card",
  "logoText" : "Easter Egg Hunt",
  "foregroundColor" : "rgb(255, 255, 255)",
  "backgroundColor" : "rgb(235, 162, 205)",
  "labelColor" : "rgb(45, 54, 129)",
  "generic" : {
    "primaryFields" : [
      {
        "key" : "credit",
        "label" : "StoreCredit",
        "value" : "",
        "changeMessage": "Your Big City Mall credit is now %@"
      }
    ],
    "secondaryFields" : [
      {
        "key" : "nextlocation",
        "label" : "Next egg hunt location",
        "value" : ""
      }
    ]
  }
}
```

```
        "value" : "",
        "changeMessage": ""
    }
],
"backFields" : [
{
    "key" : "terms",
    "label" : "TERMS AND CONDITIONS",
    "value" : "Check the Terms & Conditions of the game on the
web site of Big City Mall"
}
]
```

This is a generic pass and it is pretty simple. It features two fields on the front: the current credit on the pass, and another field to show the next location where the Easter Bunny will be. On the back, there's a simple terms and conditions field.

Since you have the file open, go on and edit the `teamIdentifier` field – just as before, put in your own team ID as the value. Also edit the `passTypeIdentifier` to the pass type you created earlier. (If you don't remember what to put for the `teamIdentifier` or `passTypeIdentifier`, refer to the `pass.json` file from the previous chapter.)

As you can see, there are a few fields with strange values like "INSERT TOKEN" – these fields will be filled in dynamically from your PHP script, but for the time being, they have some placeholder values.

There are also some new JSON keys that weren't covered in the last chapter. These help implement some advanced features and you will learn about them in a few moments.

Do you remember where your keys are?

Well, as the subtitle suggests, I hope you still have your pass certificate and key files in a handy location, because you are going to need them in this chapter again. Copy **WWDR.pem**, **passcertificate.pem** and **passkey.pem** into your new project's **egghunt/pass** folder. If you have lost these files, refer back to the instructions in the previous chapter about how to generate them.

The only thing remaining is to write that `EggHuntPass` factory method. Open up **class/EggHuntPass.php** and add the following method at the end of the class body:

```
//1) create a new game pass with unique serial number
static function createPassWithUniqueSerialNr(&$error)
{
```

```
//2) get new instance
$pass = new EggHuntPass();

//3) fill in the details
$pass->content['passTypeIdentifier'] = self::$passTypeID;
$pass->content['serialNumber'] =
    (string)round(microtime(true)*100);
$pass->content['barcode']['message'] =
    (string)round(microtime(true)*100) .
    mt_rand(10000,99999);
$pass->content['generic'][['primaryFields'][0]['value']] =
    sprintf("$ %.2f", 0);

//4) save the pass bundle
$pass->writeAllFiles();

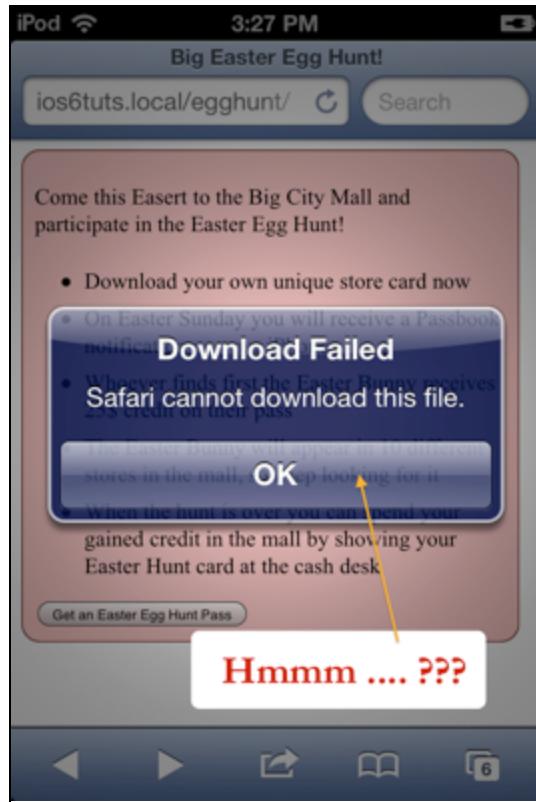
//5) return the pass object
return $pass;
}
```

It looks pretty simple and somewhat familiar, right? Here is it, step-by-step:

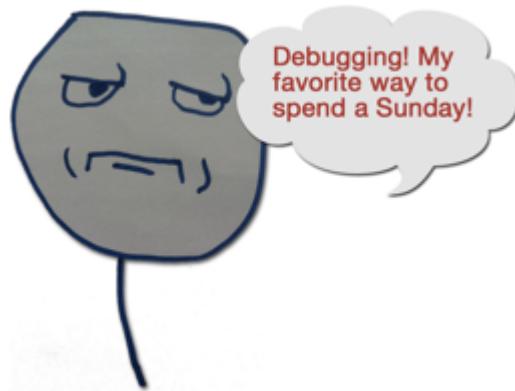
1. The method declaration begins with the keyword **static**, which declares a static class method pretty much like a + does in front of a method declaration in Objective-C. Note also the parameter has an & before the variable name – this denotes that the parameter value is passed as a pointer and not as a value.
2. You use the handy new constructor of `EggHuntPass` to create a new instance.
3. Just as you did before, you fill in some of the pass values dynamically:
 - `passTypeIdentifier` – you have the type ID stored inside the class, so you can just fill it in the JSON as well.
 - `serialNumber` – `microtime()` generates a timestamp, including the microseconds of the current time (microseconds are included as the floating point part of the result). You multiply by a hundred and round it, so you end up with a unique-enough serial number for your pass.
 - `barcode/message` – for the barcode you need another unique value, so you take in the microseconds precision time again and concatenate it with a random number: that should do it!
 - Finally, as for the first primary field, you save a “\$0.00” value – the initial credit amount on the new card.
4. You call the new `writeAllFiles()` method to finish up the pass generation.
5. And you return the new pass instance.

Your pass is ready for download!

Browse to <http://yourdomain.com/egghunt/> on your device and press the button at the bottom (after uploading the modified files to the web server of course). Oops! Instead of the pass popping up, you see this alert:



Something is apparently not working correctly in your scripts! But what *exactly*?



Well, now is the perfect time to learn how to debug passes.

What is wrong with my passes?

Make sure your device is enabled for development (i.e., you can build and run a project straight from Xcode on the device).

Note: Debugging passes can only be done on-device. The debug options discussed below aren't available on the Simulator.

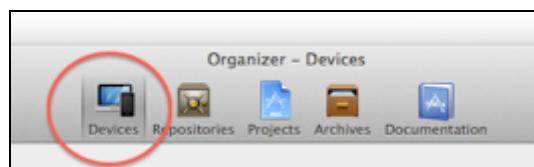
Open the Settings app (on your iPhone or iPod Touch) and find the "Developer" item in the list. Tap it and scroll down until you see these two PassKit-specific options:



Make sure they are both turned on.

1. **Additional Logging** makes PassKit dump detailed information to the device's console, so you can debug more easily.
2. **Allow HTTP Services** – this option allows for a normal non-encrypted connection between your device and your server. For production environments, your server should support HTTPS – we'll talk about why and how later in the chapter.

Now from Xcode's menu, choose Window/Organizer to bring up the Organizer window. Make sure the "Devices" tab is selected on the top toolbar:



Now connect your development device via cable to your computer and wait until it gets a green "connected" light on the left menu, like so:



Now you can choose “Console” from the device’s sub-menu, and this will display the system output console on the right.

The console is probably full of all kinds of system output, so use the “Clear” button at the bottom of the window to clear it. Clearing the console before every test makes it a lot easier to handle all the output, because it tends to be overwhelming when all apps are outputting debug info at the same time.

Tip: If your console is empty, it might be due to a known Xcode glitch: just click on another menu item like “Applications” or “Screenshots,” then switch back to “Console” – you should then see the system output.

Also, try to close all the background apps running on your device in order to minimize the debug output to your console.

So... hit the Clear button, browse to <http://yourdomain.com/egghunt/> again, and press the download button.

Sometimes you need to skim through the output, but you surely will spot these two lines in the Console:

```
Jul 27 17:43:09 MTT-iPod-4 MobileSafari[1473] <Warning>: Invalid  
data error reading card  
pass.com.yourdomain.couponfreehug/134340378884.  
authenticationToken '<INSERT TOKEN>' is too short. It needs to be  
at least 16 characters.  
  
Jul 27 17:43:09 MTT-iPod-4 MobileSafari[1473] <Warning>: PassBook  
Pass download failed: The card cannot be read because it isn't  
valid.
```

Pretty handy! Right?

Since the process of importing and working with passes is not really transparent (a big part of it is happening in the background), Apple included this kind of useful console output on developer devices.

As you can see, PassKit complains that the `authenticationToken` is too short. You still don't know what the `authenticationToken` is all about and why you need it in `pass.json`, but play along for the moment.

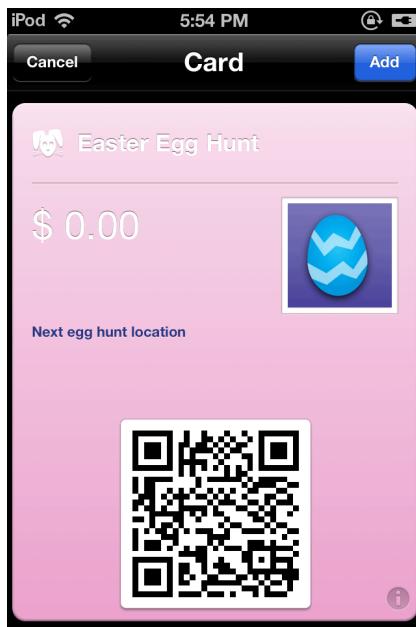
Open **EggHuntPass.php** and inside `createPassWithUniqueSerialNr()`, find the line where you set the pass serial number. Just before this line add the following code to set the `authenticationToken`:

```
$pass->content[ 'authenticationToken' ] =  
    sha1(mt_rand().microtime(true));
```

This code generates a random value, concatenates it with the current timestamp to create more randomness, and then returns the SHA1 checksum of this. That creates a random 40-character `authenticationToken`, which should make PassKit happy for the moment. You'll learn all about the token later on.

Now it's time to try opening the pass again!

Switch back to Safari on your device and tap the download button again. The pass pops up before your excited eyes. ☺



Okay – that's pretty cool. You just learned how to serve passes for download from the browser, and how to debug passes – we're moving right along, footloose and fancy free!

Have another look at the device Console and you'll notice these two lines, which indicate that the pass was successfully opened:

```
Jul 27 17:57:26 MTT-iPod-4 MobileSafari[1473] <Warning>: begin  
ingestion perf testing  
  
Jul 27 17:57:26 MTT-iPod-4 MobileSafari[1473] <Warning>: Ingestion  
Perf Testing: 0.362439
```

You are ready for the next step!

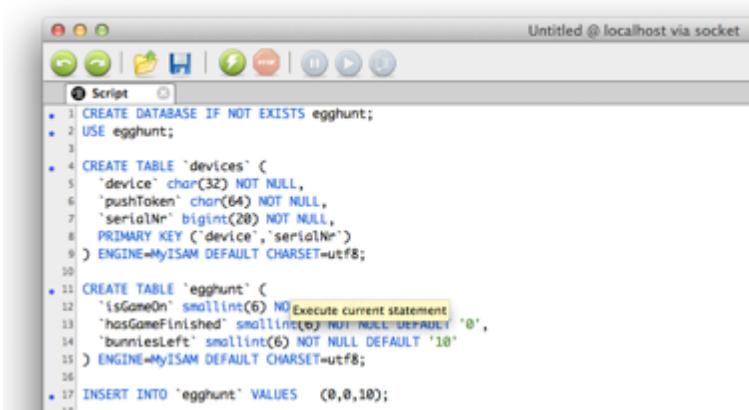
Storing passes on your server

Preparing the passes database

To store and manage passes, you need several tables in your database. You are going to use a MySQL server (the usual companion to PHP web applications), and I already have created an SQL file with the database structure and initial data ready for you.

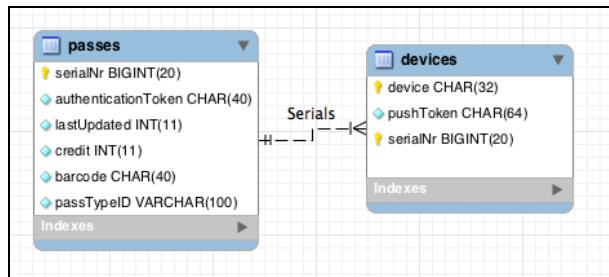
The **Resources.zip** file you extracted earlier should include a file called **egghunt.sql**. Run the SQL script in your favorite MySQL utility to create the egghunt database and the tables you're going to need.

I'm using MySQL Query Browser to connect to my local MySQL server; you, on the other hand, are free to do this task in any way you want, but importing SQL scripts is outside the scope of this chapter, so I won't be covering it.



```
Untitled @ localhost via socket  
Script  
1 CREATE DATABASE IF NOT EXISTS egghunt;  
2 USE egghunt;  
3  
4 CREATE TABLE `devices` (  
5     `device` char(32) NOT NULL,  
6     `pushToken` char(64) NOT NULL,  
7     `serialNr` bigint(20) NOT NULL,  
8     PRIMARY KEY (`device`, serialNr)  
9 ) ENGINE=MyISAM DEFAULT CHARSET=utf8;  
10  
11 CREATE TABLE `egghunt` (  
12     `isGameOn` smallint(6) NO Execute current statement,  
13     `hasGameFinished` smallint(6) NOT NULL DEFAULT '0',  
14     `bunniesLeft` smallint(6) NOT NULL DEFAULT '10'  
15 ) ENGINE=MyISAM DEFAULT CHARSET=utf8;  
16  
17 INSERT INTO `egghunt` VALUES (0,0,10);  
18
```

After importing the SQL you should have four tables in your egghunt database. You need the first two in order to manage the created passes and the devices where each pass is installed (a single pass can be installed on more than one device):



In the `passes` table you store all the dynamically-generated information for each pass:

1. **serialNr** is, of course, the pass serial number.
2. **authenticationToken** is the token that the device sends to your server when it connects and asks you for updates, etc.
3. **lastUpdated** is the timestamp of the last update to the pass.
4. **credit** is the amount of store credit currently on each Easter Egg Hunt pass.
5. **barcode** is the unique barcode string for the pass.
6. **passTypeID** is exactly what it sounds like. In the current project you are going to have only one pass type ID, but having this column in place puts you one step closer to enhancing the system further to handle different pass types.

Next, let's look at the `devices` table:

1. **device** is a unique device ID that PassKit sends over to your server. Note that this has nothing to do with the device UDID. The ID from PassKit is a unique ID generated especially for you to use when sending updates to your passes and maintaining your passes database.
2. **pushToken** is a device push token paired to the device ID. You use this token when you want to send a push notification to a given device.
3. **serialNr** is the same `serialNr` from the `passes` table. It indicates which pass is installed on the given device from the `devices` table (remember: one pass can be installed on more than one device).



The other two tables in the database are related to the game itself, so let's not worry about them for the moment.

Let's connect the script generating Easter Egg Hunt passes to the database!

Storing pass details for future use

Open up **class/EggHuntPass.php** again and consider the factory method for generating new passes. You generate a lot of random data there, and once the pass is generated, there's no way to recall what the serial number of a given pass was or its barcode message.

In this section, you are going to add database connectivity to this class and store the pass data in the *passes* table.

Create a new PHP file in the **class** subfolder of your project and name it **Database.php**.

You need a simple class to connect to your database instance – just a singleton class that opens a new database connection. Paste in the code for this new class:

```
<?php

class Database extends PDO
{

    //database connection details
    static private $host      = "localhost";
    static private $dbname    = "egghunt";
    static private $user      = "username";
    static private $pass      = "password";

    private static $instance = null;

    //get the singleton instance
    static function get() {
        if ($self::$instance!=null) return $self::$instance;

        try {
            $self::$instance = new
Database("mysql:host=".$self::$host.";dbname=".$self::$dbname,
$self::$user, $self::$pass);
            return $self::$instance;
        }
        catch(PDOException $e) {
            print $e->getMessage();
            return null;
        }
    }
}
```

```
}
```

Your class extends the built-in PHP class PDO, which provides database connectivity. What you need is just a static method, which returns you a singleton instance of the database connection, so you do that in the static `get()` method.

Replace the values of the four static class properties with the connection data and credentials for your own MySQL server and database.

The body of the static `get()` method is pretty simple:

- First you check if there is already an instance stored in the static `self::$instance` class property.
- Then, inside the `try` block, you try to create a new `Database` instance (the `PDO` constructor is automatically called when you don't provide your own) and store it in `self::$instance`.
- At the end of the `try` block you return the static instance.
- If an exception occurred, inside the `catch` block you print the error message and return `null`.

Tip: Notice that in PHP, the `$this` keyword always refers to the current instance of the class; `self`, on the other hand, points to the class itself. So, from within a static method of the `Database` class `Database::$host` and `self::$host` point to the same value. (Of course, if you use `self`, then in your class's ancestors the code will still work properly, i.e., accessing the ancestor's static properties.)

So far, so good – the new `Database` class should provide you access to your MySQL database, and you can go on with implementing the logic inside the `EggHuntPass` class.

Open **EggHuntPass.php**. First at the top of the file, just below the `require_once` line, include another `require` function for the `Database` class:

```
require_once("Database.php");
```

Inside the `createPassWithUniqueSerialNr()` method, add this line to get an active connection to the database (add it just before the call to `writeAllFiles()`):

```
//get database connection  
$db = Database::get();
```

OK, you got a database connection (if your connection data is correct). So now save the pass data into the `passes` table:

```
try {
```

```

// 1
$statement = $db->prepare("INSERT INTO passes(serialNr,
authenticationToken, lastUpdated, credit, barcode, passTypeID)
VALUES(?, ?, ?, ?, ?, ?)");
// 2
$statement->execute(array(
    $pass->content['serialNumber'],
    $pass->content['authenticationToken'],
    time(), //last update is now
    0, //initial credit is 0
    $pass->content['barcode']['message'],
    self::$passTypeID
));
// 3
if ($statement->rowCount()!=1)
    throw new PDOException(
        "Could not create a pass in the database");
}
// 4
catch (PDOException $e) {
    $error= $e->getMessage();
    return null;
}

```

In this code, you execute an `INSERT` on the `passes` table, conveniently wrapped in a `try` statement to catch any database exceptions.

Let's go over it line-by-line:

1. The `prepare()` method of the database class creates a prepared statement with the SQL you supply to it. You probably already noticed that instead of supplying values to the `INSERT`, you have several question marks – those are the statement parameters, which you will fill in when you want to execute the statement.
2. When you call `$statement->execute()`, you pass it an array whose values will be mapped to the `?` placeholders in your `INSERT` query. This way, you don't need to worry about escaping the data to prevent SQL injections yourself. The prepared statement takes care of it for you.
3. After the statement is executed, you check `$statement->rowCount()` to see if there was exactly one affected row in the database. If not, it means something went wrong, so you throw an exception.
4. Finally, inside the `catch` block you do a bit of error handling – you save the exception error message in `$error` and return `null` (this makes the factory method `createPassWithUniqueSerialNr()` return a `null` value).

If you quickly switch back to `geteasterpass.php` you will see how the errors returned from the database are handled. There's a check for a `null` value and the returned error is printed in the browser:

```

if ($coupon!=null) {...}
else {
    //there was an error
    die("Error: ".$error);
}

```

Now it's time for another test! In your device's browser, navigate to <http://yourdomain.com/egghunt/> again and press the button – a new pass should show up for review (just as it did before).

Now switch to your favorite MySQL querying tool and have a look at the contents of the passes table. WOOT! There's the pass data, stored safely for future reference:

The screenshot shows a MySQL Workbench interface with two tabs: 'Query' and 'Script'. The 'Query' tab is active and contains the SQL command: 'SELECT * FROM passes;'. Below the query, the results are displayed in a table format. The table has columns: serialNr, authenticationToken, lastUpdated, credit, barcode, and passTypeID. One row is visible, with the following values: 134400108, b3013c0c8b8ee32fea97447ed3742a552fc02, 134400108, 0, 1344001081427030, and pass.yourdomain.wstest.

| serialNr | authenticationToken | lastUpdated | credit | barcode | passTypeID |
|-----------|---------------------------------------|-------------|--------|------------------|------------------------|
| 134400108 | b3013c0c8b8ee32fea97447ed3742a552fc02 | 134400108 | 0 | 1344001081427030 | pass.yourdomain.wstest |

Sweet!

The main idea behind keeping the pass data in a database is to be able to regenerate the pass later on – i.e., when you are requested to send an updated version of the pass to the user's device. Then you will need to include the same unique data you had in the first place – serial number, barcode message, authentication token, etc.

Let's add a method to the `EggHuntPass` class to generate a .pkpass file out of a row of data grabbed from the database. This will prepare you for the next big step – making Passbook on the iPhone and your server talk to each other.

First things first – add this method to `EggHuntPass`:

```

static function passWithSerialNr($serialNr)
{
    // 1
    $pass = new EggHuntPass();

    // 2
    $db = Database::get();

    // 3
    $statement = $db->prepare(
        "SELECT * FROM passes WHERE serialNr = ?");

    $statement->execute(array($serialNr));

    // 4
}

```

```

$row = $statement->fetch(PDO::FETCH_ASSOC);

// 5
if (!$row) {
    //no pass with such serialNr found
    return null;
}

// 6
$pass->content['serialNumber'] = $row['serialNr'];
$pass->content['authenticationToken'] =
    $row['authenticationToken'];
$pass->content['barcode'][ 'message' ] = $row['barcode'];
$pass->content['passTypeID'] = self::$passTypeID;
$pass->content['generic'][ 'primaryFields'][0][ 'value' ] =
    sprintf("$ %.2f", $row['credit']);

// 7
$pass->writeAllFiles();
return $pass;
}

```

If you compare `createPassWithUniqueSerialNr()` and `passWithSerialNr($serialNr)`, you will notice that in structure they are pretty similar – the main difference is that one creates a new pass with random values, while the other loads the necessary data from the database.

Again, let's very quickly go over the code:

1. First you get an instance of `EggHuntPass` (in the `$pass` variable).
2. Then you get an instance of a connection to the database (in the `$db` variable).
3. “`SELECT * FROM passes WHERE serialNr = ?`” fetches the row from the `passes` table for the given serial number.
4. You fetch the pass details into the `$row` dictionary. When you execute a prepared statement, which returns a result set from the database, you call `fetch()` on it to fetch rows one by one. Since you expect a maximum of one returned database record, calling `fetch()` once is enough.
5. If a row wasn't found in the `passes` table for the given serial number, you return `null`.
6. Then you fill in the pass metadata and field values in the same way as you did when creating it, but this time you use the serial number, token, barcode and credit from the `$row` dictionary.
7. Finally, you write the `.pkpass` file and return the pass object.

All right! Now that you have this awesome method in the `EggHuntPass` class, you can update the credit column in the database and call `passWithSerialNr()` in order to create an updated version of the pass. Of course, you don't have any way to test this yet – but you're one step closer!

The difference between creating a new pass and getting an updated existing one is depicted below:



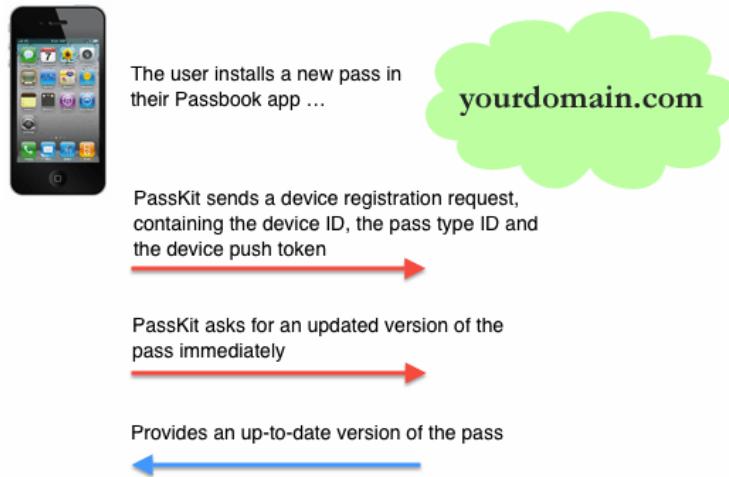
Cool! Things are coming together. ☺

Making Passbook talk to your server

Quickly open `pass/source/pass.json` and have a look at the fields at the top of the file. There are two fields you haven't seen before:

- `webServiceURL` – this field tells PassKit the URL of your web service on your server, so that PassKit can talk to you on this address. Go ahead and update the domain name for this from `yourdomain.com` to whatever the domain name is that you are using for your server (but keep the rest of the path as-is).
- `authenticationToken` – a shared secret between the device where the pass is installed, and your server. This token will authenticate the client asking you for data as an owner of a copy of the pass. You can leave this with the placeholder text, because the server code will update this.

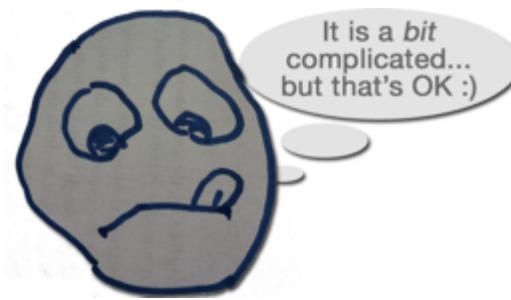
Let's have a look at a very simplified schema of the process of installing and updating a pass:



When a user downloads a pass and imports it into their Passbook application, a few things happen if the pass has a `webServiceURL` field:

- PassKit sends a registration request to your server. The request contains the pass serial number, the device ID, the authentication token you assigned to the pass, and the push token to use if you want to send a push notification to the pass.
- You need to store all the information from that registration request in your database (in the `devices` table) so that you can send updates to specific devices with specific passes installed on them.
- When the registration goes through, PassKit also asks your server for the latest version of the pass. This is necessary because passes can be sent around via email, saved as files, etc., so there's no guarantee that some of the pass data hasn't changed by the time the pass is imported into Passbook.
- In response to PassKit's request, you need to regenerate the pass using the data stored in your database, and send the `.pkpass` file back as a download.

This doesn't seem so complicated, right?



Note: The request for an update doesn't always seem to happen, but it is safe to assume that PassKit will ask you for a pass update within a reasonable amount of time.

Let's look at what you need to implement in order to have this initial pass registration process work:

1. Create a new pass and update it with the latest pass information – done.
2. Provide the random value stored in the `authenticationToken` field in your pass (the shared secret between PassKit and your server) – done.
3. Web service to handle device registrations – to do.
4. Web service to handle requests of updated passes – to do.

You're half way there! Let's start implementing the PassKit web service on your server.

The PassKit web service

Let's see how to implement a web service on your own server according to Apple's specifications. The most important thing to note is that you're going to need the `mod_rewrite` Apache module enabled on your server. (This was one of the required server features I mentioned at the beginning of the chapter.)

When PassKit desires to communicate with your server, it makes HTTPS requests to your web service. The device registration URL, for example, might look like this:

```
https://yourdomain.com/egghunt/ws/v1/devices/aaa56af38cfabc90d4b3  
6fff451f2a2/registrations/pass.com.yourdomain.freehugcoupon/134356  
534391
```

Or, if we highlight the parts of the URL that are dynamic:

```
https://yourdomain.com/egghunt/ws/v1/devices/<DEVICE_ID>/registrat  
ions/<PASS_TYPE_ID>/<PASS_SERIAL_NR>
```

You can quite easily figure out that this URL won't match an actual path on your website – you will have to configure Apache's `mod_rewrite` module to handle requests similar to this one, and direct them to your script so that it can handle the requests.

Fortunately, that's incredibly easy. You can use an Apache module to redirect all URLs in the `ws` subfolder to a single script of your choice.

Create a new subfolder inside **egghunt** and call it **ws** (short for web service). Inside `ws`, create a new file and call it **.htaccess**.

Copy the following into .htaccess:

```
<IfModule mod_rewrite.c>

    RewriteEngine On

    RewriteCond %{REQUEST_FILENAME} !-f

    RewriteRule .* /egghunt/ws/index.php

    RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]

</IfModule>
```

If this looks like some kind of black magic, it's not. ☺ It's fairly simple:

1. The first line turns on the rewrite engine.
2. The second line instructs the engine to keep redirecting all URLs for which a matching file cannot be found.
3. All redirected URLs will be handled by this script: /egghunt/ws/index.php. (If, for some reason, egghunt is not at the root level on your server, you would need to modify this path accordingly to give the full path from the root of the server.)
4. The fourth and last line checks whether there's an HTTP header called Authorization, and if so, imports it as a server environment variable into PHP. We'll discuss the Authorization header in a moment.

Note: You probably don't need the last line. However, I tested on several different servers with various Apache + PHP versions and some failed to read the HTTP Authorization header automatically. So it's safest to just have that last line in your .htaccess file to make sure everything works as expected.

That's it! A single PHP script in the ws folder will handle all of the possible URLs PassKit might request from your server. Pretty handy! Your script will know what the original URL looked like, so it will be able to fetch the requested serial number, pass type ID, etc.

The web service controller

First copy **lib.php** from the resources for this chapter to your project's **ws** folder. This file includes two helper functions, which are going to help you out in developing your web service:

- `httpResponseCode($code)` – PassKit counts on a proper HTTP response code from your web service to determine the result of the HTTP request. If the request goes through, PassKit expects a "200 OK" response, but when invalid authorization

data was sent over, you need to correctly answer with a “401 Unauthorized.” This little function helps you respond with the correct HTTP headers.

- `getallheaders()` is a built-in PHP function that helps you read the HTTP request headers. You need it because Apple sends over the pass authentication token as an HTTP request header. However, some older versions of PHP don’t have this function, so there’s one in `lib.php` to partially mimic the functionality.

Now create a new file inside your `ws` subfolder and call it **index.php**. This will be your web service controller – it’ll parse the input data, validate it, and then invoke the proper class to handle the request.

Paste inside **index.php**:

```
<?php
require_once("lib.php");

//get the request parameters out of the request URL
$requestURL = (($_SERVER['REDIRECT_URL']!="")?
    $_SERVER['REDIRECT_URL']:$_SERVER['REQUEST_URI']);
$scriptPath = dirname($_SERVER['PHP_SELF']);

$requestURL = str_replace($scriptPath, "", $requestURL );
$requestParts = explode("/", $requestURL);
```

On the first line, you include `lib.php`. Then you parse the request parameters out of the request URL. Here’s what happens line-by-line:

1. In `$requestURL` you save the contents of the environment variable “`REDIRECT_URL`” (or if it is blank, “`REQUEST_URI`”, since the first one might be missing on some servers, depending on configuration).

`$requestURL` now holds something like:

```
/egghunt/ws/v1/devices/aaa56af38cfabc90d4b36fff451f2a2/registrations/pass.com.yourdomain.freehugcoupon/134356534391
```

2. The “`PHP_SELF`” environment variable is the file path to the currently executed PHP script. Then in `$scriptPath`, you store only the folder part of the “`PHP_SELF`” file path. `$scriptPath`’s value should now be:

`/egghunt/ws`

3. On the next line, you strip the “`/egghunt/ws`” part out of `$requestURL`.

`$requestURL` now looks something like this:

```
/v1/devices/aaa56af38cfabc90d4b36fff451f2a2/registrations/pass.com.yourdomain.freehugcoupon/134356534391
```

4. On the last line, you split the requested URL into parts, using / as the delimiter. The \$requestParts array now contains:

```
[ "", "v1", "devices", "aaa56af38cfabc90d4b36fff451f2a2",  
 "registrations",  
 "pass.com.yourdomain.freehugcoupon", "134356534391" ]
```

As you can see, at the end of this piece of code \$requestParts holds all the data your controller needs to handle the request:

- The API version – “v1”
- The API controller to handle the request – “devices”
- The device ID making the call
- The action the API controller has to take – “registrations”
- The pass type ID
- The pass serial number

Sweet! Let’s validate the request to the web service! At the end of **index.php** add:

```
//check for valid API version  
$validAPIVersions = array("v1");  
$apiVersion = $requestParts[1];  
  
if (!in_array($apiVersion, $validAPIVersions)) {  
    httpResponseCode(404);exit();  
}
```

You define a list of the valid API versions (right now it’s only “v1”) and store the requested API version in \$apiVersion. On the next line, you check if the requested API version is listed in \$validAPIVersions and if not, you send a “404 Not found” HTTP response and exit the program.

That was pretty easy! Let’s add a validation for the possible API endpoints – these are:

- devices – for handling device registrations and updates.
- passes – for requesting up-to-date pass downloads.
- log – for Apple to talk to your server about your passes.

Add the PHP code to check for these:

```
//check for valid API endpoint  
$validEndPoints = array("devices", "passes", "log");  
$endPoint = $requestParts[2];  
  
if (!in_array($endPoint, $validEndPoints)) {
```

```
        httpResponseCode(404);exit();  
    }
```

Once again you take the same approach – define a list of allowed endpoints in `$validEndpoints`, store the requested endpoint in `$endPoint`, and then if the requested one is not found in the list, return a “404 Not found” HTTP response.

Next, create a sub-folder inside ws and call it **v1**. This will be the folder where you store all the API controllers for version 1.0 of the API. (Again – this is the only version right now, but it’s nice to build a flexible solution from the start, right?)

Inside the v1 folder, create three new files and call them **Devices.php**, **Passes.php** and **Log.php**.

Now you need to make your web service controller load and instantiate the respective class from within the v1 folder. Let’s add the code to do that. At the end of **ws/index.php** add these lines:

```
//get the endpoint class name  
$endPoint = ucfirst(strtolower($endPoint));  
$classFilePath = "$apiVersion/$endPoint.php";
```

The first line converts the endpoint name from the URL to lowercase and then makes the first character uppercase. Thus, if the request URL you’ve got contained “/v1/devices/acc3...” in `$endPoint`, you will now have “Devices”.

Then you put the API version and the modified endpoint name together, and you end up with a file path like “v1/Devices.php” stored in `$classFilePath`. Exactly the file path you need to include the respective endpoint controller class!

One more check is never too much, so add a check to see if the controller class file exists:

```
if (!file_exists($classFilePath)) {  
    httpResponseCode(404);exit();  
}
```

Finally, after everything is validated and you have the API controller class you need, add the code to make an instance of the target class and send the request details to it:

```
//load the endpoint class and make an instance  
try {  
    require_once($classFilePath);  
    $instance = new $endPoint($requestParts);  
}  
catch (Exception $e) {  
    //save $e->getMessage() to a log file
```

```
    httpResponseCode(500);  
}  
  
exit();
```

Thanks to the fact that PHP is an interpreted language, you can dynamically include classes by just providing the file path as a string. But not only that, you can then create an instance of the class by having the class name in a string variable!

Let's look at the code:

1. Wrap the class instantiation in a `try` block to handle any exceptions from within the class constructor.
2. Include the endpoint class file (its path is stored in `$classFilePath`).
3. This is a wild one – if you have stored the string “Devices” in the `$endPoint` variable, then the effect of calling:

```
new Device($requestParts)
```

and:

```
new $endpoint($requestParts)
```

is the same.

“But why all this trouble? Can’t we just have a few `if` statements?”, some of you might ask.

Well, when version 2 of the PassKit API comes out, there might be a new endpoint called “waffles.” If you use the code above, you will only need to create a new controller class “Waffles” and add “waffles” to the list of allowed endpoints. No need for new static file includes, if statements or additional code to the web service controller. Much simpler, huh? ☺

4. Finally, in case an exception is thrown inside the `catch` block, you return an HTTP response “500 Server error” to let PassKit know that something went wrong on your side.

That’s all! This single file handles all possible calls to your PassKit web service.

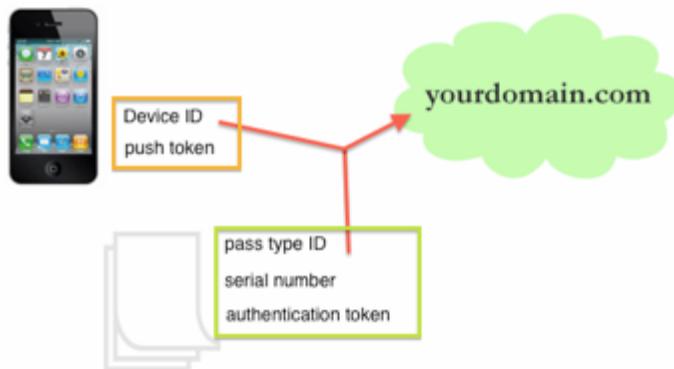
Next up, adding the code behind registering a new device with the server!

Register me please, I want to be up-to-date

When a new pass is imported on to the device, PassKit sends a registration request to your web service. The key facts about the request are:

- The API endpoint is “**devices**.”
- The HTTP request method is **POST**.

- You get the **device ID**, **pass type ID** and **pass serial number** in the request URL.
- The pass **authenticationToken** is sent as an HTTP header called “**Authorization**.”
- The **POST data** sent with the request is a JSON-encoded dictionary having one key called “**pushToken**” and it provides you with the push token you need to use when talking back to the device in question.



Apple then expects a few different response codes from you, depending on the outcome of the registration request:

- “401 Unauthorized” if the `authenticationToken` doesn’t match the serial number of the pass.
- “201 Created” if the device registration was created successfully in your database.
- “200 OK” if the device was already registered for that pass.

You’ll start by implementing the `Devices` class constructor – it’ll take in the request, as sent over by `index.php`, and then it’ll invoke the proper methods for all possible controller actions. Open up **ws/v1/Devices.php** and add the initial class code:

```
<?php

require_once(dirname(__FILE__)."/../../../../class/Database.php");

class Devices
{
    //takes in the request URL parameters and creates a response
    function __construct($params) {
        //detect the HTTP method - can be get, post or delete
        $method = strtolower($_SERVER['REQUEST_METHOD']);
        $action = strtolower($params[4]);
    }
}
```

```
//switch between the list of valid requests or return 401
switch ($method . ":" . $action) {
    case "post:registrations":
        $this->createRegistration($params);
        break;

    default: //not a valid web service call
        httpResponseCode(401);exit();
        break;
}
}

}
```

This is the code to define the new class and its initial constructor:

1. First you include the `Database` class from the shared “class” folder, since you will need to interact with the database in this class.
2. Then you define the `Devices` class and its constructor, which takes in one parameter: the array of the request URL parameters, `$params`.
3. Next you store in `$method` the HTTP request method. The HTTP methods that PassKit would send to the “devices” endpoint are POST, GET and DELETE.
4. You also store into `$action` the endpoint action – in version 1.0 of the PassKit web service, this is always “registrations”.
5. Then you use a switch on the combined value of the HTTP method and the endpoint action.
6. For now you have only one case inside the switch: “post:registrations”. This is the case when a new pass is installed on a user device and PassKit registers the device for that pass. You call the class method `createRegistration()` to create a new database device registration (you’ll add this method in a moment).
7. Finally, in the default case of the switch, which handles all cases not listed above, you return a “401 Unauthorized”.

So far things are going great – now let’s also add the implementation of the `createRegistration()` method in order to handle the registration requests.

Just before the closing bracket at the bottom of the code, add the first part of this method:

```
//register device with ID for a pass instance
private function createRegistration($params)
{
    $deviceID = $params[3];
    $passTypeID = $params[5];
```

```

$serialNr    = $params[6];
	payload      =
	json_decode(file_get_contents('php://input'),true);
$pushToken   = $payload['pushToken'];
$headers     = getallheaders();
$authenticationToken = str_replace("ApplePass ","",
$headers['Authorization']);

$db = Database::get();

}

```

This is all the input that the method will use. Quite a few parameters in there, so let's see what is what:

- The device ID, serial number and pass type ID you read from the URL elements passed to the constructor.
- Then `file_get_contents('php://input')` fetches the raw contents of the HTTP POST request, and by running the result through `json_decode()`, you get a dictionary holding the device push token.
- You store the device push token into `$pushToken`.
- You use the `getallheaders()` function to grab the HTTP request headers.
- You expect an HTTP header with the name of "Authorization" and a value in the format "ApplePass <THE PASS AUTH TOKEN>". You grab the "Authorization" header and strip "ApplePass" from it – `$authenticationToken` should now contain the proper token you generated originally for your pass.
- Finally, you fetch an active connection to your MySQL database.

OK, you are now ready to go on with validating the request and then store the registration in the database, if it is valid.

At the end of the `createRegistration()` method (within the method body, not after), add this:

```

//check authorization token
$statement = $db->prepare("SELECT COUNT(*) FROM passes
WHERE passTypeID=? AND serialNr=? AND authenticationToken=?");
$statement->execute(array($passTypeID, $serialNr,
$authenticationToken));

if ($statement->fetchColumn() == 0) {
//no such pass found in the database
httpResponseCode(401);exit();
}

```

First you check the validity of the authorization token – if the token is not valid, you don't need to do anything more! Just bail out.

You can uniquely identify the pass by searching for the **pass type ID + serial number + authentication token** combination, so you build up the relevant SQL query for this.

Once you execute the statement, you go ahead and fetch the value returned by the SQL COUNT() function. If there were no matching passes, you return "410 Unauthorized" and exit.

If the registration request is authorized, you need to check if there's an existing registration for that device, so add this next:

```
//check for existing registration
$statement = $db->prepare("SELECT COUNT(*) FROM devices
    WHERE device=? AND serialNr=?");
$statement->execute(array($deviceID, $serialNr));
```

This time – if there are no matched table rows – you go on to create a new registration, and if there's already a registration with the given device ID and serial number, you will update it with the provided push token. Add this:

```
if ($statement->fetchColumn() == 0) {
    //insert the registration in the database
    $stmtInsert = $db->prepare("INSERT INTO devices (device,
pushToken, serialNr) values (?, ?, ?)");
    $stmtInsert->execute(array($deviceID, $pushToken, $serialNr));
    httpResponseCode(201);
} else {
    //update the pushToken of the existing registration
    $stmtUpdate = $db->prepare("UPDATE devices SET pushToken=? WHERE
device=? AND serialNr=?");
    $stmtUpdate->execute(array($pushToken, $deviceID, $serialNr));
    httpResponseCode(200);
}
```

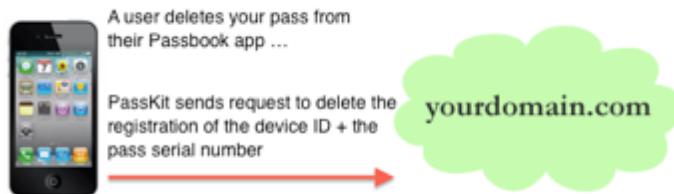
This code should look familiar – it's just building up more prepared statements and firing them at the database.

In the first branch of the `if` statement, you create an `INSERT` query to add a new registration in the database, and in the `else`, you create an `UPDATE` query to update an existing device registration.

And that's a wrap for the registrations!

I'd like to unregister please

Let's add one more API request – namely the one that PassKit makes to your server when a user does not need your pass anymore and deletes it from their Passbook. (Or they just shred a pass for the fun of it.)



The URL is the same as the one used for registrations, with the one difference being that the HTTP request method is DELETE.

Apple expects a couple of different response codes from your web service:

- “401 Unauthorized” if the `authenticationToken` doesn’t match the pass type ID and serial number of the pass.
- “200 OK” if the device was successfully unregistered.

Let's add the code to do this. Scroll up to the constructor body in **Devices.php** and add a new case inside the `switch` block:

```
case "delete:registrations":
    $this->deleteRegistration($params);
    break;
```

Your whole `switch` block should look now like:

```
//switch between the list of valid requests or return 401
switch ($method.":".$action) {
    case "post:registrations":
        $this->createRegistration($params);
        break;

    case "delete:registrations":
        $this->deleteRegistration($params);
        break;

    default: //not a valid web service call
        httpResponseCode(401);exit();
        break;
}
```

OK, so if a request comes in via DELETE it will be handed over to `deleteRegistration()`. Let's add that too! At the end of the class body (just before the final closing bracket in the file), add the initial method code:

```
//delete device with ID
private function deleteRegistration($params)
{
    $deviceID = $params[3];
    $passTypeID = $params[5];
    $serialNr = $params[6];
    $headers = getAllHeaders();
    $authenticationToken = str_replace("ApplePass ", "", 
        $headers['Authorization']);

    $db = Database::get();

}
```

Just as before, you fetch all the input data in advance and store it in local variables.

Let's go on – at the end of the `deleteRegistration()` method body add:

```
//check authorization token
$statement = $db->prepare("SELECT COUNT(*) FROM passes
    WHERE passTypeID=? AND serialNr=? and authenticationToken=?");
$statement->execute(array($passTypeID, $serialNr,
    $authenticationToken));

if ($statement->fetchColumn() == 0) {
    //no such registration found in the database
    httpResponseCode(401);exit();
}
```

Above, as in `createRegistration()`, you check whether the passed type ID, serial number and token match a pass record in the database. If not, you return "401 Unauthorized" and quit the script execution.

Finally, you need to add the code to actually delete the device registration from the database:

```
//delete the registration from devices
$statement = $db->prepare("DELETE FROM devices
    WHERE device=? AND serialNr=?");
$statement->execute(array($deviceID, $serialNr));

httpResponseCode(200);exit();
```

OK! Registering and unregistering devices is done!

You *could* give importing passes on your device a try, but you have only half of the required web service methods implemented. So it's probably best to hold off for just a tiny bit more.

What will happen if you import one of your passes right now? The pass is valid, so it will be added to your Passbook.

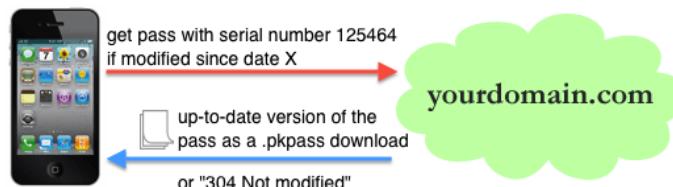
Then Passbook will try to fetch the latest version of your pass, but you don't have this web service call implemented, so PassKit will get into a loop trying to fetch the updated pass over and over again. This will jam up your Console with debug output. Not nice.

For the moment, go on implementing code and you will test as soon as you have the minimum web service implementation.

The next step is to provide the latest version of the pass.

An up-to-date pass

As previously mentioned, immediately after registering a device, PassKit will ask for an up-to-date version of the pass, so that the user doesn't see any outdated information in Passbook.



The web service request URL for getting an updated pass looks like this:

```
https://yourdomain.com/egghunt/ws/v1/passes/pass.com.yourdomain.fr  
eehugcoupon/134356534391
```

And here it is with the dynamic parts highlighted:

```
https://yourdomain.com/egghunt/ws/v1/passes/<PASS_TYPE_ID>/<PASS_S  
ERIAL_NR>
```

The key information for this request is:

- The API endpoint is "**passes**."
- The HTTP request method is **GET**.
- You get the **pass type ID** and **pass serial number** in the request URL.

- The pass **authenticationToken** is sent as an HTTP header called “**Authorization**.”
- The HTTP request contains an “**If-Modified-Since**” HTTP header. It contains the time of last update of the pass that the device already has. You should check to see if you have a newer version in the database, and if not, return a “304 Not modified” status.

The expected result codes for the call are as follows:

- “200 OK” response, including the contents of the .pkpass file.
- “401 Unauthorized” if there is an authorizations issue.
- “304 Not modified” if the client already has the latest version of the pass.

Let’s go ahead and add the implementation of the Passes API controller class. This class will contain quite a bit of code (which you are already familiar with, though): detecting the HTTP method, fetching the request parameters, and fetching data from the database.

Open up **ws/v1/Passes.php** and paste in the initial class body:

```
<?php
require_once(dirname(__FILE__)."/../../class/Database.php");

class Passes
{

    //takes in the request URL parameters and creates a response
    function __construct($params) {

        //detect the HTTP method - can be get, post or delete
        $method = strtolower($_SERVER['REQUEST_METHOD']);

        //switch between the list of valid requests or return 401
        switch ($method) {
            case "get":
                $this->getPass($params);
                break;

            default: //not a valid web service call
                httpResponseCode(401);exit();
                break;
        }
    }
}
```

Much of this should already be very familiar.

1. First you include the `Database` connection class.
2. Then you define the `Passes` class and its constructor.
3. You fetch the HTTP request method into `$method`.
4. Then you declare a `switch` block with a single case, "get". The `passes` endpoint responds only to GET requests.
5. You also have a default case, which returns "401 Unauthorized" if a client sends any request besides a GET request.

Time to add the `getPass()` method, which will validate the input, fetch the pass data from the database, and then serve the `.pkpass` as a file download to the client.

Add a new method to the class body (just before the final closing bracket):

```
//download the pass with given
//authentication token and serial number
private function getPass($params)
{
    $passTypeID = $params[3];
    $serialNr   = $params[4];

    $headers = getAllHeaders();
    $authenticationToken = str_replace("ApplePass ", "", 
        $headers['Authorization']);
    $ifModifiedSince = $headers['If-Modified-Since'];

    $db = Database::get();

}
```

You fetch the pass type ID and serial number, then you fetch the authentication token from the HTTP headers (exactly as before), and you also fetch the HTTP header called "If-Modified-Since". (We'll discuss If-Modified-Since later on.)

In the next step, you check whether the requested pass exists and if proper authentication has been sent over. Paste this in directly after the previous piece of code:

```
//select the pass with the given details
$statement = $db->prepare("SELECT * FROM passes
    WHERE passTypeID = ? AND serialNr = ?
        AND authenticationToken = ?");

$statement->execute(array($passTypeID, $serialNr,
    $authenticationToken));

//try to fetch the pass from the database
```

```
$row = $statement->fetch(PDO::FETCH_ASSOC);

//check if a matching pass was found
if (!$row) {
    //no pass was found
    httpResponseCode(401);exit();
}
```

Much like the previous instances, you create a new prepared statement with an SQL query, which will match a pass record in the database, but only if the correct serial number, authentication token, and pass typeID were sent over to the web service.

You execute the statement and provide the actual parameter values, then call `fetch()` on the statement object in order to try and fetch one record.

If a pass record was not found, you respond with a "401 Unauthorized".

If a pass was matched in the database, it's time to ask yourself a certain question: Does the client really need a new copy of the pass?



"If-Modified-Since" contains the timestamp of the last update to the pass that the device already has installed. You can compare it with the `lastUpdated` timestamp of the pass in the database and see if there's anything new. If there's no new information on the pass, you should try to save the user's bandwidth and not send the .pkpass file. Add the following condition:

```
//check if the pass was modified on the server
if ($ifModifiedSince >= $row['lastUpdated']) {
    //the pass was not modified
    httpResponseCode(304);exit();
}
```

Both timestamps are integers, so simple comparison is enough to check whether or not the `lastUpdated` column of the pass record contains a later timestamp than the HTTP request header.

If the timestamp sent via the request is greater than or equal to the local timestamp (the greater than case must clearly be a mistake, but let's handle it

anyway), then you simply return a “304 Not modified” HTTP header to PassKit and exit.

At this point in the code, you already know that the request is valid, and also that you will need to send over a newer version of the pass, so you might as well send back a newer “**Last-modified**” timestamp for the requested pass:

```
//provide the last modified time to the client  
header("Last-modified: \"$row['lastUpdated']", true);
```

From here on, it’s very easy. All you need to do is:

1. Include the `EggHuntPass` class.
2. Get a pass instance for the serial number sent via the request.
3. Dump the `.pkpass` file contents back to the client.
4. Finish the script execution.

So add this bit of code to implement that:

```
//create a new pass instance with the latest data  
require_once("../class/EggHuntPass.php");  
$pass = EggHuntPass::passWithSerialNr($serialNr);  
  
//output the pass contents to the client  
$pass->outputPassBundleAsWebDownload();  
  
exit();
```

Now you have a complete passes endpoint for your web service!

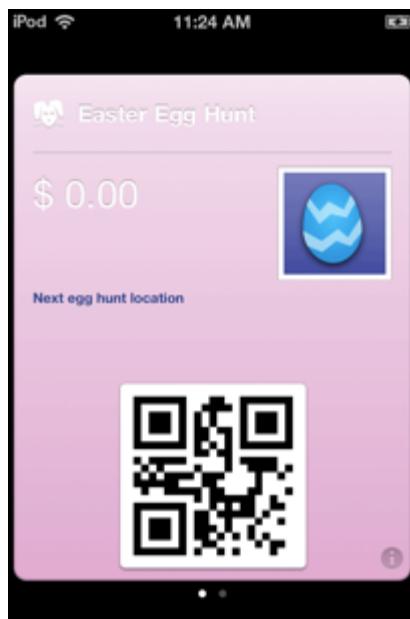
Congratulations, you also have the bare minimum for the PassKit web service! What you’ve got is still not complete, but at this point you can afford to do a little test to see how things are working out. (Testing might still confuse PassKit a bit, but it will have to take one for the team right now. ☺)



Passbook talking to your server

It's finally time to test your new server code that communicates with Apple's servers for the first time!

Make sure you have migrated all the new files and changes you've made so far to your web server. Then go ahead and open the <http://yourdomain.com/egghunt/> page in Safari on your device, and import a pass. Then switch to the Passbook app to make sure the card pops up:



Do you notice that page indicator at the bottom? It's another neat Passbook feature – I have already imported two passes into my Passbook from the Easter Egg Hunt page, so they stack up nicely and you can scroll horizontally through them. (The game participants need only one.)

By looking at the device Console output, you can trace the whole registration process.

Note: If you see different log messages than the below (such as 401 errors), try removing all passes from Passbook, re-starting your device, and re-importing the pass into Passbook just to make sure you're starting from a clean slate.

First Passbook registers the device for the just-imported pass:

```
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Registering pass  
with serial number <134424504716>, typeID  
<pass.com.yourdomain.freehugcoupon>, at service URL
```

```
http://yourdomain.com/egghunt/ws/  
  
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Generating POST  
request with URL  
<http://yourdomain.com/egghunt/ws/v1/devices/f91fcba90ddd91e1c946f0  
e887563412/registrations/pass.com.yourdomain.freehugcoupon/1344245  
04716>  
  
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Request contains  
header field <Authorization: ApplePass  
10aaa2a080e12d23787fa6877340dc3548356349>  
  
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Request contains  
body dictionary { pushToken =  
eeec287b1d589e7744bc7464a6065f9ef0df08c237b7514ee012e189e3d2adce; }
```

And the registration response feedback (handy to check what HTTP status code it returned):

```
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Register task got  
response with code 201  
  
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Registration task  
succeeded: <WDRegistrationTask: 0x1ddb6a00>
```

While the device registration task is going through to your server, PassKit also fires another request:

```
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Registering with  
APNS for topics {"pass.com.yourdomain.freehugcoupon"}
```

This request is sent to the Apple Push Notification Service servers and allows you to send push notifications to PassKit – more about push notifications and passes soon!

Finally, the device registration process sends a request to your server for a fresh copy of the pass contents:

```
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Generating GET  
request with URL  
<http://yourdomain.com/egghunt/ws/v1/passes/pass.yourdomain.wstest5  
/134424235028>  
  
Aug  6 11:24:19 MTT-iPod-4 passd[409] <Warning>: Request contains  
header field <Authorization: ApplePass  
123ab0081e61c3490c87b99794137b613b471245>
```

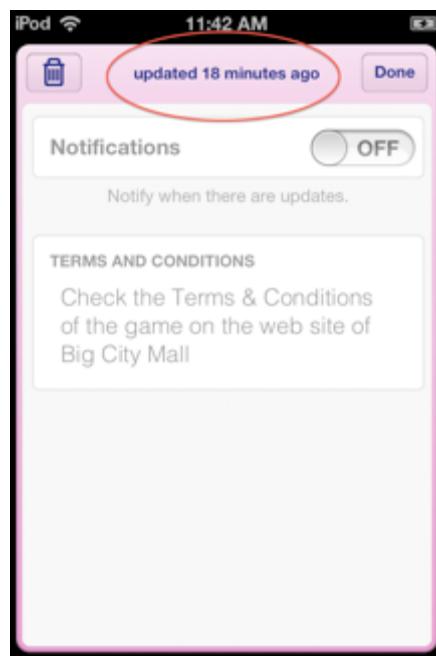
```
Aug  6 11:24:20 MTT-iPod-4 passd[409] <Warning>: Get pass
connection got response with code 200

Aug  6 11:24:20 MTT-iPod-4 passd[409] <Warning>: Get pass
connection received data and will try to instantiate a pass.

Aug  6 11:24:20 MTT-iPod-4 passd[409] <Warning>: Get pass task
succeeded.
```

Open Passbook on your device, open your pass, tap the “i” button at the lower right-hand corner, and look at the top. Sweet! The pass shows that it’s been updated from the server.

This is a big step forward – until now, your passes always showed their status as: “Never updated.” Awesome, you’re on your way!



OK, time to check how things are developing on the server side. Run a “`SELECT * FROM devices`” SQL query from your favorite querying tool, you should see your device registration like so:

| select * from devices | | |
|-----------------------|-------------------|--------------|
| *device | pushToken | *serialNr |
| f91fcba90ddd91e1c94 | eeec873b1d589e774 | 134424504716 |
| | | |

Neat! You have the device ID and the serial number of the pass installed. Additionally you have the push token that you will need later on to send push notifications to PassKit.

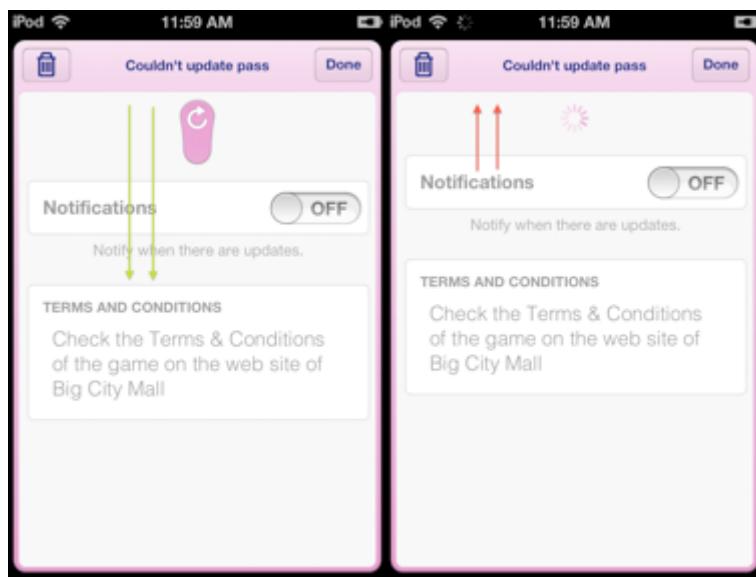
Your first pass updates!

Since you haven't even started implementing the game logic, you are going to make some little updates to the pass by hand, so that you can see how the process works.

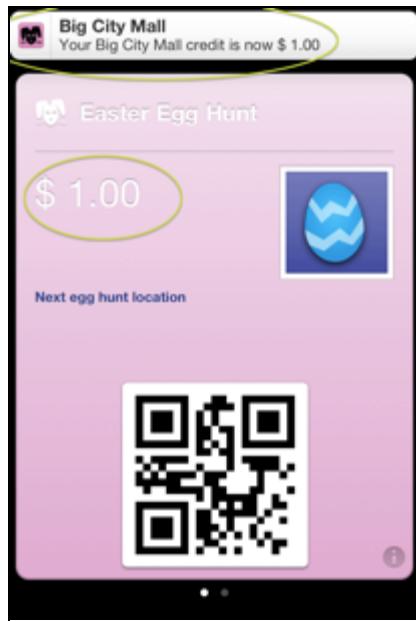
While you have your SQL query tool open, fire up this query in order to update the credit on your passes to \$1.00, and to change the last updated timestamp to the current time:

```
UPDATE passes SET lastUpdated=UNIX_TIMESTAMP(), credit = 1
```

OK, now your database is updated. You also need to fetch the updated version from the server, so switch to the back of your pass. Pull down and you will see one of those fancy new iOS 6 loading indicators:

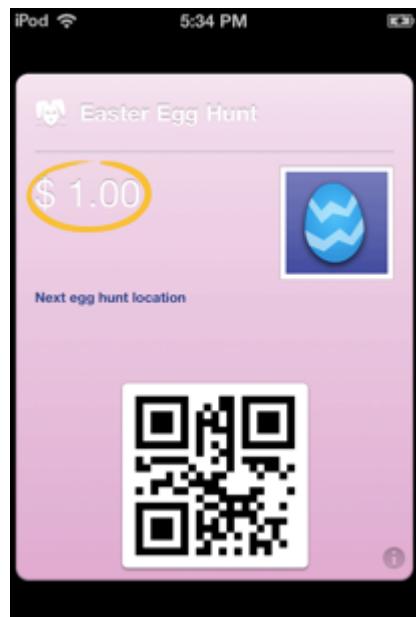


When Passbook finishes fetching the update from your web server, it switches to the front of your pass and shows a notification at the top! Fancy!



This is seriously awesome!

If you are quick and tap the notification banner at the top of the screen, this will invoke the Passbook animation, which highlights the changes to the pass. Give it a try!



Now the user can update their pass by pulling down when viewing the back of the pass. That's brilliant, but time to continue implementing the web service. ☺

Note: You better shred the passes you imported just now to stop PassKit from working itself up into a frenzy. (Remember that if the web service is not

complete, PassKit will keep trying to complete requests.) You are going to get more passes when the web service is fully implemented.

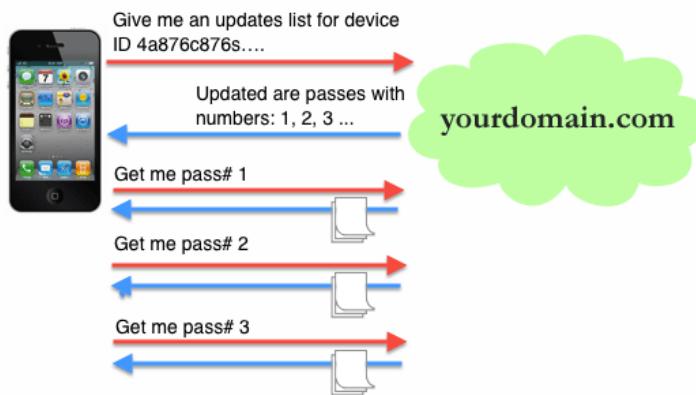
Did any of my passes change while I was out of reach?

Sometimes the user might be on a boat, in a tunnel, or on a mission to the moon so that they are out of reach for a while. When they get back online, PassKit will try to update all their passes with the latest information via the Internet.

In this case, your server will receive a device update request. Unlike the pass update request, this request doesn't expect information about just one pass. Rather, it wants an update for all passes registered for the device.

Basically, PassKit pings your server with: "Hey there baby, I'm back online – did I miss something?" Your server then responds with either "No, it's all good – relax!" or, "Yes, this and that changed, you better get busy."

And this is a diagram of the whole device update process:



You can quickly implement this web service call. You already implemented the passes endpoint (which provides the up-to-date pass versions), so let's have a look at the details of the device update request, and go on with coding.

The URL for the web service call looks like this:

```
https://yourdomain.com/egghunt/ws/v1/devices/aaa56af38cfabc90d4b3  
6fff451f2a2/registrations/pass.com.yourdomain.freehugcoupon?passes  
UpdatedSince=1358603580
```

It looks familiar, but there is also something you haven't seen so far – the `passesUpdatedSince` query parameter. Have a look at the dynamic parts of the URL, highlighted below:

```
https://yourdomain.com/egghunt/ws/v1/devices/<DEVICE_ID>/registrations/<PASS_TYPE_ID>?passesUpdatedSince=<UPDATE_TAG>
```

As you can see, the URL is pretty similar to the other calls to the devices' endpoint. The interesting part of the URL is, of course, `passesUpdatedSince`. This parameter is optional – the first time PassKit contacts your server, it won't send it over. The first time you send a response to this call, you provide an update tag. The next time PassKit asks you for updates, it will pass back to you the tag you sent it during the previous call.

Note: Apple does say this update tag can be anything, but to me it makes the most sense to use the timestamp of the latest update from the `passes` table. This makes things pretty easy. You have a "lastUpdated" column in the `passes` table, so you just need to update that column whenever you make changes to a pass record, and send the latest timestamp along with the response. Easy-peasy.

Let's have a look at the workflow of this web service call:

1. The API endpoint is "**devices**".
2. The HTTP request method is **GET**.
3. You get the **device ID** and **pass type** in the request URL.
4. In the **passesUpdatedSince** GET parameter, you receive the tag you sent over to the server the last time this endpoint was called.

The possible response statuses for this call are:

- "200 OK" and a JSON-encoded list of the passes that are outdated on the device.
- "204 No content" – in case there are no pass updates and the device's library is up-to-date.

So you can just open **ws/v1/Devices.php** and add this last action to the controller class. First, inside the constructor method, add another `case` to the `switch` statement:

```
case "get:registrations":  
    $this->getDeviceUpdates(  
        $params, @(int) $_GET['passesUpdatedSince']);  
    break;
```

When the HTTP request method is GET, you pass the URL request parameters and the `passUpdatedSince` GET parameter to a method called `getDeviceUpdates()`.

Note: `$_GET` is a special globally-accessible PHP dictionary, which is automatically populated with all parameters coming through the query string of the requested URL.

At the end of the class implementation, just before the final closing bracket, paste in the initial code for the new method:

```
//get list of pass updates for a device ID
private function getDeviceUpdates($params, $updateTag)
{
    $deviceID = $params[3];
    $passTypeID = $params[5];

    $db = Database::get();

    //grab the updated serial numbers
    $statement = $db->prepare(
        "SELECT passes.serialNr, lastUpdated FROM passes
        JOIN devices on (devices.serialNr = passes.serialNr)
        WHERE passTypeID = ? AND lastUpdated > ? AND device = ?");

    $statement->execute(array($passTypeID, $updateTag,
        $deviceID));

}
```

Let's see what's going on in this piece of code:

1. First you fetch the device ID and pass type ID from the parameters array supplied to the class constructor. You also have the `$updateTag` as a separate method argument, so you have all the data you need to proceed.
2. You grab a link to the database.
3. Then you create a prepared statement to return a list of all the serial numbers that have been updated since the `$updateTag` timestamp.

If the last update tag that PassKit received from your server is “1344247311”, then the next time it asks for updates, the SQL query might find these matches:

| serialNr | authenticationToken | lastUpdated | credit | barcode |
|--------------|------------------------|-------------|--------|------------------|
| 134424218046 | 8f6e414b750cb0ba2223f9 | 1344247341 | 1 | 1344242180464941 |
| 134424235028 | 647ab0081e61c3490c87b | 1344247321 | 1 | 1344242350281929 |
| 134424271493 | 2b7bcfc461cd9d45eab0c | 1344247301 | 1 | 1344242714932026 |
| 134424445501 | 86e84de6bd0c656600740 | 1344247311 | 1 | 1344244455014789 |
| 134424504716 | 10aaa2a080e12d23787fa6 | 1344247341 | 1 | 1344245047162600 |

The SELECT filters out any rows from the passes table that have a value in their lastUpdated column less than "1344247311", and it returns the list of the corresponding serial numbers from the serialNr column.

Now you also need to determine the latest timestamp. You could run another query on the MySQL database, but for once let's save resources – you can just as easily loop through the result set and do the check yourself.

Let's combine the two tasks of 1) building the serials list as an array, and 2) finding the latest update timestamp in one pass over the database result set. Add this code after the lines you just pasted in:

```
$serialList = array(); //1
$newUpdateTag = 0; //2

//3 loop and get the the update tag of the latest updated pass
//also add all serials into an array
while($row = $statement->fetch(PDO::FETCH_ASSOC)) {
    $serialList[] = $row['serialNr'];

    if ($row['lastUpdated'] > $newUpdateTag) {
        $newUpdateTag = $row['lastUpdated']; //4
    }
}
```

1. First you define a new empty array to store the serial numbers – \$serialList.
2. You use \$newUpdateTag to get the latest update timestamp and you set its initial value to zero.
3. The while loop executes as long as there are rows in the statement result set. Inside the body, each pass record data is stored in the \$row dictionary. In the body of the loop you keep adding the values of the "serialNr" column to the \$serialList array.
4. For each pass, you check if the row's "lastUpdated" timestamp is newer than the one you already saved in \$newUpdateTag. This way, after the loop goes over the whole result set you'll have the latest timestamp – pretty neat! It's an easy piece of code to write, but saves you an extra database query. Good job!

Next you have to proceed based on the result set actually returned from the database:

1. If there were no updated passes found in the database:
 - One case might be that the device ID is not registered at all – so you have to return a proper "404 Not found".
 - If the device is registered but there were no pass updates for that particular device, return a "204 No content".
2. And of course, in case there are passes with updates – go on with returning a JSON response.

OK, let's give that a try! Add the code to implement the logic for point #1 from the list above to the end of the `getDeviceUpdates` method:

```
//1
if (count($serialList)==0) {

    // 2
    $statement = $db->prepare("SELECT COUNT(*) FROM devices
        WHERE device = ?");
    $statement->execute(array($deviceID));

    if ($statement->fetchColumn() == 0) {
        // 3
        httpResponseCode(404);
    } else {
        // 4
        httpResponseCode(204);
    }

    // 5
    exit();
}
```

1. First you check whether the elements count in the `$serialList` array is zero – in this case, you won't be returning any JSON, only an HTTP response.
2. Then, to confirm the proper HTTP response, you run a query on the database to find whether the device is registered with your web service.
3. The next `if` condition checks whether there are device registrations found at all, and if there aren't, returns a "404 Not found" response.
4. Finally, if there are device registrations found, but no pass updates, return a "204 No content".
5. After you set the HTTP response code, you exit the execution of the script.

At this point, you know that there are updates, and you already have all the necessary data for a response, so you can go ahead and build the JSON response:

```
//prepare the response
$response = new ArrayObject();
$response['serialNumbers'] = $serialList;
$response['lastUpdated'] = $newUpdateTag;

//return JSON encoded response
print(json_encode($response));
exit();
```

`$response` is the dictionary object you will return to the client – you set the key `serialNumbers` to the list of updated pass serial numbers, and the `lastUpdated` key to the latest timestamp you already fetched.

Finally, you JSON-encode the `$response` dictionary and output it to the client. Success!

Now you have almost the complete set of calls for your web service – awesome! You can start thinking about the next steps for implementing the complete Easter Egg Hunt game.

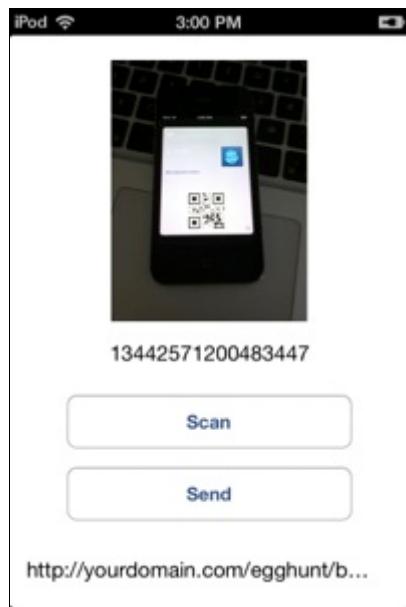
Your thinking might lead you to another part of the project, which we haven't tackled yet: building an app for the Easter Bunny so that it can scan people's passes and send store credit directly to their iPhones!

iOS barcode scanner app

Now for a little Xcode and Objective-C detour! Oh joy!

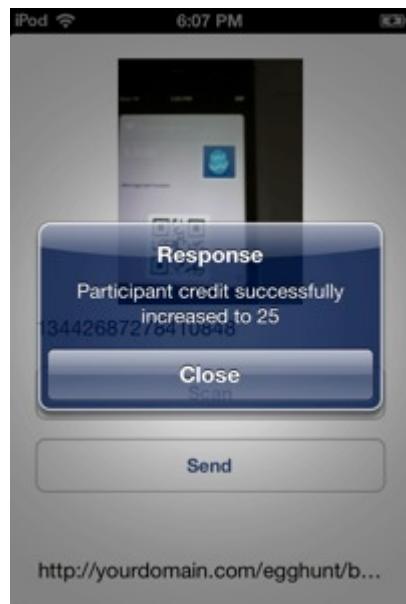


Let's have a look at what the finished app should look like:



Again, remember that this app is for the Easter Bunny. When a user finds him, the Easter Bunny will tap the Scan button and a camera view will pop up. When an iPhone with a pass on its screen enters the camera viewport, the app will scan the barcode and show the original barcode message under the photo of the barcode.

Then when the Easter Bunny taps Send, the barcode will be sent to the server and the server will return a response like this:



Cue one happy mall visitor! 😊

OK, so let's start building this app. Fire up Xcode and create a new project by choosing **File/New/Project...** and then from the dialogue select **iOS/Application/Single View Application**, and click **Next**.

Set the Product name to **BunnyApp**; make sure that the **Use Storyboards** and **Use Automatic Reference Counting** checkboxes are checked, and click **Next**.

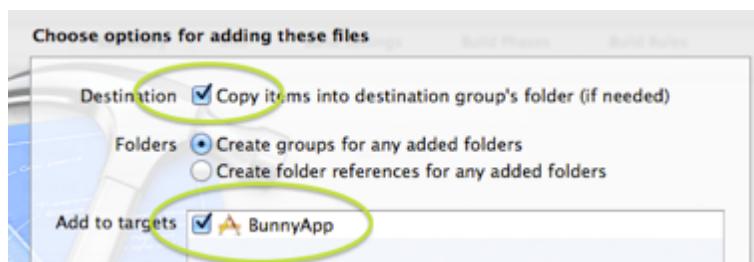
Save the project to a folder of your choice.

You are going to use two great libraries in order to implement barcode scanning and talking to your JSON API: **ZBar** and **AFNetworking**.

First you'll add barcode scanning to the app. You can read up about the ZBar library for iOS here: <http://zbar.sourceforge.net/iphone/>. Scroll down the page and find the link to download the "ZBar iPhone SDK" – the current version as of the time of writing is 1.2 and the direct link to the install file is:

<http://sourceforge.net/projects/zbar/files/iPhoneSDK/ZBarSDK-1.2.dmg/download>

Download the .DMG file and open it in Finder. There you will find a ReadMe, the license for the library, extra documentation, and a folder called ZBarSDK. Great! Just drag **ZBarSDK** into your Xcode project and make sure you choose to copy the files.



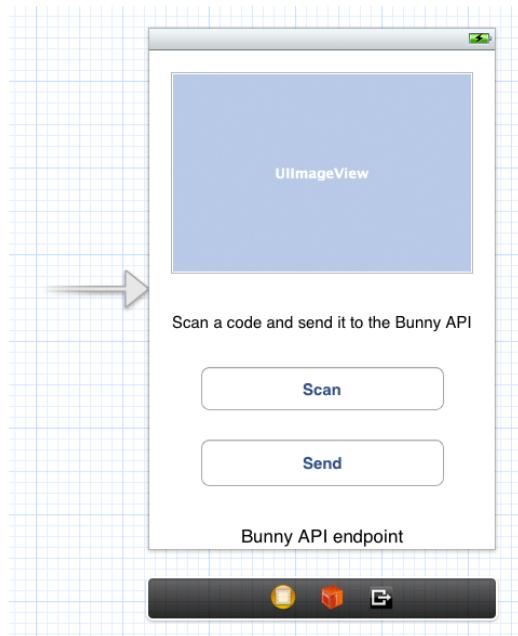
ZBar requires a few other iOS frameworks in order to work properly (some advanced image and video processing happens in the background when ZBar recognizes barcodes in the camera viewport), so choose the project root in Xcode's project navigator, select the **BunnyApp** target, and choose the **Build phases** tab.

Expand the **"Link binary with libraries"** pane and click on the **+** button. While holding the Command key, select all of these frameworks from the list:

- AVFoundation
- CoreMedia
- CoreVideo
- QuartzCore
- libiconv.dylib

Make sure that the list already contains **libzbar.a** as well. (If you copied the ZBar files over to your project, then you should see this library in the list.)

Now you need to add create the UI for the Bunny App. Open **MainStoryboard.storyboard** from your project file list and try to replicate the user interface as per the image below:



You need these elements:

- `UIImageView` – should be relatively big, as it will show the snapshot of the barcode being scanned; make sure Mode is “Aspect Fit.”
- A `UILabel` under the image view – make it almost as wide as the screen, and make sure the Autoshrink property in for the label is NOT “Fixed Font Size” (select either one of the others).
- Two `UIButton` elements – call the first one “Scan” and the second “Send.”
- And another `UILabel` at the bottom.

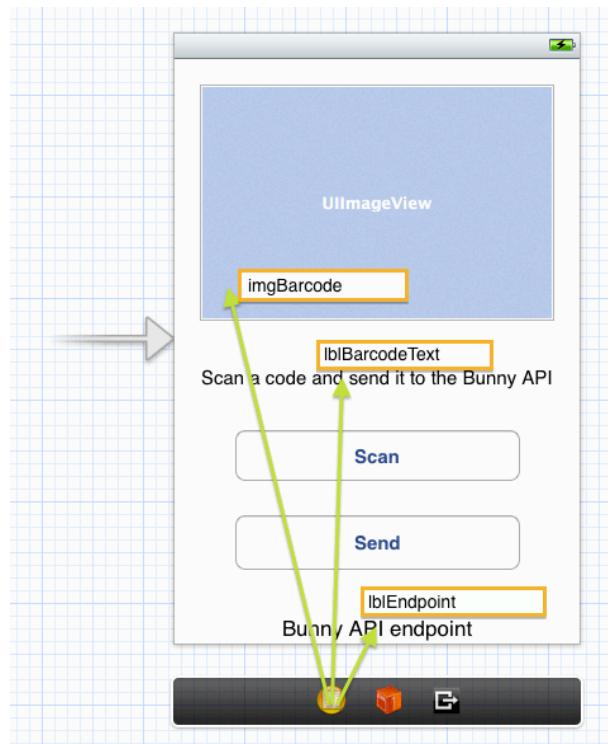
Now switch to **ViewController.m** and replace the empty class extension at the top of the file with:

```
#import "ZBarSDK.h"

@interface ViewController : UIViewController <ZBarReaderDelegate>
{
    IBOutlet UIImageView* imgBarcode;
    IBOutlet UILabel* lblBarcodeText;
    IBOutlet UILabel* lblEndpoint;
}
@end
```

First you import the ZBarSDK headers and declare `ViewController` as conforming to the `ZBarReaderDelegate` protocol. Then you list the outlets you need to connect to the UI – the image where you will show the scanned barcode, and the two labels.

Now switch back to **MainStoryboard.storyboard** and connect the outlets. Hold the Ctrl key and drag as shown in the diagram below. Choose the appropriate outlets from the popup menu. (Or, you can right-click on the view controller icon at the bottom and then select the relevant endpoints from the popup and drag from them to the controls.)

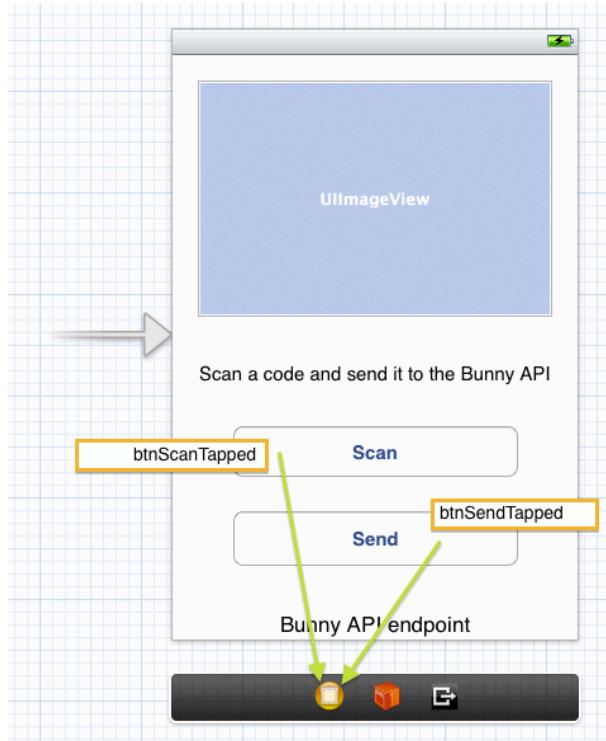


You also need to connect two action methods for when the two buttons are tapped. Add these two placeholders to **ViewController.m**:

```
- (IBAction)btnScanTapped
{
}

-(IBAction)btnSendTapped
{}
```

Now go back to **MainStoryboard.storyboard** to do the connecting. While holding the Ctrl key, drag with the mouse from the buttons to the View Controller as in the image below, and choose the respective method names from the popup menu:



OK, that's all the UI setup you need for this project! The Easter Bunny keeps it real. Go back to **ViewController.m** and paste this code into the `btnScanTapped` method:

```
ZBarReaderViewController *reader =
    [[ZBarReaderViewController alloc] init];
reader.readerDelegate = self;
reader.supportedOrientationsMask = ZBarOrientationMaskAll;

[self presentViewController:reader animated:YES completion:nil];
```

Believe it or not, that's all you need to do to add barcode recognition capability to your app thanks to the power of the ZBarSDK! Let's have a look:

1. `ZBarReaderViewController` is a view controller that presents the video feed from the camera to the user. It also shows an overlay UI when a barcode is recognized.
2. You set the delegate of the bar scanner view controller to be your own class.
3. You set `supportedOrientationMask` to all device orientations.
4. Finally, you present the view controller on the screen.

Go ahead! You can give it a try right now!

Compile errors or not?

If at this point your project compiles without errors and runs the app just skip this section and hop to the next one "Fetching the barcode".

If you are still reading this then at this point you're probably looking into the following error:

```
ld: file is universal (3 slices) but does not contain a(n) armv7s slice: /Users/marin/BunnyApp/ZBarSDK/libzbar.a for architecture armv7s
clang: error: linker command failed with exit code 1 (use -v to see invocation)
File is universal (3 slices) but does not contain a(n) armv7s slice: /Users/marin/Desktop/Projects/iOS/Passbook/BunnyApp/Build/Products/Debug-iphoneos/BunnyApp.app/BunnyApp
① Linker command failed with exit code 1 (use -v to see invocation)
② Create universal binary BunnyApp ...in /Users/marin/Library/Developer/Xcode/DerivedData/BunnyApp/Build/Products/Debug-iphoneos/BunnyApp.app
③ Command /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/ld ...in /Users/marin/Library/Developer/Xcode/DerivedData/BunnyApp/Build/Products/Debug-iphoneos/BunnyApp.app
④ Generating BunnyApp.app.dSYM ...in /Users/marin/Library/Developer/Xcode/DerivedData/BunnyApp/Build/Products/Debug-iphoneos/BunnyApp.app
```

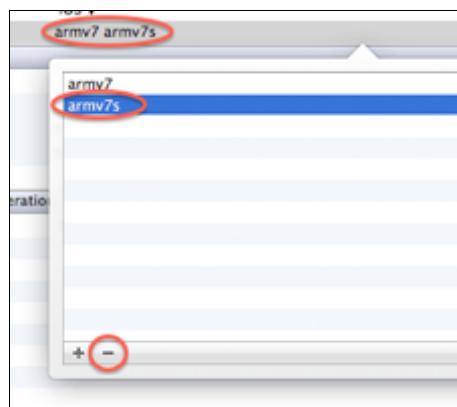
libzbar.a is a static library and therefore it must be compiled with support for every possible architecture your app will run on. At the time of this writing the guys from ZBar still didn't include support for the CPU of the new iPhone5. (Therefore the error from the image above.)

To solve this problem is pretty easy. For the time being (until the ZBar guys update the library properly) you can disable the new armv7s architecture for your project. Your app will still run on the new iPhone, but it won't use all the advantages of the new chip. Once ZBar supports armv7s – you can include again support for this architecture also for your app.

Open the project build settings and search for "architecture":



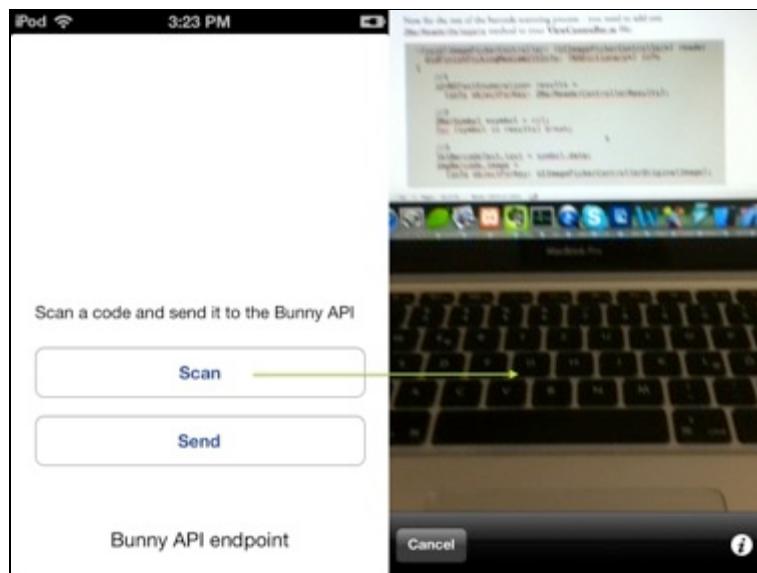
Than double click the row for Valid Architectures saying "armv7 armv7s" and you will be presented with the list of supported architectures. Select "armv7s" and then click the (-) button to remove it:



Now you should be able to compile and run with no problems. Don't forget to check ZBar web site for armv7s support and than also enable it in your app!

Fetching the barcode

Run the project on your device and tap Scan to see the barcode reader view controller. Neat!



If you actually point the camera at a barcode, you will not fail to observe that the app crashes. Xcode complains that a method is missing in the ViewController class.

You need to add a `ZBarReaderDelegate` method to your **ViewController.m** file for the scanning process to work:

```
- (void)imagePickerController: (UIImagePickerController*) reader
    didFinishPickingMediaWithInfo: (NSDictionary*) info
{
    //1
    id<NSFastEnumeration> results =
        [info objectForKey: ZBarReaderControllerResults];

    //2
    ZBarSymbol *symbol = nil;
    for (symbol in results) break;

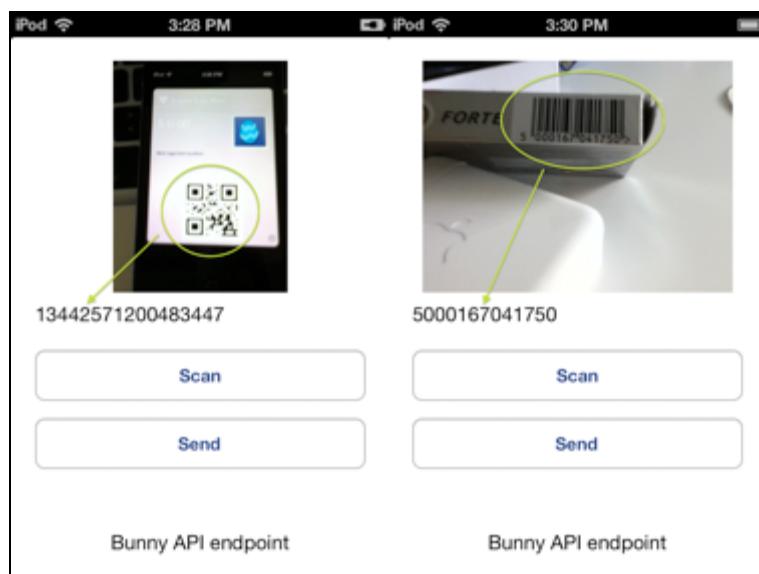
    //3
    lblBarcodeText.text = symbol.data;
    imgBarcode.image =
        [info objectForKey: UIImagePickerControllerOriginalImage];
    //4
    [reader dismissViewControllerAnimated:YES completion:nil];
}
```

```
}
```

The code is pretty easy:

1. First you get the returned scanner results from the `info` parameter. It's possible that there's more than one result coming back from the scanner.
2. Then you create a new `ZBarSymbol` object and you fetch the first result from `results` into it.
3. Now `symbol.data` contains the text that was originally encoded into the barcode, and you can fetch the photo of the barcode (if you need it) from the `info` dictionary as well.
4. After you update the UI with the data from the barcode being scanned, you dismiss the camera controller.

Hit the Run button and give the app another try – this time, when you point the device at a barcode, the camera overlay will disappear and you can see a photo of the barcode and the encoded message!



ZBar rulez!



OK, you are about halfway through the Bunny App – now you need to add network connectivity and make the app speak JSON with your server.

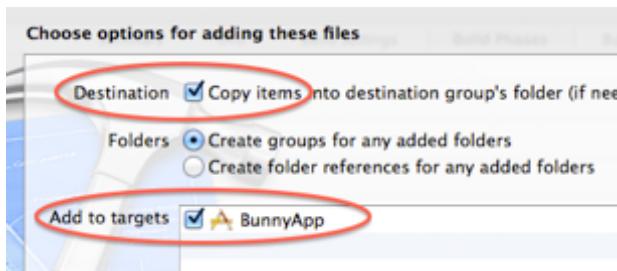
Talking to the Bunny API

Next you will add AFNetworking to your project – it's a handy networking library for iOS and OS X, so you'll use it to communicate in JSON with your server.

Head up to the AFNetworking github page:

<http://github.com/AFNetworking/AFNetworking/>, and download the ZIP version (spot the nifty button featuring a cloud and the word "ZIP" and click it).

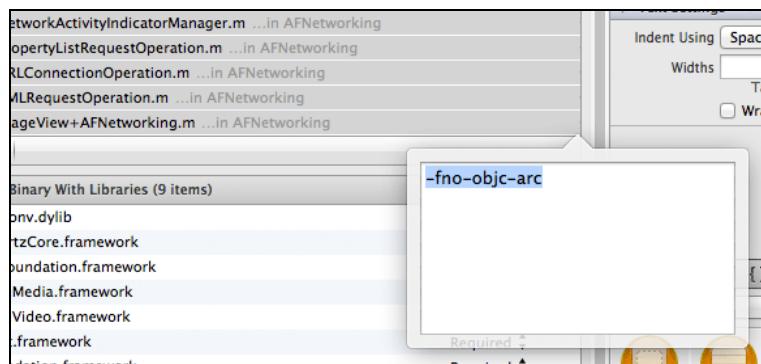
After you unarchive the downloaded ZIP file to a folder, have a look inside and copy the **AFNetworking** subfolder (inside the main AFNetworking folder) to your XCode project. Make sure you check the box to copy the files over to your own Xcode project folder:



AFNetworking currently does not support ARC, so you need to disable it for all the AFNetworking files. Select the project root in the project navigator and from the right pane, make sure you have selected the "Build Phases" tab. Expand the "Compile sources" section and select all the AFNetworking files (from AFHTTPClient.m through UIImageView+AFNetworking.m).

Hit the Enter key on your keyboard, and inside the popup window type:

```
-fno-objc-arc
```



Tap Enter again to close the popup window. Now you should be all ready to send over the scanned barcode to the server.

Open **ViewController.m** and add this import at the top of the file:

```
#import "AFNetworking.h"
```

Below the imports section, add these two defines so that you can change the configuration easily when you use the project for your own needs in the future:

```
static NSString* kBunnyAPIURL =
@"http://yourdomain.com/egghunt/bunnyapi/";
static NSString* kBunnyAPISecret =
@"askd873kjanf92n4ll298fkjwf923kjb235k23jb";
```

`kBunnyAPIURL` is the endpoint of the Bunny API on your server (it's still not implemented, but you'll get to it soon enough), so your Bunny iPhone app is going to send the scanned barcodes to that URL. Remember to change `yourdomain.com` to your own server address, as always.

If you want to secure the communications with the Bunny API, use an "https" URL, but remember that your server needs to support SSL.

`kBunnyAPISecret` is a simple authentication string used by the Bunny App. It will be sent, along with the JSON request, to the server to authenticate the app.

The iPhone app itself won't have any logic – the logic will be on the server and the iPhone will only present whatever response it gets via the Bunny API.

Let's do that – build a JSON request containing the decoded barcode message, and show the response from the server as an alert the message.

Add this line of code to the end of `viewDidLoad` so that you always know to which Bunny API endpoint you are posting:

```
lblEndpoint.text = kBunnyAPIURL;
```

Add this code to `btnSendTapped`:

```
NSURL *url = [NSURL URLWithString:kBunnyAPIURL];

NSMutableURLRequest *request =
[NSMutableURLRequest requestWithURL:url];

[request setHTTPMethod:@"POST"];
```

First you build an `NSURL` object out of the string containing your Bunny API endpoint. Then you create a mutable HTTP request for that URL. You then set the HTTP method to POST, as you will need to transmit data encoded as JSON.

Next add the JSON payload:

```
NSMutableDictionary* payload =
[[NSMutableDictionary alloc] initWithCapacity:2];
payload[@"barcode"] = lblBarcodeText.text;
payload[@"token"] = kBunnyAPISecret;
```

```
[request setHTTPBody:[NSJSONSerialization  
dataWithJSONObject:payload options:kNilOptions error:nil]];
```

You create a new mutable dictionary called `payload`. For the “barcode” key you set a string value containing the decoded barcode message, and for the “token” key you set the value of the shared Bunny API secret.

Finally, you call `setHTTPBody:` on the `request` object and set the POST request contents to be the JSON-encoded data from the `payload` dictionary.

All is ready for sending the request to the server. Add the relevant AFNetworking code to do this:

```
AFJSONRequestOperation *operation =  
[AFJSONRequestOperation JSONRequestOperationWithRequest:request  
  
success:^(NSURLRequest *request, NSHTTPURLResponse *response,  
NSDictionary* JSON) {  
  
    [[[UIAlertView alloc] initWithTitle:@"Response"  
message:JSON[@"msg"]  
delegate:nil  
cancelButtonTitle:@"Close"  
otherButtonTitles:nil] show];  
}  
  
failure:^(NSURLRequest *request, NSHTTPURLResponse *response,  
NSError *error, id JSON) {  
  
    [[[UIAlertView alloc] initWithTitle:@"Error"  
message:[error localizedDescription]  
delegate:nil  
cancelButtonTitle:@"Close"  
otherButtonTitles:nil] show];  
}];  
  
[operation start];
```

Let’s see what’s going on in this code:

1. `[AFJSONRequestOperation JSONRequestOperationWithRequest:success:failure:]` creates an operation for you, so you can execute it as a normal `NSOperation` running in the background. You supply the `request` (you already have it configured) and also blocks of code to execute on success and on network error.
2. Inside the success block you use one of the block arguments, namely `NSDictionary* JSON`, which contains the server response. You show a `UIAlertView`

with the text from the “`msg`” key of the server response. This way you strip the Bunny iPhone app of any logic. So, if you need to fix or change something quickly in the system, you can do it on the server side without needing any code changes for the app.

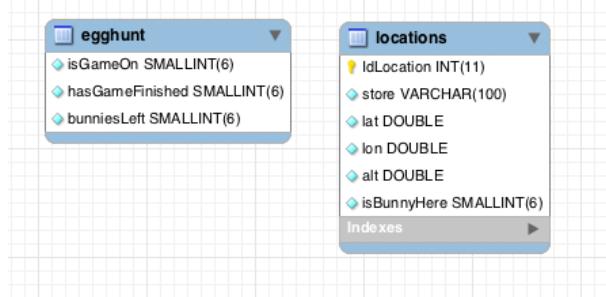
3. Inside the error block you show a `UIAlertView` with the description of the error as returned by AFNetworking.

Finally, you call the `start` method on the operation and it fires the request off to your server.

OK, that's all you want for now from the Bunny App. Let's move on to the server backend, which will handle the app requests, increase the user's credit, and update the bunny location in the database.

Implementing the game server logic

Let's go back to the project's database for a moment. There are two tables you haven't looked at so far:



The `egghunt` table contains the game's current state. The game will have three distinct states:

- **Not started** – this is the period during which the users download game passes, but the game hasn't started yet.
- **Active** – it's Easter Sunday and people are chasing the Easter Bunny around the mall to grab store credit bonuses.
- **Finished** – the bunny has been to all the game locations and all the bonuses have been given away. People are using their gained store credit to make purchases.

Before the game has started, the values in the `egghunt` table are:

```
select * from egghunt
```

| | isGameOn | hasGameFinished | bunniesLeft |
|---|----------|-----------------|-------------|
| 0 | 0 | 10 | |
| | | | |

During the time the game is active, the table looks like this (with the `bunniesLeft` column's value decreasing each time somebody finds the Easter Bunny):

| select * from egghunt | | |
|-----------------------|-----------------|-------------|
| isGameOn | hasGameFinished | bunniesLeft |
| 1 | 0 | 7 |
| | | |

When the last credit has been given to a game participant and `bunniesLeft` equals zero, then the game switches to a finished state, like so:

| select * from egghunt | | |
|-----------------------|-----------------|-------------|
| isGameOn | hasGameFinished | bunniesLeft |
| 0 | 1 | 0 |
| | | |

You will have to take care to verify the current game status and make sure, for example, to not give away credit after the game has already finished, or when the bunny is not hiding in a mall store.

The second database table is `locations`. It is pre-filled with 10 locations inside the mall – different stores that will take part in the game. You have the store name and also the store GPS coordinates – latitude, longitude and altitude. Finally you have a column called `isBunnyHere`, which, as the name suggests, indicates whether the Easter Bunny is currently to be found at that particular store.

You are going to implement a simple admin panel for the game administrators to use to change the game state, but first, let's finish the Bunny API!

The Bunny Server API

The backend for the Bunny App will be very simple – it will check if the game is active, look up the `locations` table to check where the bunny is, and finally increase the credit on the pass with the given barcode.

Switch back to web development mode and inside the `egghunt` folder on your web server, create a new sub-folder called `bunnyapi`. This is where you'll create the PHP script that will handle the requests coming in from the Bunny App.

The Bunny API will have a single task, so a plain PHP script will suffice. In short, what you need this script to do is:

- Read the raw POST request from the Bunny App and check if a valid authorization token has been sent over.
- Check whether the game is running right now.
- Find the pass that the barcode belongs to and increase the assigned store credit by \$25.
- Decrease the number of locations left for the game.

- Return a message to the Bunny App saying “Credit increased to \$XX.xx.”
- And finally (and most exciting of all), update the user’s pass with the new amount of store credit! Sweet!

Let’s start adding pieces of code and get the Bunny API running step-by-step!

Inside the **bunnyapi** folder, create a new file and call it **index.php**. You will be returning a JSON response, so first of all let’s add the proper HTTP header to instruct the client what the output will be:

```
<?php  
//you'll be returning JSON data  
header("Content-type: application/json");
```

OK, next on your to-do list is reading the POST data and comparing the token sent to the one you have on the server side. Add the code to do this:

```
//the bunny app shared secret  
$bunnyToken = "askd873kjanf92n411298fkjwf923kjb235k23jb";  
  
//read the POST payload  
$payload = json_decode(file_get_contents('php://input'), true);  
  
//read the token sent from the client  
$token = @{$payload["token"]};  
  
//check the validity of the token  
if ($token!=$bunnyToken) {  
    print json_encode(array("msg"=>"Not authorized"));  
    exit();  
}
```

By now I suspect you are already a PHP pro, but let’s quickly go over the code:

1. `$bunnyToken` contains the shared secret between the Bunny iPhone App and the Bunny API server backend.
2. As before, read the POST request body by using `file_get_contents('php://input')` and try to decode the content, expecting it is JSON-encoded data.
3. If the POST request contained proper JSON data from the Bunny App, `$payload["token"]` will be the authorization token sent over, so you store it in the `$token` variable.
4. Finally, compare the token sent with the one defined in the script, and if they differ, send back a JSON-encoded dictionary containing an error message.

Now that you have the client authorized, you can check the database to see if the game currently allows interactions with the Bunny App. You need to check whether

at the moment the bunny is hiding somewhere in the mall and is expected to scan somebody's pass.

Note: Why do you need to do that? Well... it's not that you don't trust the Easter Bunny. It's just that... checking one more time doesn't hurt. How about if the Easter Bunny "by mistake" scans his cousin's pass several times and bumps his credit up by a few hundred dollars? Better to double-check the database, don't you think?



All the game state information is stored in the `egghunt` table. So you need to open a connection to the MySQL database and read the only row of information saved inside that table.

Add this code, which you should recognize:

```
//get a database connection
require_once("../class/Database.php");
$db = Database::get();

//check whether the game is currently active
$statement = $db->prepare("SELECT * FROM egghunt LIMIT 1");
$statement->execute();
$egghunt = $statement->fetch(PDO::FETCH_ASSOC);
```

First you include the `Database` class and get an instance of it.

Then you create a prepared statement to select the first (and only) row of the `egghunt` table and execute it. You fetch the data read in to `$egghunt`. At this point you should have the default values from the `egghunt` table:

| Column | Value |
|------------------------------|-------|
| <code>isGameOn</code> | 0 |
| <code>bunniesLeft</code> | 10 |
| <code>hasGameFinished</code> | 0 |

In order to continue with adding extra credit to the target pass, you first need to make sure that the game is currently active (so no credit is added before the game started, for example). Just add this simple `if` statement:

```
if ($egghunt['isGameOn']!=1 || $egghunt['hasGameFinished']!=0) {
    print json_encode(array(
        "msg"=>"The Egg Hunt Game is not active at the moment"
    ));
    exit();
}
```

If the game state is not the right one, you return an error message encoded as JSON to the client and exit. It's nothing personal – you just don't want the system to leak out store credit, right?

The next step is to check whether the Easter Bunny is actually hiding at this very moment in a store in the mall. Let's check if there's a store in the locations table with the value of 1 in its `isBunnyHere` column. A simple SELECT to the `locations` table will suffice:

```
//check if a bunny is in a store right now
$statement = $db->prepare("SELECT * FROM locations
    WHERE isBunnyHere=1");
$statement->execute();
$currentLocation = $statement->fetch(PDO::FETCH_ASSOC);

if (!$currentLocation) {
    print json_encode(array(
        "msg"=>"There's no active location at the moment"
    ));
    exit();
}
```

First you prepare a statement to select all rows from the `locations` table where `isBunnyHere` equals 1. Then you execute it and try to fetch one row from the result set into the `$currentLocation` variable.

If there is no data fetched in `$currentLocation`, then you again return a JSON-encoded error message to the client and exit the script execution.

OK, so far you've done several validations and you are now sure that the script can go on with updating the user's credit.

Add an UPDATE statement to increase the user's credit:

```
//update the credit amount of the pass
$barcode = $payload['barcode'];
```

```
$statement = $db->prepare("UPDATE passes
    SET credit = credit + 25, lastUpdated = ?
    WHERE barcode = ? LIMIT 1");
$statement->execute(array(time(),$barcode));

if ($statement->rowCount() == 0) {
    print json_encode(array("msg"=>"Pass not found"));
    exit();
}
```

First you fetch the barcode value from the decoded POST data and store it in `$barcode`. Then you prepare a database statement that increases the `credit` column by 25 and also updates the `lastUpdated` column with the current timestamp. You then execute the statement, and if all goes well, the one pass with the provided barcode value will get its credit bumped up by \$25.

At the end of this piece of code, you do yet another check: if the statement's number of affected rows equals zero, that means that the barcode value didn't match any pass records in your `passes` table. In this case, the Bunny App sent you a barcode that does not belong to a pass registered for the Easter Egg Hunt.

It's nice to return a proper error message (very helpful for debugging the system as well) – so you do just that: return a "Pass not found" error and exit the script.

The next step is to reset the bunny's location, since only one user per location will get the store credit. You can just update all locations to have their `isBunnyHere` column to set to 0:

```
// indicate the bunny has been found at the current location
$db->prepare("UPDATE locations SET isBunnyHere=0")->execute();
```

Aha! This line looks somewhat familiar, but still something is wrong with it. No, actually nothing is wrong! In order to save an extra line of code, you chain call methods – another relatively recent PHP feature.

As you already know, the `prepare()` method returns an instance of a prepared statement. Then you usually store the statement in a variable and call `execute()` on that variable. If the statement is short and you don't need to fetch a result set back, then you might as well chain the method calls, as above.

Now to implement a bit of the game logic in the script – if the number of locations left for the game is zero, you need to set the proper game state and put the Easter Hunt to an end. If there are still stores for the bunny to hide in, you just need to decrease the locations left.

Add the code next:

```
//update game state
```

```

if ($egghunt['bunniesLeft']>0) {
    $statement = $db->prepare("UPDATE egghunt
        SET bunniesLeft=bunniesLeft-1")->execute();
} else {
    $statement = $db->prepare("UPDATE egghunt
        SET hasGameFinished=1,isGameOn=0")->execute();
}

```

In the `if` branch of the code (meaning there are still stores left for the bunny to hide in) you prepare and execute a statement that just decreases the value of the `bunniesLeft` column in the `egghunt` table.

In the `else` branch (executed when the bunny was found at the last location participating in the game), you set `hasGameFinished` to 1 and `isGameOn` to 0.

This is pretty handy, because the software at the cash desks in the mall can now check if `hasGameFinished` is 1, and in that case, will begin accepting the store cards of the Easter Egg Hunt participants for purchases of goods.

Note: You are not going to actually implement the software for the mall cash registers in this chapter, as it is already at epic length, but hey... you now have all the skills you need to implement it! Connecting to the database, preparing and executing statements, etc. – it all has been covered, so you could develop the project further yourself.

Now, after the game state has been changed and the pass has been updated, you might as well return the response to the Bunny App! You need to fetch the current pass data from the database and then you can return it to the app wrapped in a JSON response.

First fetch the pass record from the `passes` table:

```

//read the pass data
$statement = $db->prepare("SELECT * FROM passes
    WHERE barcode = ? LIMIT 1");
$statement->execute(array($barcode));
$pass = $statement->fetch(PDO::FETCH_ASSOC);

```

Nothing new in this piece of code – you just fetch the record, which has a matching `barcode` value, and you store the data in the `$pass` dictionary.

Then add the code to return the JSON response to the client:

```

//return response
print json_encode(array(
    "msg"=>"Participant credit successfully increased to ".
)

```

```
$pass['credit']  
));  
flush();
```

Again as with the errors (but this time with a more cheerful message), you create a new dictionary and return a message that the Easter Bunny should see popping up on the screen after scanning a pass. You return “*Participant credit successfully increased to X,*” so the Easter Bunny can also congratulate the user and tell them their new credit amount.

Finally, you flush the response to the Bunny App.

Tip: In the script above, you perform quite a few database operations, one after another. In a concurrent heavy load system this might become a problem after a certain point of time. If you would like to develop the project further, definitely have a look at how to wrap all consequent database queries in one atomic operation – a MySQL transaction.

PHP+MySQL examples: <http://stackoverflow.com/questions/2708237/php-mysql-transactions-examples>

PHP PDO documentation: <http://php.net/manual/en/pdo.beginTransaction.php>

MySQL documentation:

<http://dev.mysql.com/doc/refman/5.0/en/commit.html>

Well done! At this point the Bunny API is almost complete. You have implemented all the logic you need. So take a break and get yourself a cookie. Or if you’re rearing to go, head right to the last and most interesting part of the Bunny API – initiating a conversation between your server and the user device.

Talking back to PassKit

So far you’ve updated the user’s pass and increased its credit by \$25. That’s great, but if the user were to open their Passbook app, they would still see \$0 of store credit. (Or if you ran successfully the update tests from earlier – 1\$ of store credit.)



If you think about it for a moment – you already implemented all the necessary functionality to update passes in your PassKit web service, right?

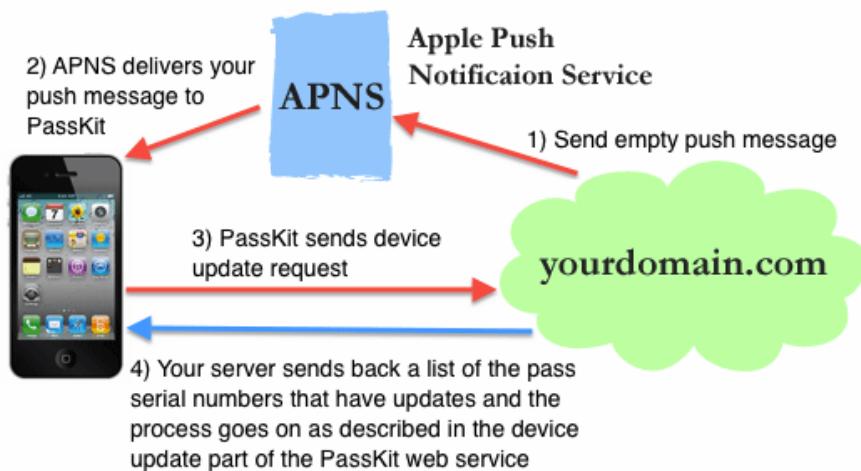
If the user were to pull down while viewing the back of your pass, PassKit would send an update request to your web service and get the latest version of the pass – and the user will be able to see the updated credit value on the front of the pass.

If only there was a way your server could ping the user's iPhone and let it know there are updates to the pass. Then PassKit on the iPhone will know how to fetch the list of updates, and also how to request all of them one-by-one from your server.

Yes, yes... I know you've guessed it – there is such a way! ☺

The process is actually pretty simple, and you will see that it doesn't take long to implement.

Whenever your server has new pass information, you can send the user's device a push notification, and PassKit will then take care of talking back to the server and getting the updates to the user's passes.



To send push notifications to the Apple Push Notification Service (or APNS) you will create a new class, featuring a couple of handy methods.

Create a new file inside your project's **class** folder and call it **APNS.php**.

Let's perform one additional step of preparation before beginning to code the class. You probably know that Apple requires you to have an Apple-issued Push Notification certificate for each of your applications. But wait... you don't have an app to send your push notifications to! In fact – you want to send push notifications to PassKit, so it can ask for updates from your server. Oh well...

You already have a certificate and your key – you use them to create DER signatures for your pass bundles. Apple accepts the same certificate and key for connecting to the APNS and for sending out push notifications. Great!

So, what is the extra preparation step, then? You will need to bundle the certificate and the key .PEM files together, so you can use the resulting file from PHP to connect through SSL to the APNS. Luckily, that's not difficult at all.

Open up Terminal, and navigate to the **egghunt/pass** subfolder of your project. Type `ls -al` and hit Enter, and you should see `certificate.pem` and `key.pem` just sitting there and begging to be bundled.

Enter the following command:

```
cat passcertificate.pem passkey.pem > passcertbundle.pem
```

This will create a new file, **passcertbundle.pem**, which simply contains the concatenated text from the other two files.

Now you can go ahead with implementing the class to connect to the APNS.

Open **class/APNS.php** and paste in the initial structure and configuration that you will need:

```
<?php
require_once( "Database.php" );

class APNS {

    //config for generating passes
    private $keyPath = 'pass';
    private $keyPassword = "12345";

    //apple push service endpoint
    private $apnsHost = 'ssl://gateway.push.apple.com:2195';

}
```

First, you include the `Database` class, as you will need to interact with the database from the current class.

Then you define the `APNS` class and add its configuration elements. You need the `$keyPath` and `$keyPassword` properties to access the `passcertbundle.pem` file that you just generated. (These properties contain the same file paths you already used in the `EggHuntPass.php` file.)

The `$apnsHost` property stores the APNS endpoint you are going to connect to.

Note: If you've developed apps that use push notifications, then you're used to using the APNS Sandbox server while in development. However, when

developing passes you should use the live server directly, since at the time of writing the sandbox server does not relay push notifications for passes.

Next, add the class constructor just before the final closing bracket of the class definition:

```
function __construct() {
    $this->keyPath = realpath(dirname(__FILE__) . "/../../".
        $this->keyPath);
}
```

This is exactly what you did before to convert the `$this->keyPath` property from a relative path to an absolute path.

You will have two similar, but sufficiently different, tasks for the APNS class:

1. Given a pass serial number, find all devices that have it installed and send them a push notification. This task is, for example, when a pass needs to receive a \$25 store credit increase.
2. Send a push notification to all devices with installed passes. This task is performed when the bunny location changes – you will update the `locations` table and then ping all devices via APNS, so that they can pull the latest pass version from your server.

First you will create the method for the second task, to update all devices. Add the method body inside the `APNS` class:

```
//update all registered devices
function updateAllPasses()
{
    $db = Database::get();
    $statement = $db->prepare("SELECT pushToken FROM devices");
    $statement->execute();
    $this->sendPushesToResultSet($statement);
    return $statement->rowCount();
}
```

First you get an active connection to the database, and then you use it to create a prepared statement that fetches all the device push tokens from the `devices` table. You execute the statement and then pass it to a method called `sendPushesToResultSet()`. Ah! This mysterious method will send the actual push notifications. On the last line, you return the count of rows the statement fetched from the database.

OK – pretty simple; let's quickly go on with implementing the method for the first task, which is not so different from the one above. Add it to the class body:

```
//update all devices which have the given pass installed
function updateForPassWithSerialNr($serialNr)
{
    $db = Database::get();
    $statement = $db->prepare("SELECT pushToken FROM devices
        WHERE serialNr=?");
    $statement->execute(array($serialNr));
    $this->sendPushesToResultSet($statement);
}
```

The method argument is a pass serial number, which you use to fetch all devices with the matching pass installed. Again you execute the SELECT statement and push the result set to the `sendPushesToResultSet()` method.

And now onto the mysterious push notification-sending method.

Note: This method will be heavily based on my colleague Matthijs Hollemans's tutorial on implementing push notifications. If you are interested in learning more about how to implement push notifications in your iOS apps and server, be sure to read his tutorial: <http://www.raywenderlich.com/3443/apple-push-notification-services-tutorial-part-12>.

There's actually not much mystery involved in the method itself – it just uses some PHP functions that you don't really need in everyday web-development. You'll have to implement two separate tasks in the method:

1. Open a new SSL socket connection to the APNS server and provide your certificate and key to identify yourself.
2. Loop over the result set of the statement provided, and send an empty push message to each of the devices found in the set.

Let's add the code to implement the first task to the end of the class body:

```
// 1
private function sendPushesToResultSet(PDOStatement $stmt)
{
    // 2
    if ($stmt->rowCount() == 0) return;

    // 3
    $ctx = stream_context_create();

    // 4
    stream_context_set_option($ctx, 'ssl', 'local_cert',
        $this->keyPath."/passcertbundle.pem");
    // 5
```

```
stream_context_set_option($ctx, 'ssl', 'passphrase',
    $this->keyPassword);

// 6
$fp = stream_socket_client($this->apnsHost, $err, $errstr,
    15, STREAM_CLIENT_CONNECT|STREAM_CLIENT_PERSISTENT, $ctx);

// 7
if (!$fp) return; //TODO: add error handling

}
```

Quite a lot of new PHP goodness in this method, so let's go step-by-step over the code:

1. First you declare the `sendPushesToResultSet()` private method – note how this time, you also declare the type of the argument. Yes – in PHP you can also define the expected method argument type, but only if it's an object or an array. So, since you only expect `PDOStatement`, you declare the required argument class.
2. On the first line of the method code, you make a quick check – if the statement result set is empty, there's no need to do anything further.
3. Next you get a stream context out of `stream_context_create()`.
4. You need to configure the stream context, and you do it with `stream_context_set_option()`. You assign the file path to the bundled .PEM file with your certificate and key to the option with name "local_cert".
5. You set another stream context option, called "passphrase", to your key's password.
6. Finally, you call `stream_socket_client()` to connect to the APNS server. You pass in the following arguments:
 - a. The URL to connect to;
 - b. Two additional variables to read back the error and error message if there's a connection error;
 - c. The connection timeout in seconds;
 - d. The additional flags for the connection;
 - e. The socket stream context.
7. Then you check whether the socket connection variable is `null` – if so, the connection failed. At this moment you can use the information stored in `$err` and `$errstr` to understand why the connection didn't go through.

Note: Your script makes a connection to port 2195 of the APNS server. Most shared hosting packages won't allow outgoing connections to custom ports; if that's the case on your server, PHP will output a warning to tell you the

connection didn't succeed. If you experience problems connecting, use the `$err` and `$errstr` variables, where the "add error handling" comment is in the code. These should provide you with more information.

Tip: If you're considering buying a virtual server, have a look at the offers from <http://www.linode.com> – they have good prices and excellent feedback about their tech support.

Now that you have an active connection to the APNS server, you can send through all the push notifications one after another.

First create an empty dictionary object and JSON encode it to set the contents for the push message. Add this inside the method before the closing bracket:

```
//create an empty push
$emptyPush = json_encode(new ArrayObject());
```

Note: Why do you send an *empty* push notification (and not some kind of a message)? Usually, when you send a notification to your application it's because you need to show a text message to the user, or send a piece of data to your app. However, Apple does not guarantee the delivery of those push notifications: the servers might be busy, the user might be out of reach, or there might be another issue.

Pass updates, however, have a **delivery guarantee**. This is one of the reasons to keep the push message as small as possible. But also, you don't need to send a piece of data or a message. You just need to ping PassKit as a way to say: "I've new info, ask me more about it when you've got some time."

All that's left to do in the `sendPushesToResultSet()` method is to loop over the `$stmt` statement result set and send a push message through the active APNS connection. Paste in the code to do that to the end of the method:

```
//send it to all devices found
while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    //write the push message to the apns socket connection
    $msg = chr(0). //1
    pack("\n",32).
    pack('H*', $row['pushToken']). //2
    pack("\n",strlen($emptyPush)). //3
    $emptyPush; //4

    fwrite($fp, $msg);
}
```

```
//close the apns connection
fclose($fp);
```

Much as you did before, you loop over the result set, fetching every row from the set into the dictionary variable `$row`.

Then in the `while` loop body, you build a push message for each of the recipient devices and store it in `$msg`. Let's have a look at what goes into a push notification:



If you look at the code again, this is exactly what you are storing into the `$msg` string variable:

1. You start off with a null character, which is the command for sending a simple push notification.
2. Then you pack a big endian value of 0 and 32 bytes to set the token length, and the string token value stored as 32 bytes.
3. You then append the packed length of the empty push message payload.
4. And at the end, you append the actual JSON-encoded payload.

After you generate the correct message for each recipient, you call `fwrite()` to actually write the data to the open socket connection to APNS.

When the `while` loop finishes, you are done with sending out data, so you call `fclose()` to close the connection and wrap up.

This is all the code that goes into the `sendPushesToResultSet()` – the method is ready to fire some pushes around the globe!



Wrapping up the Bunny API

Switch back to the **bunnyapi/index.php** file; you left things somewhat unfinished there. You updated the user's pass and the game state, but didn't send out a notification to the user's device to let PassKit know that something on the server changed.

But you can do that easily now that you have completed the APNS class! At the end of the **index.php** file (just after the `flush()` line) add:

```
//update the pass
require_once("../class/APNS.php");
$apns = new APNS();

if ($egghunt['bunniesLeft']==0) {
    //update all passes, the game is over
    $apns->updateAllPasses();

} else {
    //update the current pass only with the new credit
    $apns->updateForPassWithSerialNr($pass['serialNr']);
}

exit();
```

Ah, the code looks pretty straightforward!

You include the APNS.php file and make an instance of the APNS class. Then you check if there are any remaining locations for the bunny to hide. If not, then the game is over, so you send a push to all registered devices to update their passes with a message that the Easter Egg Hunt has finished and they now can spend the gained credit in the mall stores.

If the game is still active, then you only send a push notification to the pass that was scanned by the Bunny App. PassKit will then ask your server for available updates and get the new copy of the pass with the increased amount of store credit.

Finally, you call `exit()` to quit the execution of the script. And that's a wrap for the Bunny API backend application!

Congratulations! Let's give the Bunny API a try!



The Bunny App in action

Don't forget to update your web server with all the modified and new files.

First you will need to update the egghunt table to put the game in active status. Execute this SQL query:

```
UPDATE egghunt SET isGameOn=1, bunniesLeft=10, hasGameFinished=0
```

Next (just to make sure the test goes fine) delete all the Easter Egg Hunt passes from your Passbook app and download a fresh one.

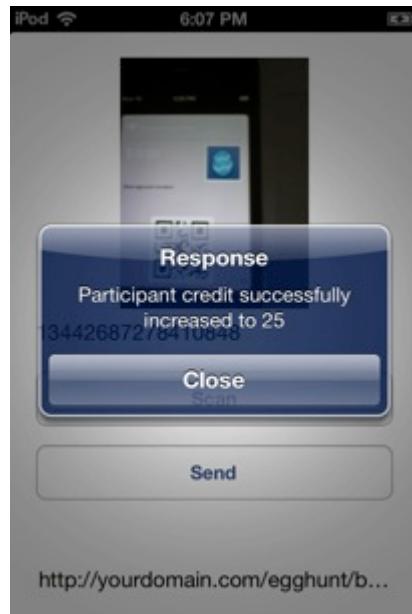
Now set the current Easter Bunny location:

```
UPDATE locations SET isBunnyHere=1 WHERE IdLocation=1
```

OK, time to see if everything works out in your grand scheme! Open the Bunny App on one device and your Easter Egg Hunt pass on another device.

Note: If you don't have two devices at hand, you can also take a screenshot of the pass on your Device and mail it to your computer – then open the screenshot on your screen and scan it with your iPhone/iPod.

Scan the barcode on your pass with the Bunny App and send it over to the Bunny server API. You should see a success message, like this:



Note: If you get compile or run-time errors from the AFNetworking code, delete the version of AFNetworking you just downloaded and replace it with the version included with this chapter's file resources.

If you check the database, you will see that the value of the `bunniesLeft` column has decreased to 9.

You will also get an update on your device for the pass value increase, similar to the following:



W00000000t!!! Also you can see your pass updates inside your Notification center:



The Easter Egg Hunt control center

The game administrators need a simple web interface to control how the game progresses. You are going to create a quick form where they can start or stop the game, or change the current bunny location within the mall.

Inside the **egghunt** folder, create a new folder and call it **admin**. Inside the **admin** folder, create a new file named **index.php**. This page will show the current state of

the game and the store where the Easter Bunny is hiding. It will also provide a UI to control the game.

Let's start coding this bad boy! Add a bit of code to begin with:

```
<?php  
//disable browser caching for this page  
header("Cache-Control: no-cache, must-revalidate");  
header("Expires: Sat, 03 Jul 1971 05:30:00 GMT");  
  
require_once("../class/Database.php");  
$db = Database::get();
```

The first HTTP header you send out instructs the browser not to store any cached versions of the page. The second header sets an expiration date way back in the past for the page content. This way, the browser always thinks that the content has expired and therefore will always try to grab a new copy from the server.

Then, in the now familiar order, you include the `Database` class and create an instance of it to use in your script.

Add this piece of code to fetch whether the game is active right now:

```
//get the game state from the egghunt table  
$statement = $db->prepare("SELECT * FROM egghunt");  
$statement->execute();  
$properties = $statement->fetch(PDO::FETCH_ASSOC);
```

You prepare a database statement and when you execute it, you fetch the entire game configuration – the number of game locations remaining, whether the game is active or finished. You store all these in a dictionary called `$properties`.

Then you go on to fetch the list of game locations:

```
//get the locations list  
$statement = $db->prepare("SELECT * FROM locations");  
$statement->execute();  
$locations = $statement->fetchAll(PDO::FETCH_ASSOC);
```

Again – you prepare a database statement and execute it; the `SELECT` query fetches the list of all game locations. You use `fetchAll()` on the statement to fetch the whole list of locations at once and store it in the `$locations` array.

Finally, the last piece of info you need – the location (if any) where the bunny currently hides:

```
//get the current location from the locations table
```

```
$statement = $db->prepare("SELECT * FROM locations  
WHERE isBunnyHere=1");  
  
$statement->execute();  
$currentLocation = $statement->fetch(PDO::FETCH_ASSOC);  
?>
```

When you execute the SQL query and fetch the data from the statement, you will have the current location data stored in the `$currentLocation` dictionary. If the bunny is not hiding in a store right now, `$currentLocation` will have a `null` value.

This is all the data you need, so you can continue with building the user interface. Onto HTML coding! (And some PHP code to fill in the proper dynamic values in the page template.)

Let's put in the `<head>` tag – you are going to define a bit of CSS and add some JavaScript. Add the following code to the end of the `index.php` file (after the closing `?>` PHP tag):

```
<html>  
<head>  
    <style>  
        form {border:1px solid #999; padding:10px; width:450px;  
              font-family: "Myriad Pro", Arial;}  
        div {color:green; font-family: "Myriad Pro", Arial;}  
    </style>  
    <script>  
        //refresh the page so the admin can see game updates  
        setTimeout("document.location.reload()", 15000);  
    </script>  
</head>
```

Let's see what's going on here. Within the `<head>` tag you define two other tags, `<style>` and `<script>`. In the `<style>` tag you add a few simple CSS definitions to style the forms and the text elements on the page a bit – nothing really complex, just setting the font family, text color, text size, etc.

In the `<script>` tag you add a line of JavaScript, which will reload the page every 15 seconds, so that the page always shows up-to-date information:

Next, you are going to start implementing the `<body>` content. As you did before, at the top of the page you will show any message that comes through the URL query string. Add the `<body>` opening tag and a block to show an incoming message:

```
<body>  
    <!-- show messages sent via GET -->  
    <? if ($_GET['message']!='') { ?>
```

```
<div><?=$_GET['message']?></div>
<? } ?>
```

You check if there's a value for the `message` key in the `GET` request – if so, you print it to the HTML code of the page.

Next you'll add a form that allows the user to turn the game ON/ OFF as needed. This form should be visible only if the game hasn't finished yet.

```
<!-- turn the game ON and OFF -->
<? if ($properties['hasGameFinished']==0) { ?>
<form action="setProperty.php?property=isGameOn" method="post">
  <? if ($properties['isGameOn']==1) { ?>
    <b>The Egg Hunt Game is ON</b>
    <input type="hidden" name="newValue" value="0"/>
    <input type="submit" value="Set Game to OFF"/>
  <? } else { ?>
    <b>The Egg Hunt Game is OFF</b>
    <input type="hidden" name="newValue" value="1"/>
    <input type="submit" value="Set Game to ON"/>
  <? } ?>
</form>
<? } ?>
```

You wrap the form contents in an `if` statement, which checks the current game state. There's no need for this form if the game has already finished.

Then you set the `<form>`'s `action` property to `setProperty.php` and the form method to "post". When the user submits the form, the data will be submitted through a POST request to the target file. There's also one extra piece of info in the `action` property – a parameter called "property" will be passed over `GET` to the target file. Its value will be "isGameOn".

Further inside the `<form>` tag, you create another `if` statement: if `isGameOn` is set to "1" in the database, then you show the proper text telling the user that the game is currently active, as well as a button to turn it OFF. If the game is not active at the moment, you show some text telling the user, as well as a button to turn it ON.

You can now open the <http://yourdomain.com/egghunt/admin/> URL and have a look at what you've done so far:



Note: The above HTML is not complete at the moment and there are several tags (`<body>` and `<html>`) that have not been closed. But most browsers these days are forgiving of such things, and the page should render properly.

Also, if the above does not work for you, your web server might not be set up to allow PHP short tags like "`<?>`". If that is the case, replace all instances of "`<?>`" with "`<?php`" and all instances of "`<?=`" with "`<?php echo`".

Looking good! Let's create the script that interacts with the database and changes the game state. Inside the **admin** folder, create a new file called **setProperty.php**.

First let's do the initial setup as for the previous file, add the caching HTTP headers code, and also get a connection to the database. Paste the following inside the empty file:

```
<?php
//disable browser caching for this page
header("Cache-Control: no-cache, must-revalidate");
header("Expires: Sat, 03 Jul 1971 05:30:00 GMT");

require_once("../class/Database.php");
$db = Database::get();
```

Next check whether the `$_GET` dictionary contains a key called `property` and if its value is `isGameON`. Now implement the code to change the game state by adding this at the end of the file:

```
//turn the game ON or OFF
if ($_GET['property']=="isGameOn") {

    $statement = $db->prepare("UPDATE egghunt SET isGameOn=?");
    $statement->execute(array((int)$_POST['newValue']));

    header("Location: index.php?message=isGameOn set to ".
        $_POST['newValue']."!");
    exit();
}
```

Inside the `if` statement you create a new prepared statement that updates the `egghunt` table and sets the `isGameOn` column to the new value sent over with the POST request. `$_POST['newValue']` is either 0 or 1, and is sent over from the `index.php` form.

When you're finished with the changes to the database, you send a `Location` header that redirects the user back to `index.php` with a `message` parameter added in the query string. This message will then be shown in `index.php`.

Cool! Now you can open <http://yourdomain.com/egghunt/admin/> again and press the button a few times to check how the game status changes:



Sweet! You've given the game administrators the ability to start the game on Easter Sunday, and if there are any unforeseen circumstances during the game, then they can also temporarily stop and then restart it.

Switch back to `index.php` and at the end of the code, add this:

```
<!-- show the remaining locations for the Easter bunny -->
<div><?= $properties['bunniesLeft'] ?> locations to go.</div>
```

This code will insert a message like "10 locations to go." This will keep the admin panel user up-to-date on the game progress. The `bunniesLeft` column in the `egghunt` table contains how many locations are left.

What's next? If the game is running at the moment you, you want to:

- Show a message saying: "The Bunny is currently at <Store Name>," when the Easter Bunny is currently hiding in a store.
- Or, show a drop-down box with the list of game locations, so the game administrators can choose and set the next location where the players should look for the Easter Bunny.

Add to the end of the code:

```
<!-- show list of locations or the current bunny location -->
<? if ($properties['isGameOn']==1 &&
      ($properties['hasGameFinished']==0)) { ?>

  <? if ($currentLocation) { ?>
  The Bunny is currently at "<?=$currentLocation['store']?>".

  <? } else { ?>
  <form action="setProperty.php?property=nextLocation"
        method="post">

    <select name="nextLocation">
      <? foreach ($locations as $l) { ?>
```

```

<option value=<?=$1['IdLocation']?>>
    <?=$1['store']?>
</option>
<? } ?>
</select>

<input type="submit" value="Activate next location"/>
</form>

<? } ?>
<? } ?>

```

Here's what this code does:

1. In the first `if` condition, you check whether the game is active (if the game isn't active or it has finished, you don't need to show this whole block of HTML at all).
2. If there was a location fetched from the database in the `$currentLocation` variable, then the bunny is currently hiding in a store, so you just print a message as to the bunny's location.
3. If `$currentLocation` is null, you create another `<form>` tag, which submits again to `setProperty.php`, but this time the `property` `GET` parameter has a value of "nextLocation". In this second form, you include a `<select>` tag to show a dropdown list on the page, and you loop over all locations in order to add an option for each store to the dropdown.

Finally, close the HTML tags on the page and you're done with the `index.php` code. At the end of the file add:

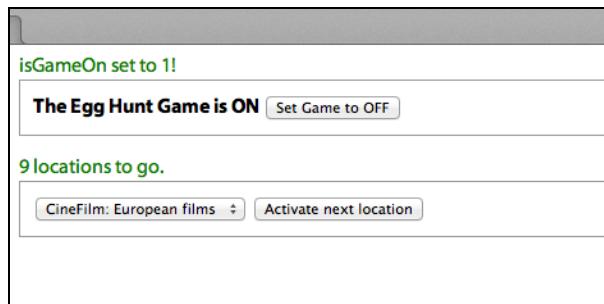
```

</body>
</html>

```

The admin panel is now complete!

Open <http://yourdomain.com/egghunt/admin/> again in your browser, and you should see the two forms on the page:



If you don't see the second form, check whether the current game state is ON. If the game is inactive, you won't see the list of locations – no worries, just click "Set

Game to ON” and you should see the list with all game locations and a button to activate one of them.

Of course, if you try to activate a new location at this point, it won’t work, since you haven’t added the code to handle the second form yet. ☺

So, let’s switch back to **setProperty.php** and add the code to handle changing the current location of the bunny. The code you need to implement has to perform three tasks:

1. Set the `isBunnyHere` table column to 1 for the location the user chose from the form.
2. Update the `lastUpdated` column of all the passes to the current timestamp.
3. Send a push message to all registered devices, so that all of them can ping your server back and download an updated version of the pass they already have installed. This up-to-date copy of the pass will show the users the new location of the game.

At the end of the file, add:

```
//set the next location the bunny can be found
if ($_GET['property']=="nextLocation") {

    //set the new bunny location
    $statement = $db->prepare("UPDATE locations
        SET isBunnyHere=1 WHERE IdLocation=?");
    $statement->execute(array((int)$_POST['nextLocation']));
```

The `if` statement checks whether the value “`nextLocation`” has been set in the `GET` request for the variable `property`. If yes, you prepare a new database statement to update the `locations` table and set the `isBunnyHere` column to 1. Then you execute it and pass the `$_POST['nextLocation']` variable (the location ID, selected via the form drop-down list) to update the correct record in the `locations` table.

Now add the next step from the task list above:

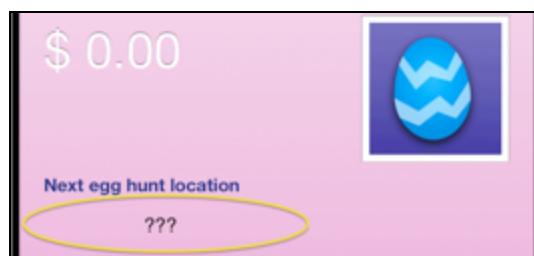
```
//set lastUpdated on all passes to now
$db->prepare("UPDATE passes SET lastUpdated=?")
->execute(array(time()));
```

You create and execute a database statement in one go. Then you just update the `lastUpdated` column with the value of the `time()` function of PHP – the current timestamp. This way, you set all passes to be in just-updated state, so when you ping PassKit on all devices and PassKit pings the server back, it will get a fresh copy of each installed pass.

Note: Don't try running any of this yet. If you haven't noticed, the last bit of code added to setProperty.php doesn't have a closing curly bracket, and this will break things on the server. ☺ The closing curly bracket (as well as the rest of the code) is coming as soon as we sort out a few things.

Showing the next location on the pass

At this point, you're missing only one important piece of the puzzle. You still don't show the next location on the front of the pass. No matter how many times you request an update from the server, the pass always shows an empty "Next location" field:



So your next mission is to implement a new method in EggHuntPass.php to determine the value of the location field and include it in the pass.json for the pass.

With all the system knowledge you've earned so far, it should be a rather simple task. ☺ So let's go!

Open **class/EggHuntPass.php** and add a new method (before the final curly bracket):

```
//get the data for the location field of the pass
//according to whether the bunny is in a store or not
static function getCurrentBunnyLocation()
{
    $db = Database::get();

    //get the current location
    $statement = $db->prepare("SELECT * FROM locations
        WHERE isBunnyHere=1");
    $statement->execute();
    $currentLocation = $statement->fetch(PDO::FETCH_ASSOC);

}
```

The `getCurrentBunnyLocation()` method will return a dictionary containing all the data the class needs to update the `nextlocation` field.

First you fetch a connection to the database. Then you grab the field from the locations table that indicates the bunny's present location. You store the matched record in \$currentLocation (it's null if there were no active locations found).

If there was a matched location, you want to prepare and return a dictionary of all the relevant data. Add this to the method body:

```
//prepare the method result
$nextLocation = new ArrayObject();

if ($currentLocation) {
    //the bunny is in a store
    $nextLocation['value'] = $currentLocation['store'];
    $nextLocation['changeMessage'] =
        "Next Easter Bunny in %@";

    $nextLocation['location'] = new ArrayObject();
    $nextLocation['location']['latitude'] =
        (double)$currentLocation['lat'];
    $nextLocation['location']['longitude'] =
        (double)$currentLocation['lon'];

    if ($currentLocation['alt']>0) {
        $nextLocation['location']['altitude'] =
            (double)$currentLocation['alt'];
    }

    $nextLocation['location']['relevantText'] =
        "Look in ".$currentLocation['store']."!";
}

return $nextLocation;
}
```

Here you create a new empty dictionary (which you will return as the result later on) called \$nextLocation.

Then, if there is a current location for the bunny, you fill in the data in \$nextLocation:

- **value** – the text value visible for the nextLocation field – it's the name of the store where the bunny is right now.
- **changeMessage** – when the value of a certain field has changed after a pass update, PassKit shows a notification banner. changeMessage contains the text for that banner.
- **location** – this is an array holding all the information needed to update the pass with the next location. As in the previous chapter, you have values for: latitude, longitude, altitude and relevantText.

Finally, you return the `$nextLocation` dictionary as the result of the method.

Now let's return a proper response if the bunny is not at a store. Add more code to the end of the `getCurrentBunnyLocation()` method body:

```
//check whether the game is still active
$statement = $db->prepare("SELECT * FROM egghunt");
$statement->execute();
$gameStatus = $statement->fetch(PDO::FETCH_ASSOC);
```

You fetch the data from the `egghunt` table and store it in `$gameStatus`, so you can check if the game has finished or is still ongoing. Add this final bit of logic, and you're almost done:

```
if ($gameStatus['hasGameFinished']==1) {
    //game finished
    $nextLocation['value'] = "Spend your store credit!";
    $nextLocation['label'] =
        "The Easter Egg Hunt has finished";
    $nextLocation['changeMessage'] = "%@";
} else {
    //just wait for the next bunny location
    $nextLocation['value'] =
        "Wait for the next bunny location";
    $nextLocation['changeMessage'] = "%@";
}
```

If the game has finished, the location field will tell the user that the Egg Hunt is over and that they can use any credit they won to buy goods.

If the game is still ongoing, but the bunny is just in-between locations, then you instruct the player to wait for the next pass notification, which will tell them where to look for the bunny.

One line of code left! Return the result dictionary:

```
return $nextLocation;
```

All right! The method is complete.

Now find the method called `createPassWithUniqueSerialNr()`. In it you generate the initial pass data. However, it's possible that somebody might join in the middle of the game. So let's add a proper value for the `nextLocation` field here.

Find this line:

```
$db = Database::get();
```

Just above it, paste in this code, which will set up the `nextLocation`:

```
//generate the next location field
$nextLocation = self::getCurrentBunnyLocation();

$pass->content['generic'][ 'secondaryFields'][0][ 'value'] =
    $nextLocation[ 'value'];
$pass->
    content[ 'generic'][ 'secondaryFields'][0][ 'changeMessage']=

        $nextLocation[ 'changeMessage'];

if (isset($nextLocation[ 'label'])) {
    $pass->content[ 'generic'][ 'secondaryFields'][0][ 'label'] =
        $nextLocation[ 'label'];
}
```

It's actually pretty easy once you have the handy `getCurrentBunnyLocation()` method! Let's have a quick look at the code:

1. First you load the location data into `$nextLocation`.
2. Then in the next two lines, you copy the values for the `value` and `changeMessage` keys to the pass content.
3. If there's a change to the label of the field, you copy that over as well.

The last element to include in the pass will be the location data. Paste the following code just below the above code:

```
if (isset($nextLocation[ 'location'])) {
    $pass->content[ 'locations'] =
        array($nextLocation[ 'location']);
    $pass->content[ 'relevantDate'] = date("c");
}
```

As a bonus, the pass now has a relevant date – the moment it was generated. You use the `date()` PHP function and as date format you pass “c” – this is the W3C date format, which PassKit expects. So easy!

You also have to implement the same functionality for the method that provides the pass updates. So find `passWithSerialNr()` and find the line `“$pass->writeAllFiles();”` – just before that line, paste in the same code you used before to update the `nextLocation` field:

```
//generate the next location field
$nextLocation = self::getCurrentBunnyLocation();

$pass->content[ 'generic'][ 'secondaryFields'][0][ 'value']=

    $nextLocation[ 'value'];
```

```

$pass->content
['generic'][['secondaryFields'][0]['changeMessage']] =
$nextLocation['changeMessage'];

if (isset($nextLocation['label'])) {
    $pass->content['generic'][['secondaryFields'][0]['label']] =
    $nextLocation['label'];
}

//add relevance fields
if (isset($nextLocation['location'])) {
    $pass->content['locations'][] =
    $nextLocation['location'];
    $pass->content['relevantDate'] = date("c");
}

```

And everything comes together! You've implemented the new smart location field in your passes.

You can now go back to the admin panel implementation and finish the script that sets the new bunny location in the database. You've implemented all the logic – the one thing remaining is to send a push to all passes and let them know something changed, so PassKit can fetch a fresh copy of the pass from your web service.

Updating all passes with the next location

I'm sure you already know how to ping all devices, so let's get cracking! This section will be short and sweet.

Open **admin/setProperty.php** and at the end of the file, paste in this piece of code:

```

//send out push notifications
require_once("../class/APNS.php");
$apns = new APNS();
$updatedNr = $apns->updateAllPasses();

```

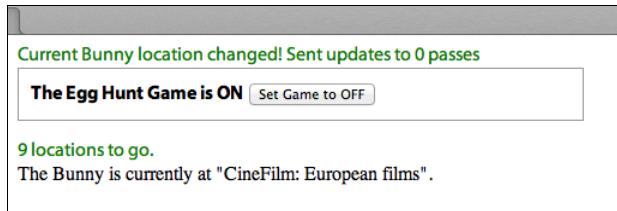
You include the APNS class file, create a new APNS instance, execute `updateAllPasses()`, and store the number of updated devices in the `$updatedNr` variable.

One last task to perform – redirecting the user back to the admin panel page and adding a message to let them know how many devices were sent update notifications:

```
//send the user back to index.php
```

```
header("Location: index.php?message=Current Bunny location  
changed! Sent updates to $updatedNr passes");  
exit();  
}
```

That's all! Open the admin panel page and choose a location from the list. You should now see something like this:



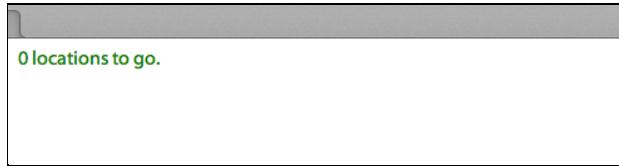
Shortly after you do that, you should receive a push notification update to your last pass, like so:



So far everything is working out just great!

You have reached the point where you are able to scan one of your passes, see the update come through, and then see the new amount of credit pop up on your pass. You can also change the current location, scan the same or another pass, receive the credit update, and so on, and so forth.

Once you use up all game locations, the game will turn itself OFF and you will see the following message on the admin panel:



Of course, when you prepare an admin panel like this one, you most definitely will have to protect it from malicious users. The best way to do this is to add an authentication mechanism with a username and a password.

You've already seen in the previous chapter how to deny access to all clients trying to open a folder by creating an .htdocs file. For your admin folder, you can create an .htaccess requiring a given username and password. Check out this website to learn how:

<http://www.htaccesseditor.com/en.shtml>

They will even create the .htaccess code and the password file for you!

The “log” PassKit web service endpoint

In order to be a good team player, you need to add one final class to your PassKit web service. Check the **ws/v1** subfolder in your web project and you will surely notice one file that remains empty – **Log.php**.

Since PassKit is a brand-new service, Apple expects that sometimes the server code on the developer side might have some glitches, especially right after the release of iOS 6, when things are still new and chaotic.

What kind of problems might there be?

If you have followed the instructions in this chapter exactly, you have already taken care to implement all Apple requirements. So you should not really be concerned about problems on your side.

But, here's a short list of things to always keep an eye on:

- Take care to remove device registrations every time a DELETE registrations request comes to the Device endpoint. Apple won't appreciate it if you send push notifications to devices that have unregistered from your web service, and neither will users.
- Send push notifications ONLY when something changed on the pass that is installed on the target device.
- Make sure you are checking the If-Modified-Since HTTP header. Passes are of considerable file size, especially if they include a background image, so you need to take care not to transfer them over the network if the device already has the same copy of the pass installed.
- Take care to provide a correct Last-modified response HTTP header along with the pass file download.

If you pay close attention to these points, everything in the interactions between Apple and your web service should be A-OK.

However, no matter how perfectly you've implemented the PassKit web service specifications, Apple still requires you to have one more endpoint, which they can use to talk to you. If they notice anything wrong with your PassKit web service implementation, they will leave you a message with a human-readable description of the problem.

The kind of log messages Apple will send you are very useful, so if anything is wrong with your PassKit service, they can be of great help to you. Let's look at an example message:

```
[2012-08-08 14:30:29 +0200] Web service error for
pass.com.yourdomain.freehugcoupon
(http://yourdomain.com/egghunt/ws/): Server ignored the 'if-
modified-since' header (1344403847) and returned the full
unchanged pass data for serial number '134440384769'.
```

As you can see, this message lets you know the exact time a problem occurred, provides the pass type ID and the serial number of the pass, and finally gives a detailed description of what went wrong. Thanks Apple, I'll take it!

Open **ws/v1/Log.php** and paste in the initial class code:

```
<?php

class Log
{

    //takes in the request URL parameters and creates a response
    function __construct($params) {

        $method = strtolower($_SERVER['REQUEST_METHOD']);

        switch ($method) {
            //save a message to the log
            case "post":
                $this->saveLog();
                break;

            default:
                //not valid request parameters
                httpResponseCode(401);exit();
                break;
        }
    }
}
```

```
}
```

```
}
```

The basic class is pretty similar to what you've already implemented for the Passes and Devices classes in the web service.

You declare a new class called `Log` and a constructor that takes in as an argument the URL request parameters from the web service controller. (You won't use that method argument, but you still need to have it, as all your web service classes have one in their constructor.)

You fetch the HTTP request method and store it in the `$method` variable. Then you check the value of the `$method` variable.

If the request method was "post" then you call the class method `saveLog()`, which will take care of saving the message from Apple.

If the method, on the other hand, was not "post", then you return a "401 Unauthorized" HTTP status, as you did many times before in the other classes.

Let's add the `saveLog()` method, which is the only class method you need to implement. Paste inside the class body, just before the final closing bracket:

```
private function saveLog()
{
    //read the POST data and decode the JSON
    $payload = json_decode(file_get_contents('php://input'), true);

    //validate the input
    if ($payload && $payload['logs']) {

        $emailFrom = "noreply@yourdomain.com";
        $emailTo = "you@yourdomain.com";

        mail($emailTo,
            "Apple Pass Service Log",
            "Log message on ".date("Y-m-d H:i:s")."\n".
                print_r($payload['logs'],true),
            "From: ".$emailFrom
        );

    }
}
```

First you read the raw POST request data, since you expect a JSON request from Apple. The JSON object is supposed to have one key called "logs", and its value should be a list of messages Apple wants to send you. These can be error or

warning messages, so take care to keep an eye on Apple's feedback after launching your PassKit service.

You do a little check if `$payload` is not `null` and if it is a dictionary with a "logs" key; unless these conditions are fulfilled, the request is not valid.

Then you define the sender and recipient email addresses – you should change these to the ones that **you want to use**.

Finally, you call the `mail()` function to send an email message to your mailbox containing the Apple problem report.

As you can see, this class is pretty simple – a good note on which to end your PassKit web service implementation. ☺

Warning: As you can see from the Log class implementation, the log endpoint of your web service does not implement any authentication of the client talking to it. At the time of this writing, the Apple specification does not describe a method to verify who is accessing the log endpoint and whether the message is authentic Apple feedback.

Sending the log feedback to your email address is pretty handy, but on the other hand, if somebody wanted to abuse the web service they could jam your inbox pretty fast.

I contacted Apple with a request for more information about the log endpoint. If they provide any additional information, this part of the chapter will be updated accordingly.

The big PassKit round-up

You've made it through two very detailed PassKit chapters, and as a reward you have a deep understanding of most everything related to iOS passes. Congratulations!

I feel very satisfied myself – now not only do you have all the code to help you kick-start your pass development, but you also know everything you really need to build scalable, sustainable, and robust services.

Let's highlight the most important points of what you learned over the course of the past two chapters:

- Designing and building all types of iOS passes.
- Distributing passes via email, direct web download, and bundled with iOS applications.
- Localizing pass contents into different languages.

- Making passes location and date/time relevant.
- Building a two-way connection between PassKit running on iPhone and your own web server.
- Building a web application to control and update your passes.
- And finally, building simple iOS software to scan passes and connect to your server.

That's quite impressive!

In the beginning, I promised you a "wild trip," and I hope I delivered it and that you are satisfied with everything you've learned.



Where to go from here?

The possibilities with PassKit are of course endless! But I can give you some pointers if you are serious about developing pass-powered services and you want to go forward with the project you developed in this chapter:

- The project right now allows you to manage only one pass type ID. You can easily make it handle different pass type IDs, which might be a common case if you are working on a larger-scale system. You can either add another field for the pass type ID in the devices table, or better yet, add another table that will make the connection between the passes table and the devices table. This will add the possibility of filtering passes by pass type ID.
- Have a look at the PassKit iOS documentation – you can use the `PKPassLibrary` to access the installed passes. This gives you the opportunity to create companion iOS apps that can provide complimentary services and information related to your pass services.
- If you are interesting in developing heavy load systems, you can look into things such as: having better uniqueness for the barcodes and the serial numbers; database transactions to ensure database coherency; setting up and testing all

the services through https (which might bring some surprises); and implementing your own login system so that users can register before downloading a pass.

- You can develop the project further and try to implement the software running on the cash desks at the mall as well. You would need to implement the barcode scanning functionality again, connect to the database to verify the credit assigned to the pass, and then subtract the amount of the purchase from the current pass credit. Then, of course, you'd need to send a push notification to the user's devices to update the store credit amount on the front of the pass.
- The script sending push notifications to all the registered devices can take quite some time to do its job. The best implementation of this task is to have a separate table in the database where you save all "push-sending-tasks" and a script running in the background that picks them up and fires them away. To read more about this, consult Matthijs Hollemans' tutorial on Push Notifications: <http://www.raywenderlich.com/3525/apple-push-notification-services-tutorial-part-2>.

Passbook is without a doubt one of the most exciting and promising new features of iOS 6, and now you've got the power of the pass in your hands. Use it well!

Chapter 9: Beginning In-App Purchases

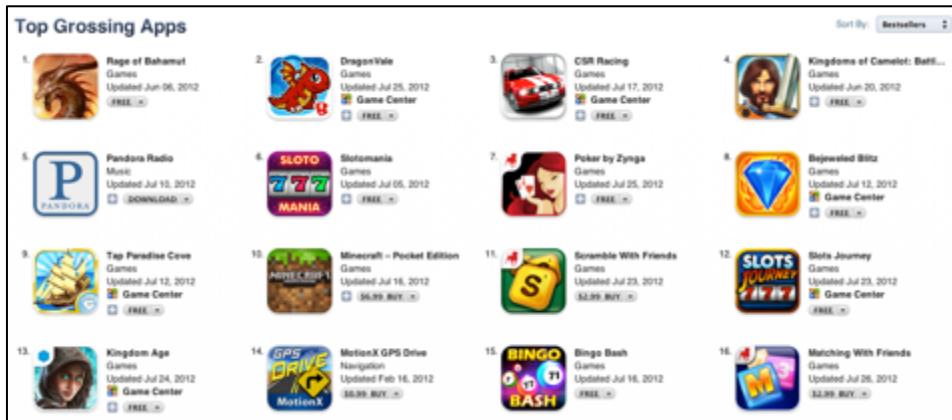
By Ray Wenderlich

In-App Purchases is one of the most exciting features for iOS developers – at least if you like making money. ☺

With In-App Purchases, you can sell content from within your app – whether it be extra levels, in-game items, extra skins, additional features, or anything else you might dream up.

The nice thing about In-App Purchases is you don't just earn money one time when a user first buys your app. Instead, you continue to earn money as users enjoy your app and make In-App Purchases. In other words, the more fun and addicting you make your app, the more you can earn!

To see this for yourself, take a look at the Top Grossing iPhone apps. You'll see around 75% of the 25 top-grossing iPhone apps use In-App Purchases:



So are you ready to take advantage of this amazing capability and start earning some money? These chapters will show you exactly how!

In-App Purchases is a big topic, so the content is split into two chapters. Check the following to see which chapter is best for you to start with:

Beginning In-App Purchases: If you are completely new to In-App Purchases (or need a refresher), you should start here. Note that the material in this chapter is

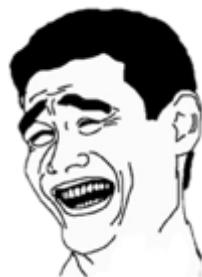
not new to iOS 6, but we thought it was necessary to cover anyway since to understand new iOS 6 features you have to first understand the basics.

- In this chapter, you'll learn how to integrate In-App Purchases into your app in the simplest and easiest way possible.
- You'll learn how to add both consumable and non-consumable In-App Purchases into your app, where the content to unlock is bundled with the application itself.
- Overall, the focus will be on the basics – and getting things done in an easy and straightforward way, but with a solid foundation for expansion.

Intermediate In-App Purchases: If you are already familiar with the basics of In-App Purchases and want to learn about the new iOS 6 features, you should start here.

- You'll learn how to use iOS 6's new Hosted Content feature to store your content on Apple's servers and download it on demand.
- You'll learn how to set up your own server to store your product information and validate receipts.
- You'll learn how to use iOS 6's new `SKStoreProductViewController` to allow users to purchase content from the iTunes Store without having to leave your app.
- Overall, the focus will be on the new iOS 6 features and on developing a flexible server-based solution so you can add new in-app purchases without having to upgrade your app.

"But wait!" you might say. "This stuff seems like a lot of work, and I've got important things to do like catch up on the latest season of the *Walking Dead...*"



"I heard about this cool service/library that makes implementing In-App Purchases a lot easier. Why bother learning this stuff myself?"

Well, learning how to implement In-App Purchases yourself is well worth the time and effort. Here's why:

- Many of these "cool services" actually take a chunk of your hard-earned money to route your In-App Purchases through their system. Don't lose money because you're feeling lazy!
- By relying on a third-party service or library, you now are dependent on them. What happens if you want to move to a different service? Or there is something

you want to do that it doesn't support? Or what if it has a bug, or goes out of business? This could negatively impact your app and business.

- You shouldn't think of adding In-App Purchases as a "side feature" or "last minute thing" to add into your app. In-App Purchases should be considered one of the primary features of your app, and you should spend a lot of time and energy making sure the store is really attractive to your users and your In-App Purchases are fun and well designed. After all, this can make a huge difference in how much money your app makes!
- And most importantly, Apple has made implementing In-App Purchases easy and fun! There's no reason not to learn it – especially because these chapters are here to guide you through the process step-by-step, with a cool and fun real-world example.

So are you ready to become an In-App Purchase boss? Then flip ahead and get ready to make some money!

How do In-App Purchases work?

Before you begin coding, it's important to understand how In-App Purchases work. You'll get a firm overview of the subject in three steps:

1. First, you'll learn about the types of In-App Purchases that are available.
2. Then you'll learn how to register In-App Purchases with Apple.
3. Finally, you'll learn how to make an In-App Purchase in your app.

Reading this section will give you some important background knowledge to make learning easier, but if you are impatient and just want to "learn as you go" by writing code, feel free to skip ahead to the "Introducing Hangman" section.

In-App Purchase types

There are several different types of In-App Purchases – let's discuss what they are and when you'd use them.

Non-Consumable

These are purchases that the user should only need to purchase once. An example would be extra levels – once the user has purchased this, they should never have to purchase the same level again.

As an example, my app Wild Fables allows users to purchase additional fables as non-consumable In-App Purchases:



Once the user purchases a Wild Fable, they have it forever – they never have to purchase it again.

Note that the expectation of users is that when they make a non-consumable purchase, they should be able to access that particular item on any of their devices. Apple provides an easy way for your app to restore non-consumable purchases made on other devices, which you'll learn about later in this chapter.

Consumable

These are purchases that users can make multiple times. An example would be an in-game item such as "10 gems" that you want the user to be able to purchase over and over again if they choose.

As an example, in the game Jetpack Joyride, you can buy various sets of Coin Packs that you can then redeem for in-game items:



There is no limit to how many coin packs you can buy – so users can buy the same pack of coins over and over again.

When it comes to making money, consumables are where it's at. It makes sense – users can purchase non-consumables only once, but with consumables there is no limit!

This allows your app to take advantage of the "long tail" effect. Your app might have some users who really enjoy your app and are willing to spend a lot of money on it. Consumables let these users spend as much as they want.

Note: For more information on the "long tail" effect, check out this great set of slides from Noel Llopis: <http://gamesfromwithin.com/the-power-of-in-app-purchases>

Also, note that for your consumable purchases to be as effective as possible, you need to design your app from the ground up with consumable purchases in mind!

For a good collection of articles about freemium app design (that applies to any app with consumable purchases), check out this collection by Alex Curylo: <http://www.alexcurylo.com/blog/2011/05/07/freemium-design/>

Auto-Renewable Subscriptions

Auto-renewable subscriptions allow you to charge users a set amount on a periodic basis in order to continue to receive new content delivered by your app.

Sounds cool, right? Well, the drawback is that at the time of writing this chapter, Apple seems to have made a policy decision that auto-renewable subscriptions can only be used for magazine or newspaper-style apps.

This means you cannot use this feature for other types of apps (such as having a subscription to unlock a service or feature in your game), as tempting as auto-renewable subscriptions may be for your business case.

Instead, Apple encourages you to use non-renewing subscriptions, which unfortunately requires more of an effort on the user's end to choose to renew each month (and hence likely less sales for you). For more information, check out this blog post: <http://blog.migrantstudios.com/2011/09/12/apple-says-app-store-subscriptions-are-media-only/>

Free Subscriptions

These are similar to auto-renewable subscriptions, but are free and only available for Newsstand apps.

Non-Renewing Subscriptions

If you can't use auto-renewable subscriptions (because your app isn't a magazine or newspaper-style app) but still want to provide time-limited access to content,

you might want to use non-renewing subscriptions. An example would be if you wanted to allow your users to access a particular feature for one week.

Note: In this book, you're going to focus on consumable and non-consumable purchases only, as they are the most common types. Plus, subscriptions are an entirely different beast.

Registering a purchase

Before you can provide an In-App Purchase in your app, you need to let Apple know about it by registering the purchase in iTunes Connect.

The process is fairly simple, and you'll try it out for yourself later. But for now, just know that you will log onto iTunes Connect and fill out a form with some information:

The screenshot shows the 'In-App Purchase Summary' section of the iTunes Connect interface. It includes fields for Reference Name (set to 'ioswords'), Product ID (set to 'com.cazeware.hangman.ioswords'), and Pricing and Availability details like Cleared for Sale (Yes) and Price Tier (Tier 1). A note at the top states: 'Enter a reference name and a product ID for this In-App Purchase. You must also add at least one language, along with a display name and a description in that language.'

This shows you filling out the Product ID for your in-app purchase, which you can think of as a unique string identifying your purchase, as well as its price.

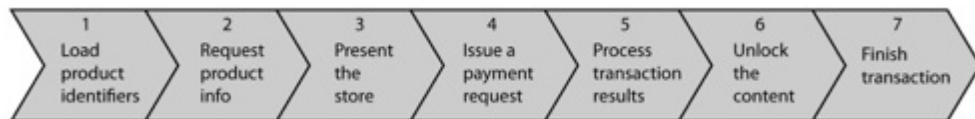
The screenshot shows the 'Add Language' screen in iTunes Connect. It includes fields for Language (English), Display Name (iOS Words), and Description (containing the text: 'Geeks rejoice - expand your game with words related to iOS development! This time, the penalty for entering the wrong keyword isn't an error - it's death! :)'). The 'Save' button is visible at the bottom right.

And this shows you filling out some display information about your purchase, such as its (localized) name and description.

Making a purchase

OK, so you've registered your In-App Purchases in iTunes Connect and want the user to create a purchase. How does that work?

Well, there are just seven simple steps:



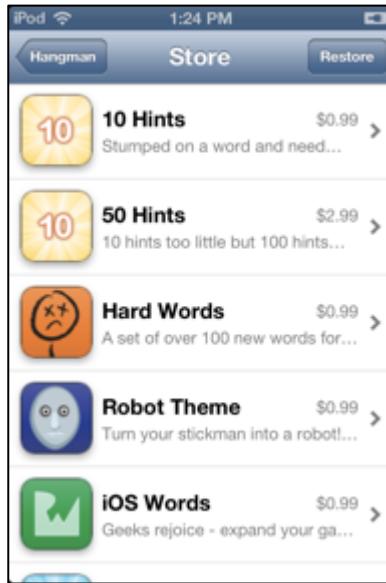
- 1. Load product identifiers.** Before you can begin, your app needs to know the product identifiers of your In-App Purchases that you set up in iTunes Connect. These could be hard-coded into your app (as you'll learn in this chapter), or retrieved from a server (as you'll learn in the next chapter).

```
NSSet * productIdentifiers = [NSSet setWithObjects:  
    @"com.razeware.hangman.tenhints",  
    @"com.razeware.hangman.hundredhints", nil];
```

- 2. Request product info.** Next, your app should connect to the App Store to get the product information that you set up in iTunes Connect. You'll get back all the information that you set up there in a delegate callback in a special object called an `SKProduct`.

```
_productsRequest = [[SKProductsRequest alloc]  
    initWithProductIdentifiers:productIdentifiers];  
_productsRequest.delegate = self;  
[_productsRequest start];
```

- 3. Present the store.** Next you need to present an in-app store listing the available purchases. Note that Apple does not provide a built-in view controller to do this for you – you have to write it yourself. This is because how you want the store to look depends heavily on what your app is like – i.e., a store for a game might look very different than a store for an app. You're encouraged to spend time making the store as appealing as possible to users, because after all, this is your chance to convince them to spend their hard-earned money! Here's what the store you'll create in this chapter will look like:



- 4. Issue a payment request.** Once the user selects an item to buy, you issue a payment request to the In-App Purchase API. It will automatically present any popups, like "Are you sure you want to purchase this?"

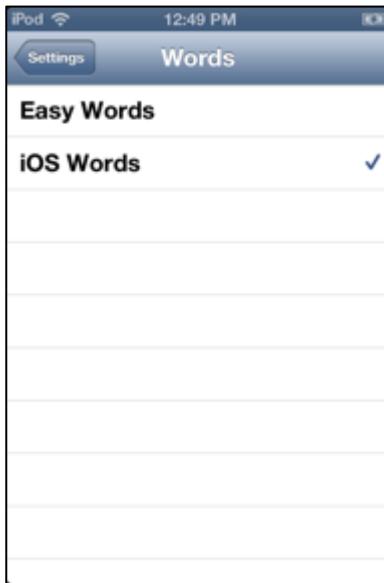
```
SKPayment * payment = [SKPayment
    paymentWithProduct:skProduct];
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

- 5. Process transaction results.** The In-App Purchase API will attempt to charge the user and will notify you when the payment succeeds or fails. At your app's startup, you will have registered to receive notifications of transaction results like this. At this point, you can optionally verify that the purchase is valid by contacting Apple's servers (more on this later), and download any content if it's not already embedded in your app.

```
- (void)paymentQueue:(SKPaymentQueue *)queue
updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction * transaction in transactions) {
        switch (transaction.transactionState) {
            {
                case SKPaymentTransactionStatePurchased:
                    [self completeTransaction:transaction];
                    break;
                case SKPaymentTransactionStateFailed:
                    [self failedTransaction:transaction];
                    break;
                case SKPaymentTransactionStateRestored:
                    [self restoreTransaction:transaction];
                    default:
```

```
        break;  
    }  
}  
}
```

6. **Unlock the content.** This is the important part – here you unlock the content and make it available to the user!



7. **Finish transaction.** As a final step, you need to tell the In-App Purchase API that you're done – otherwise it will assume it didn't finish and will try to complete the transaction over and over.

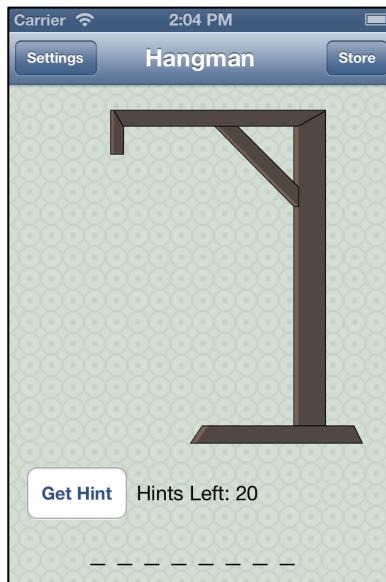
```
[[SKPaymentQueue defaultQueue] finishTransaction: transaction];
```

But enough talk – it's time to try this out for yourself! In this chapter, you're going to implement In-App Purchases from scratch in a simple iPhone game – Hangman!

Introducing Hangman

For the rest of this chapter, you will be working with a starter project for a simple iPhone game – Hangman!

In the resources for this chapter, look for a project called **HangmanCh9Starter**. Open it in Xcode, run the project and you will see the following:



This is the game of Hangman you may have played as a child. Your goal is to guess the word at the bottom of the screen. Tap the label to bring up a keyboard and try to guess the letters of the word:



If you are correct, the letter will appear in the word, but if you are wrong, part of your hangman will appear on the noose. If you are wrong enough times, you are dead! It's a rather a morose game, if you think about it. ☺

See if you can beat the game – and if you get stuck, you can always tap the Get Hint button to have a handy free letter appear in your word. But be careful – you only have 20 hints!

Poke around through the app. You'll notice a settings screen that lets you choose the types of words or the theme, but there's only one choice for each at the moment.

You'll also notice there's an option for a "Store", but it's completely empty. If you tap the Restore button, it will bring up a view controller that is supposed to be the "Store Details" page (it's temporarily attached to this button for testing purposes so you can see all the screens).

This is your mission in this chapter, should you choose to accept it:

Add In-App Purchases to Hangman. Specifically, allow the user to buy extra hints, and unlock extra packs of words. This message will self-destruct in 10 seconds. ☺

If you complete your mission, your users will be happy because they'll be able to purchase additional fun sets of words, and more hints to avoid the noose.

And of course you know the real reason you're doing this – so you can be like this guy!

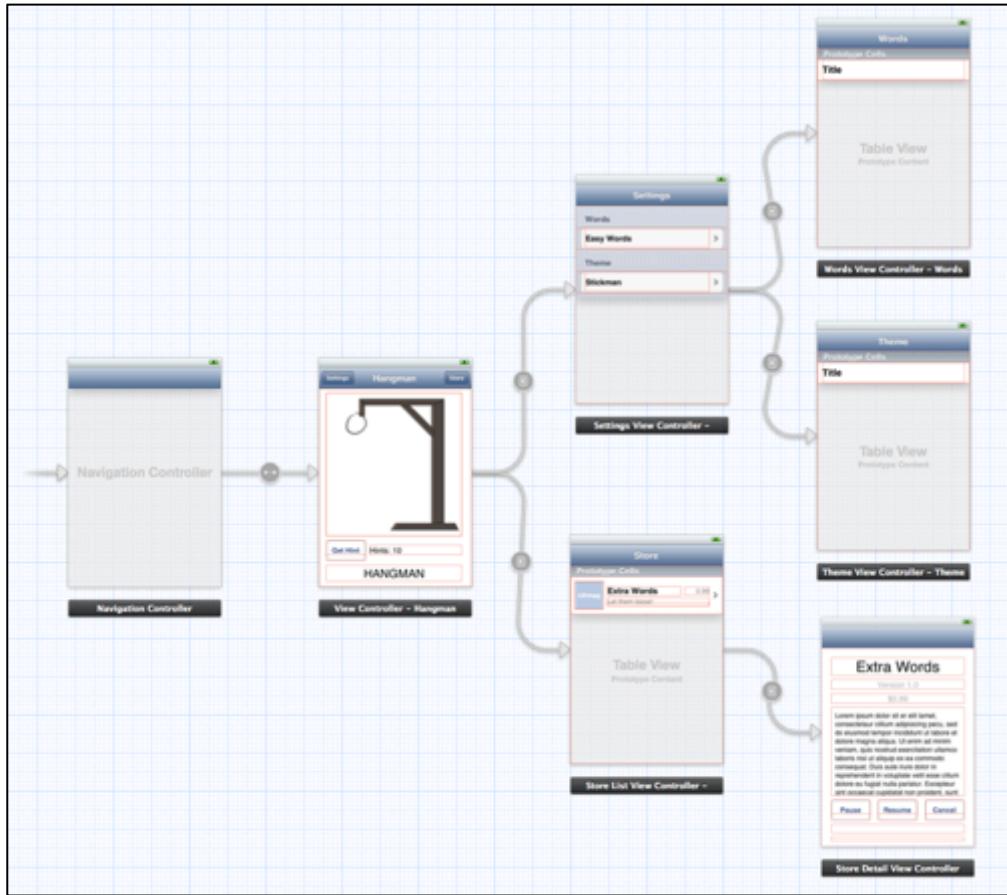


A quick tour of the code

Next let's take a quick tour through the current code so you have a good understanding of how it works.

Reading this section and the next will give you a good understanding of how the code you're about to write fits in with the rest of the project. But if you are impatient and just want to "learn as you go" by writing code, feel free to skip ahead to the "Getting Started" section.

OK, all aboard the tour bus! A good place to start is with **MainStoryboard.storyboard**. Open it and you'll see the layout for the app:



As you can see, the main view controller for the game (the one with the hangman picture) is `HMViewController`. Note that the Storyboard is set up to use Auto Layout, so everything works fine on the new iPhone 5 screen size!

Open up `HMViewController.m` to get a feel for what it does:

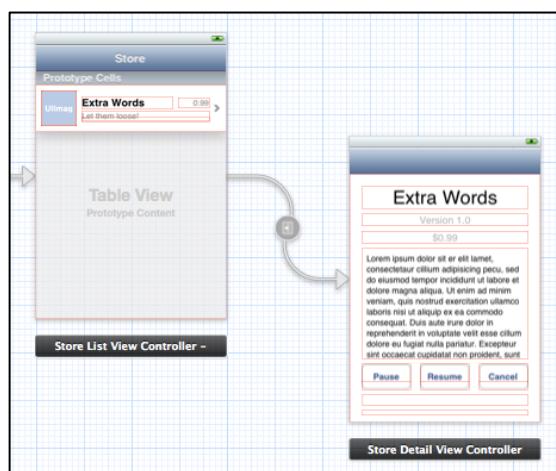
- In `viewDidLoad`, it performs some initial configuration steps, then creates a new `HMGame` and calls `setup`.
- `setup` gets ready to play some sound effects (based on the current theme), then calls `newGame`.
- `newGame`: gets the current theme and set of words from the `HMContentController` and then calls `newGameWithWords:maxWrongGuesses:` on the `HMGame` created earlier. Finally, it calls `refresh`.
- `refresh` simply updates the user interface (the hangman image and label) based on the `HMGame` state.
- Finally, when you tap a keystroke, `hiddenTextFieldValueChanged:` is called. This calls `guess:`, which in turn calls `guess:` on the `HMGame`. It also checks to see if the game is won or not, and refreshes the screen appropriately.

There are some other bits in there, but those are the important parts. The main takeaway is the following: this view controller doesn't do much except forward the work to the real workers – the model classes like `HMGGame`. Kinda like an annoying boss! ☺

So let's turn to where the work is actually performed. Open the **Model** group and browse through the classes:

- `HMGGame`: This class contains the game logic. When you create a new game, it chooses a random word from the `HMWords` passed in, and stores it. It also keeps track of what letters the user has guessed so far, and calculates the appropriate string to display to the user, like "TUT_RIAL".
- `HMTHEME`: A theme for Hangman consists of a bunch of images for the character to "hang" (in this case, a Stickman), along with some sound effects to play when the user enters a correct or incorrect answer, wins, or loses. Notice that instead of being hard-coded, all of this information is retrieved from a property list – look at **Resources\Stickman\theme.plist** for an example. I'll explain more on why this is done this way later.
- `HMWORDS`: Contains a list of words that can be used in a Hangman game. Like `HMTHEME`, this list of words is not hard-coded, it is drawn from a property list – look at **Resources\EasyWords\words.plist** for an example.
- `HMContentController`: You can think of `HMContentController` as the object that knows what app content is currently available/unlocked. Right now, it's hard-coded to unlock some "built-in" content in `init`. It also keeps track of "consumable" items like hints, which in this case are simply stored in `NSUserDefaults`.

Next, take a quick look `HMStoreListViewController` and `HMStoreDetailViewController`. The user interface and table view cell for these have already been made for you:



But the code for these has not yet been added – that's part of what you'll be doing in this chapter. Open **HMStoreListViewController.m** and

HMStoreDetailViewController.m and notice how they are pretty empty at this point.

Finally, notice that some handy libraries have already been added to your project, that you'll be using in the rest of this chapter and next. Feel free to take a look around – there's some useful stuff in there.

And that's it – you've looked through the most important classes in this project and should have a good understanding of how things work. This concludes our tour – thanks for coming along!

Design considerations

There's one final thing to discuss before you start writing code.

There are two things this project does to make implementing In-App Purchases easier:

1. Themes and words have been designed to be file-based.
2. Each theme or word list is stored within its own private directory.

In this section, we will discuss both of these and why they make implementing In-App Purchases easier for this game. You might find it useful to do something similar in your own apps.

Note that if you don't understand why this makes anything easier after reading this section, it's no big deal. It will become clearer as you continue through this chapter and the next.

Let's take a closer look at these things the project did to make implementing In-App purchases easier:

1. Themes and words have been designed to be file-based.

A naive approach to implementing themes would have been to create a `StickmanTheme` class that returns the appropriate images and sound effects – perhaps as a subclass to `HMTHEME` or `HMWORD`.

However, if you ever want to make a new theme for purchase, this requires you to write new code! This is not good for a number of reasons:

- The more code you write, the more chances to introduce bugs.
- You cannot have downloadable code as an In-App Purchase. Therefore, you'd have to release an update to your app whenever you want to add new content.

So instead, the `HMTHEME` and `HMWORDS` classes have been designed to be file-based – they read the information they need from a property list. This way, adding a new `HMTHEME` and `HMWORDS` is as easy as adding a new file – which will pay large dividends later!

2. Each theme or word list is stored within its own private directory.

It makes things a lot easier for you if you know each theme or word list is stored in its own directory.

That way, when you want to read a theme for example, all you need to know is the URL of the directory where it's stored. You can then look for a file called theme.plist in that directory, and read that to find the names of anything else you might need.

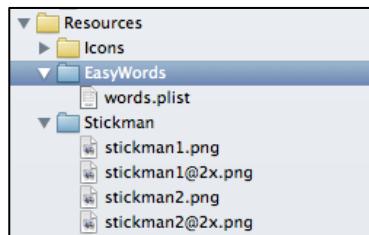
To see for yourself, take a look at the `initWithDirURL:` methods in `HMTHEME` and `HMWORDS` – this is exactly what they do.

And as you'll see later in this chapter and the next, this makes adding new themes and words very easy – you just need to store a directory of files somewhere and call this initializer!

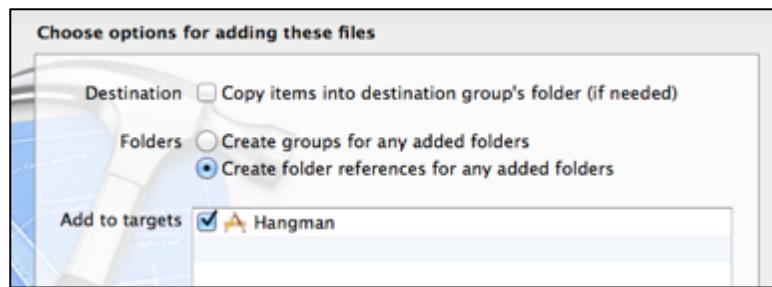
But what about the practical aspect of this – how can you make sure that a group of files that you add into your app is stored in its own directory inside your application bundle?

You might not know this, but when you normally add files into your Xcode project and compile your app, all of the files are stored in the root application bundle directory – not in any subdirectories.

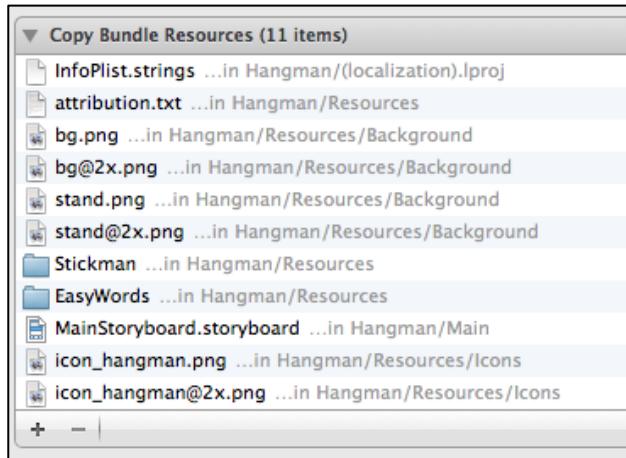
However, you can change this behavior. If you look at the Project Navigator, you will see that the EasyWords and Stickman folders are blue:



This means that they are folder references, not group references. You can mark something as a folder reference when you add it into Xcode in this dialog, which I'm sure you've seen many times before:



What this means is that when the files are copied into your app's bundle, they are copied with the directory structure intact. See this for yourself by selecting your Hangman project in the Project Navigator, selecting the Hangman target, selecting the Build Phases tab, and expanding the Copy Bundle Resources section:



As you can see, most files like bg.png or icon_hangman.png are copied into the root of the Application bundle. However, the Stickman and EasyWords directories are copied into their own directories!

Getting started

Now that you're armed with a firm overview of In-App Purchases and how the Hangman starter project is organized, you are finally ready to start adding some In-App Purchases to Hangman!

You are eventually going to add multiple types of In-App Purchases into Hangman: extra themes, extra words, and extra hints. But to keep things simple, you're going to add them one at a time, and the first one you'll add is extra hints.

Before you can add In-App Purchases to your app, you have a couple administrative details to take care of. You've probably done this already, but just in case:

- Make sure you are a member of the iOS Developer Program.
- Make sure you have agreed to the latest iOS Developer Program License Agreement in iTunes Connect.
- Make sure you have completed your iOS Paid Applications contract in iTunes Connect.

As mentioned in the "Registering a Purchase" section, the first step to add an In-App Purchase is to register it in iTunes Connect. But of course to register an app in iTunes Connect you first need an App ID – so let's take care of that first.

Log onto the iOS Provisioning Portal (via a link on the [iOS Developer Center](#)) and click the tab for **App IDs**. You will see something like this:

The screenshot shows the Razeware LLC Provisioning Portal. The left sidebar has links for Home, Certificates, Devices, App IDs (which is selected and highlighted in blue), Pass Type IDs, Provisioning, and Distribution. The main content area has tabs for Manage and How To, with 'Manage' selected. Under Manage, there's a sub-section for 'App IDs'. A large button labeled 'New App ID' is visible. Below it, a text box explains what an App ID is: 'An App ID is the combination of a unique ten character string called the "Bundle Seed ID" and a traditional CF Bundle ID (or Bundle'. There's also a note about Push Notifications and iCloud sharing.

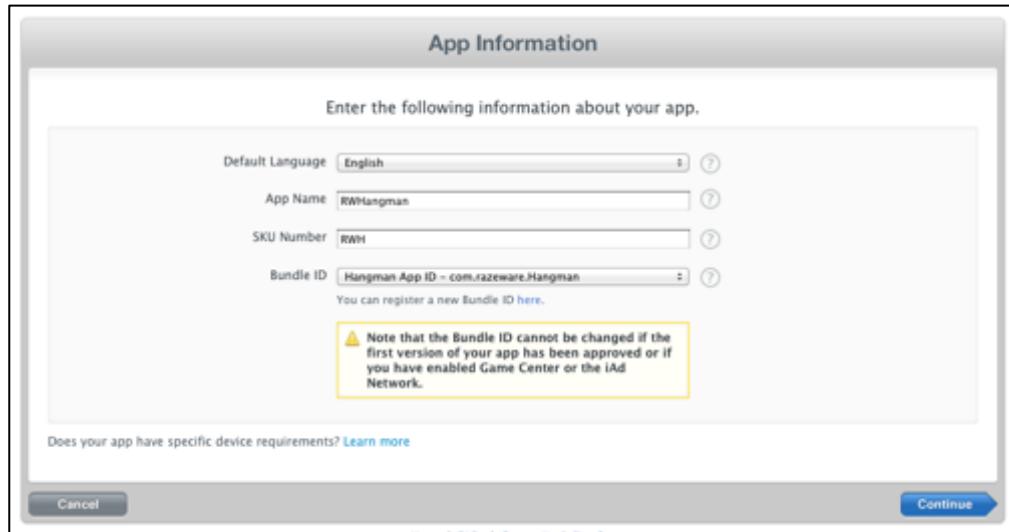
Click the button in the upper right that says **New App ID**. Then fill out the form that appears to create a new App ID for Hangman, similar to the following:

The screenshot shows the 'Create App ID' form in the Razeware LLC Provisioning Portal. The left sidebar is identical to the previous screenshot. The main form has three sections: 'Description' (with a note about alphanumeric characters and a field containing 'Hangman App ID'), 'Bundle Seed ID (App ID Prefix)' (with a note about Team ID and a dropdown menu set to 'Use Team ID'), and 'Bundle Identifier (App ID Suffix)' (with a note about reverse-domain name style and a field containing 'com.razeware.Hangman'). At the bottom are 'Cancel' and 'Submit' buttons.

Of course, you should replace the Bundle Identifier with your own unique ID, ideally based on a domain name you control, for example: com.yourdomain.Hangman. If you do not own any domains, you could use your name, like com.raywenderlich.Hangman.

Note that the App ID cannot include a wildcard character – only explicit App IDs are supported with In-App purchases.

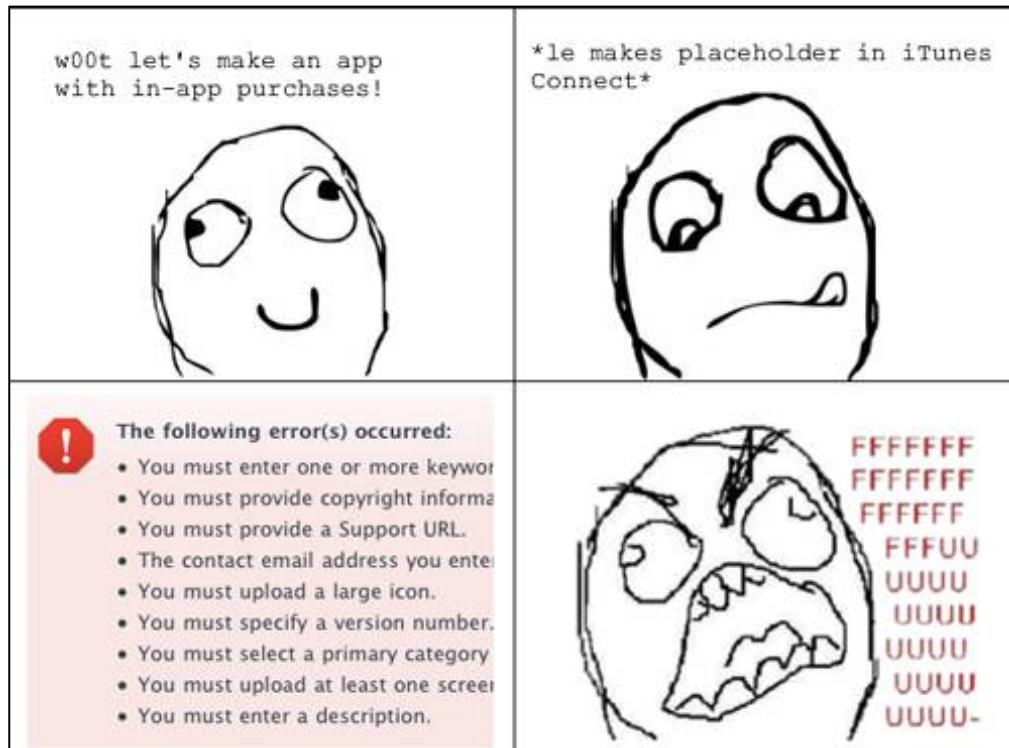
Next, log into [iTunes Connect](#), click **Manage Your Applications**, and click **Add New App**. If a screen appears that asks you to select iOS App or Mac OS X App, select **iOS App**, of course. Then create a new app with the Bundle ID you just created, like the following:



Note that you will have to choose something different for the App Name, as I've already claimed this one and app names must be unique.

Click **Continue**, and the next two pages will ask for your app's information. Just put in placeholder information for now – you can change it all later. But unfortunately, at the time of writing this chapter you have to fill out almost every field in order to make it validate correctly – including an icon and screenshots!

Just so you know, here's how I feel about this:



If you get any errors like the above, just keep entering in dummy data. Also, the resources for this chapter contains an **iTunesConnectResources** folder with a 1024x1024 icon and screenshots that you can upload to make iTunes Connect happy.

After you get through all of the errors, click on your new app and it should show you the App Information page:

The screenshot shows the 'App Information' page for an app named 'RW Hangman'. The 'Identifiers' section includes fields for SKU (RWH), Bundle ID (com.razeware.Hangman), Apple ID (544291393), Type (iOS App), and Default Language (English). The 'Links' section contains a 'View in App Store' link. To the right, there is a vertical column of blue buttons: 'Rights and Pricing', 'Manage In-App Purchases', 'Manage Game Center', 'Set Up iAd Network', 'Newsstand', and 'Delete App'.

Click the **Manage In-App Purchases** button and then **Create New**, and you will see the following dialog:

The dialog is titled 'Select Type' and asks the user to choose an In-App Purchase type. It includes a note about agreeing to contracts and legal agreements. Below this, there are two sections: 'Consumable' and 'Non-Consumable'. Under 'Consumable', there is a description and a 'Select' button. Under 'Non-Consumable', there is a description and a 'Select' button.

This allows you to choose the type of In-App Purchase as discussed in the "In-App Purchase Types" section. You want the users to be able to purchase hints as often as they like, so you want to make it a consumable In-App Purchase. Click **Select** under the **Consumable** section.

You are going to provide two denominations of hints the user can purchase: a set of 10 hints, or a set of 100 hints. Let's start with 10 hints, so fill out the form as follows:

In-App Purchase Summary

Enter a reference name and a product ID for this In-App Purchase. You must also add at least one language, along with a display name and a description in that language.

Reference Name: [?](#)

Product ID: [?](#)

Pricing and Availability

Enter the pricing and availability details for this In-App Purchase below.

Cleared for Sale Yes No

Price Tier [?](#)
[View Pricing Matrix](#)

Note that for Product ID, you should replace this with your own reverse DNS notation as earlier. So, for example, it might be com.yourdomain.hangman.tenhints for you.

Scroll down to the **In-App Purchase Details\Language** section and click **Add Language**. Fill out the popup as follows:

Add Language

Language [?](#)

Display Name [?](#)

Description [?](#)

[Cancel](#) [Save](#)

Click **Save**, and then scroll down to the bottom of the page and click **Save** again. w00t – your first In-App Purchase is registered!

Now repeat to create a second consumable In-App Purchase for 100 Hints. Click **Create New**, and then **Consumable\Select**. Enter the following information up top:

In-App Purchase Summary

Enter a reference name and a product ID for this In-App Purchase. You must also add at least one language, along with a display name and a description in that language.

Reference Name: [?](#)

Product ID: [?](#)

Pricing and Availability

Enter the pricing and availability details for this In-App Purchase below.

Cleared for Sale Yes No

Price Tier [?](#)
[View Pricing Matrix](#)

Again, don't forget to replace the Product ID with one based on your own Bundle Identifier.

Then scroll down to the **In-App Purchase Details\Language** section and click **Add Language**. Fill out the popup as follows:

Add Language

Language [?](#)

Display Name [?](#)

Description [?](#)

[Cancel](#) [Save](#)

Click **Save**, and then scroll down to the bottom of the page and click **Save** again.

You're done with iTunes Connect for now. To finish this process, you just need to make sure your project is set up with the correct Bundle Identifier for your new App ID and add a framework to your project.

Switch back to your Hangman project in Xcode and open **Supporting Files\Hangman-Info.plist**. Find the Bundle Identifier line and set it to the correct value from your App ID, similar to the following:

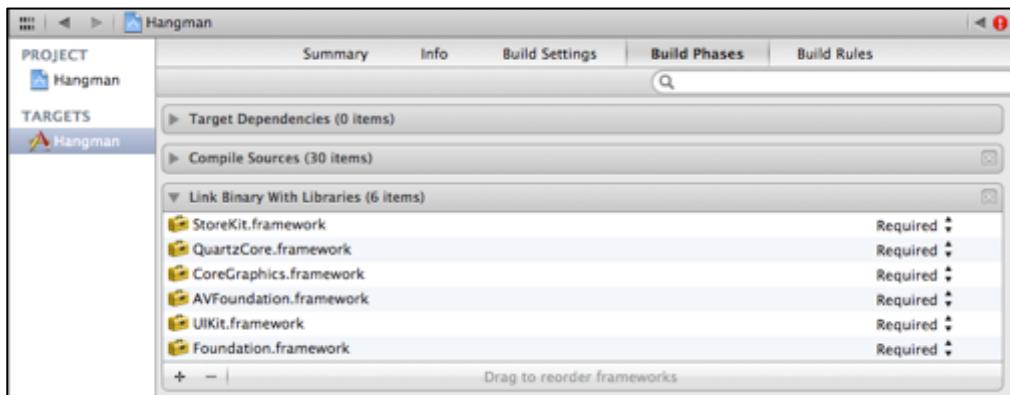
| Key | Type | Value |
|-------------------------------------|--------|----------------------|
| Localization native development reg | String | en |
| Bundle display name | String | \$(PRODUCT_NAME) |
| Executable file | String | \$(EXECUTABLE_NAME) |
| Bundle identifier | String | com.razeware.hangman |

After changing your bundle identifier, do the following just to make sure Xcode isn't using the old bundle identifier anywhere:

- Go to Product\Clean in Xcode
- Delete the app off your simulator or device
- Shut down and restart Xcode and the simulator or device
- Run again and make sure all is OK.

One final step. To call Apple's APIs that deal with In-App Purchases, you need to add a framework to your project – the StoreKit framework. To do this, click on your **Hangman** project root in the Project Navigator and then the **Hangman** target.

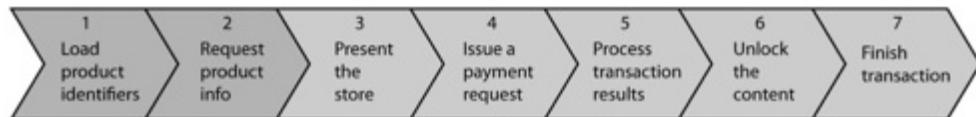
Select the **Build Phases** tab, scroll down to the **Link Binary with Libraries** section, and click the **(+)** button. Select **StoreKit.framework**, and click **Add**. At this point, your list of libraries should look like the following:



That's it! You have now successfully registered two consumable In-App Purchases in iTunes Connect and have set up your project to use StoreKit. Time to write some code!

Retrieving products

If you recall, back in the "Making a Purchase" section, the first two steps are to load the product identifiers and request the product info:

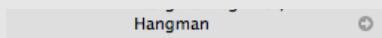


So you'll start with exactly that! You'll put all of this code in a helper class so you can keep all of the In-App Purchase code nicely centralized.

Control-click on the Hangman group at the root of the Project Navigator, select **New Group**, and name the new group **In-App Purchases**.

Tip: If you want to keep your project directory organized, you can set things up so that any file you create in this group gets stored in a similarly named folder inside your project directory. If you don't care about this, skip ahead.

To do this, click on the new **In-App Purchases** group, open the **Utility panel**, and select the **File inspector**. Under Path, click the button next to None:



The popup should take you to your Hangman project directory – navigate there if it doesn't. Then click **New Folder**, name it **In-App Purchases**, click **Create**, and then click **Choose**.

Now, any file you create in this group will be saved to this folder, preventing your Xcode project directory from getting cluttered up. Cue massive sigh of relief! ☺

Control-click on the **In-App Purchases** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **IAPHelper**, make it a subclass of **NSObject**, and click **Next** and finally **Create**.

Open **IAPHelper.h** and replace it with the following:

```

@interface IAPHelper : NSObject
- (void)requestProductsWithProductIdentifiers:
    (NSSet *)productIdentifiers;
@end
  
```

As you can see, you are going to write a single method that will retrieve information about a set of product identifiers from the App Store.

Then switch to **IAPHelper.m** and add the first part of the implementation:

```

// 1
#import "IAPHelper.h"
  
```

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

// 2
@interface IAPHelper () <SKProductsRequestDelegate>
@end

@implementation IAPHelper {
    SKProductsRequest * _productsRequest; // 3
}

- (void)requestProductsWithProductIdentifiers:
    (NSSet *)productIdentifiers {

    // 4
    _productsRequest = [[SKProductsRequest alloc]
        initWithProductIdentifiers:productIdentifiers];
    _productsRequest.delegate = self;
    [_productsRequest start];
}

@end
```

Let's go over what you did here section by section.

1. You need to use StoreKit to access the In-App Purchase APIs, so you import the StoreKit header here.
2. To receive a list of products from StoreKit, you need to implement the `SKProductsRequestDelegate` protocol. Here you mark the class as implementing this protocol in the class extension.
3. You create an instance variable to store the `SKProductsRequest` you will issue to retrieve a list of products, while it is active. You keep a reference to the request so a) you can know if you have one active already, and b) you'll be guaranteed that it's in memory while it's active.
4. Given a set of product identifiers (such as `com.razeware.hangman.tenhints`), ask the App Store to return information about those products by making a `SKProductsRequest`. Note that it sets the current class as the delegate of the `SKProductsRequest` so it will receive callbacks when the request succeeds or fails.

Speaking of delegate callbacks, add those next! Add the following code before the `@end`:

```
#pragma mark - SKProductsRequestDelegate

- (void)productsRequest:(SKProductsRequest *)request
didReceiveResponse:(SKProductsResponse *)response {
```

```
NSLog(@"Loaded list of products...");
_productsRequest = nil;

NSArray * skProducts = response.products;
for (SKProduct * skProduct in skProducts) {
    NSLog(@"Found product: %@ %@ %0.2f",
          skProduct.productIdentifier,
          skProduct.localizedTitle,
          skProduct.price.floatValue);
}

- (void)request:(SKRequest *)request
didFailWithError:(NSError *)error {

    NSLog(@"Failed to load list of products.");
    _productsRequest = nil;
}
```

Here you implement the two delegate callbacks – for success and failure. On success, you log out some information about the returned products, such as the product identifier, localized title, and price. Either way, you set the `_productsRequest` instance variable back to nil because you're done with it.

Next, you're going to make a subclass for `IAPHelper` called `HMIAPHelper`. This is because `IAPHelper` is meant to be completely project-independent, so you can easily reuse it in your own projects. When you need project-dependent code (like the list of In-App Purchase product identifiers for your project), you'll add that to the project-specific subclass.

So control-click on the **In-App Purchases** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **HMIAPHelper**, make it a subclass of **IAPHelper**, and click **Next** and finally **Create**.

Open **HMIAPHelper.h** and replace it with the following:

```
#import "IAPHelper.h"

@interface HMIAPHelper : IAPHelper

+ (HMIAPHelper *)sharedInstance;
- (void)requestProducts;
```

```
@end
```

This defines two methods – a static method to return the single, global instance of this class, and a method to request the list of products. Note that this time, it takes no parameters – the implementation will hard-code the list of In-App Purchase product identifiers and call the superclass.

Next switch to **HMIAPHelper.m** and replace it with the following:

```
#import "HMIAPHelper.h"

@implementation HMIAPHelper

+ (HMIAPHelper *)sharedInstance {
    static dispatch_once_t once;
    static HMIAPHelper * sharedInstance;
    dispatch_once(&once, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}

- (void)requestProducts {
    NSSet * productIdentifiers = [NSSet setWithObjects:
        @"com.razeware.hangman.tenhints",
        @"com.razeware.hangman.hundredhints", nil];
    return [super requestProductsWithProductIdentifiers:
        productIdentifiers];
}

@end
```

This is as discussed above – the `sharedInstance` method implements the Singleton pattern in Objective-C to return a single, global instance of the `HMIAPHelper` class. `requestProducts`, on the other hand, simply hard-codes the product identifiers you created earlier in iTunes Connect in an `NSSet`, and passes them to the superclass.

Don't forget to replace these product identifiers with the actual identifiers you set up in iTunes Connect!

Now let's try this out. Switch to **HMStoreListViewController.m** and add this import to the top of the file:

```
#import "HMIAPHelper.h"
```

Then call `requestProducts` at the end of `viewDidLoad`:

```
[[HMIAPHelper sharedInstance] requestProducts];
```

That's it! Build and run and tap on the Store button. Look at your debug Console, and if all goes well you should see the information about the two In-App Purchases you set up in iTunes Connect!

```
2012-07-28 17:29:30.485 Hangman[17719:c07] New game. Hidden word: NATIONAL
2012-07-28 17:29:32.443 Hangman[17719:c07] Loaded list of products...
2012-07-28 17:29:32.444 Hangman[17719:c07] Found product: com.razeware.zombies.hundredhints 100 Hints 4.99
2012-07-28 17:29:32.445 Hangman[17719:c07] Found product: com.razeware.zombies.tenhints 10 Hints 0.99
```

Note: If you don't see anything, double-check that the Bundle Identifier in your Hangman-Info.plist matches the App ID you set up, and that it matches the App you created in iTunes Connect. Also double-check that the product identifiers hard-coded into `HMIAPHelper` match what you set up in iTunes Connect.

Keep in mind that it can be some time after you set up an In-App Purchase in iTunes Connect before it will be returned in this API call. I generally see it start working within about 1-30 minutes after setting it up in iTunes Connect.

Finally, sometimes you might get an error that says, "Cannot connect to iTunes Store." If you see this, it could mean you have network problems, or the iTunes sandbox is down. Check this URL - if it doesn't respond, the iTunes sandbox may be down:

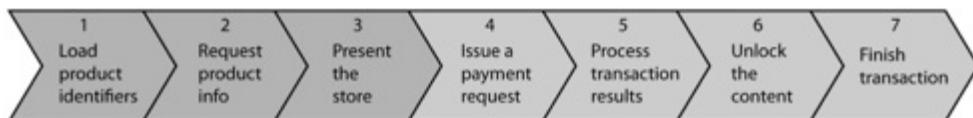
<https://sandbox.itunes.apple.com/verifyReceipt>

For more troubleshooting hints, check out the forum thread for our Introduction to In-App Purchases tutorial:

<http://www.raywenderlich.com/forums//viewtopic.php?f=2&t=188>

Displaying products

Awesome – you have a list of products, so now let's take this one step further and make them show up in the store list view! This is the third step in the process:



To implement the store, you could just send the list of `SKProducts` returned to you by the App Store to the list view and display the information to the user. But over

time, you're going to need a lot more information about the products other than just the `SKProduct` class.

So to keep things clean, you are going to create a class to contain all the information you need about a product called `IAPPProduct`. The `IAPPProduct` class will contain the `SKProduct`, among other things.

Control-click on the **In-App Purchases** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **IAPPProduct**, make it a subclass of **NSObject**, and click **Next** and finally **Create**.

Open **IAPPProduct.h** and replace it with the following:

```
@class SKProduct;

@interface IAPPProduct : NSObject

- (id)initWithProductIdentifier:(NSString *)productIdentifier;
- (BOOL)allowedToPurchase;

@property (nonatomic, assign) BOOL availableForPurchase;
@property (nonatomic, strong) NSString * productIdentifier;
@property (nonatomic, strong) SKProduct * skProduct;

@end
```

At this stage, the `IAPPProduct` just keeps track of the product identifier, the `SKProduct`, and if it's "available for purchase."

You'll start out with the list of products you think should be available, but will mark them all as not available for purchase. You will then ask the App Store which it thinks are available – for any it finds, you'll mark them as available for purchase. Obviously, you will only display those available for purchase in the store.

Most of the time these two lists will be the same, but sometimes iTunes will say a product isn't available. This could be due to:

- You added a product very recently to iTunes Connect and it hasn't fully synchronized yet.
- You hard-coded your app with a list of product identifiers, but later marked one of the In-App Purchases as no longer for sale in iTunes Connect.

Next, switch to **IAPPProduct.m** and add the implementation:

```
#import "IAPPProduct.h"

@implementation IAPPProduct

- (id)initWithProductIdentifier:(NSString *)productIdentifier {
```

```

    if ((self = [super init])) {
        self.availableForPurchase = NO;
        self.productIdentifier = productIdentifier;
        self.skProduct = nil;
    }
    return self;
}

- (BOOL)allowedToPurchase {
    if (!self.availableForPurchase) return NO;
    return YES;
}

@end

```

The only thing to mention here is the `allowedToPurchase` method. Right now this returns `YES` as long as the product is available for purchase, but you'll come back to this later and modify it to prevent the user from purchasing a product in other situations as well.

Now you'll refactor `IAPHelper` and `HMIAPHelper` to a) make use of this new class, and b) return the list of `IAPPProducts` to the caller. Let's start with `IAPHelper`.

Modify `IAPHelper.h` to look like the following:

```

typedef void (^RequestProductsCompletionHandler)
(BOOL success, NSArray * products);

@interface IAPHelper : NSObject

@property (nonatomic, strong) NSMutableDictionary * products;

- (id)initWithProducts:(NSMutableDictionary *)products;
- (void)requestProductsWithCompletionHandler:
    (RequestProductsCompletionHandler)completionHandler;

@end

```

There are two main differences here:

1. There is now an initializer that takes a dictionary. The key of the dictionary is a product identifier, and the value is an `IAPPProduct`. In other words, this is the list of products you think should be available for purchase, but you'll have to double-check with iTunes Connect.
2. The `requestProducts` method now takes a completion handler block. This way you can pass back the resulting list of products to the caller. On return, you will

pass whether it succeeded or failed, along with an array of products that are available for purchase if it succeeded.

Tip: If you are still confused about block syntax, it's time to learn it! Blocks are an incredibly useful feature in iOS and you should learn how to use them both as a user, and in your own APIs.

For more information about blocks, check out this tutorial series:

<http://www.raywenderlich.com/9328/creating-a-diner-app-using-blocks-part-1>

Next switch to **IAPHelper.m**. Add an import for the new **IAPPProduct** class you just created at the top of the file:

```
#import "IAPPProduct.h"
```

And add two new instance variables and the new initializer:

```
@implementation IAPHelper {
    SKProductsRequest * _productsRequest;
    RequestProductsCompletionHandler _completionHandler;
}

- (id)initWithProducts:(NSMutableDictionary *)products {
    if ((self = [super init])) {
        _products = products;
    }
    return self;
}
```

Straightforward so far. Next replace `requestProductsWithProductIdentifiers:` with the following:

```
- (void)requestProductsWithCompletionHandler:
(RequestProductsCompletionHandler)completionHandler {

    // 1
    _completionHandler = [completionHandler copy];

    // 2
    NSMutableSet * productIdentifiers =
        [NSMutableSet setWithCapacity:_products.count];
    for (IAPPProduct * product in _products.allValues) {
        product.availableForPurchase = NO;
        [productIdentifiers
            addObject:product.productIdentifier];
    }
}
```

```
}

// 3
_productsRequest = [[SKProductsRequest alloc]
    initWithProductIdentifiers:productIdentifiers];
_productsRequest.delegate = self;
[_productsRequest start];

}
```

Let's quickly review this:

1. Before you store a block into an instance variable, you need to copy it. This is important because if the block that is passed in is on the stack, it won't be available when you need it unless you copy it first as shown here.
2. This loops through the dictionary of products (that you think should be available) and puts each product identifier into a set – because a `SKProductsRequest` requires a set, not a dictionary.
3. Finally, you make the `SKProductsRequest`, the same as before – no changes here.

The final change for this file is to replace the `SKProductsRequestDelegate` methods as follows:

```
- (void)productsRequest:(SKProductsRequest *)request
didReceiveResponse:(SKProductsResponse *)response {

    NSLog(@"Loaded list of products...");
    _productsRequest = nil;

    // 1
    NSArray * skProducts = response.products;
    for (SKProduct * skProduct in skProducts) {
        IAPPProduct * product =
            _products[skProduct.productIdentifier];
        product.skProduct = skProduct;
        product.availableForPurchase = YES;
    }

    // 2
    for (NSString * invalidProductIdentifier in
        response.invalidProductIdentifiers) {
        IAPPProduct * product =
            _products[invalidProductIdentifier];
        product.availableForPurchase = NO;
        NSLog(@"Invalid product identifier, removing: %@", invalidProductIdentifier);
    }
}
```

```
}

// 3
NSMutableArray * availableProducts = [NSMutableArray array];
for (IAPPProduct * product in _products.allValues) {
    if (product.availableForPurchase) {
        [availableProducts addObject:product];
    }
}

_completionHandler(YES, availableProducts);
_completionHandler = nil;

}

- (void)request:(SKRequest *)request didFailWithError:(NSError *)error {

    NSLog(@"Failed to load list of products.");
    _productsRequest = nil;

    // 5
    _completionHandler(FALSE, nil);
    _completionHandler = nil;

}
```

There's a fair bit of code here, but don't worry, it's pretty simple:

1. This loops through each of the `SKProducts` that iTunes says are available, and finds the associated `IAPPProduct` in the dictionary. It then marks those products as available for purchase.
2. When you issue an `SKProductsRequest`, in addition to returning the list of `SKProducts` that are available, it also returns the list of product identifiers that are *not* available in a property called `invalidProductIdentifiers`. Here you loop through those, mark the associated identifiers as not available for purchase, and log them out. Note this step isn't strictly necessary, because you marked all products as not available for purchase before you made the products request, but it is still nice to log out when this happens for debugging purposes.
3. You then find all the products available for purchase, put them into an array, and send them to the completion handler.
4. Similarly, on a failure you also call the completion handler, but return FALSE to indicate failure.

That's it for `IAPHelper` – now time for a few tweaks to `HMIAPHelper`. Open `HMIAPHelper.h` and delete the `requestProducts` method so that the header now looks like the following:

```
#import "IAPHelper.h"

@interface HMIAPHelper : IAPHelper
+ (HMIAPHelper *)sharedInstance;
@end
```

You don't need the old `requestProducts` method anymore, because the caller will now use the new `requestProductsWithCompletionHandler` method on the superclass.

Next switch to `HMIAPHelper.m` and delete the old `requestProducts` method. Then add the following import to the top of the file:

```
#import "IAPPProduct.h"
```

And add the following new method:

```
- (id)init {
    IAPPProduct * tenHints = [[IAPPProduct alloc]
        initWithProductIdentifier:
        @"com.razeware.hangman.tenhints"];
    IAPPProduct * hundredHints = [[IAPPProduct alloc]
        initWithProductIdentifier:
        @"com.razeware.hangman.hundredhints"];
    NSMutableDictionary * products = @{
        tenHints.productIdentifier: tenHints,
        hundredHints.productIdentifier: hundredHints};
    mutableCopy];
    if ((self = [super initWithProducts:products])) {
    }
    return self;
}
```

This makes it so when you initialize the class, it sets up the products dictionary with the list of products you think should be available. And don't forget to change the product identifiers in `init` to the ones you created earlier.

OK, done with the refactoring – time to try this out and get the list of products to display!

Open `HMStoreListViewController.m`, add a few imports to the top of the file, and add two instance variables:

```
#import "IAPPProduct.h"
#import <StoreKit/StoreKit.h>
```

```
@implementation HMStoreListViewController {
    NSArray * _products;
    NSNumberFormatter * _priceFormatter;
}
```

Here you add two instance variables – one containing the list of products to display, and a number formatter that will help with formatting prices for the products using the correct locale.

Next delete the call to `requestProducts` at the end of `viewDidLoad` and replace it with the following:

```
_priceFormatter = [[NSNumberFormatter alloc] init];
[_priceFormatter
    setFormatterBehavior:NSNumberFormatterBehavior10_4];
[_priceFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];

self.refreshControl = [[UIRefreshControl alloc] init];
[self.refreshControl addTarget:self action:@selector(reload)
    forControlEvents:UIControlEventValueChanged];
[self reload];
[self.refreshControl beginRefreshing];
```

This creates a price formatter to display a value as currency. It also creates a refresh control for the table view – a cool new feature in iOS 6. For more information on this, check out Chapter 20, “What’s New with Cocoa Touch.”

When the refresh control is triggered, it will call `reload` (and you also call it here once to kick it off). So implement that next:

```
- (void)reload {
    _products = nil;
    [self.tableView reloadData];
    [[HMIAPHelper sharedInstance]
        requestProductsWithCompletionHandler:^(BOOL success,
            NSArray *products) {
            if (success) {
                _products = products;
                [self.tableView reloadData];
            }
            [self.refreshControl endRefreshing];
        }];
}
```

This starts by setting the list of products to `nil` and reloading the table view to clear out any data. Then it calls the new `requestProductsWithCompletionHandler` method

you just wrote. If it's successful, it stores the list of products in the instance variable, and reloads the table view.

Almost done! Just modify `tableView:numberOfRowsInSection:` to return the count of products:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return _products.count;
}
```

And modify `tableView:cellForRowAtIndexPath:` to populate the cell based on the product's info:

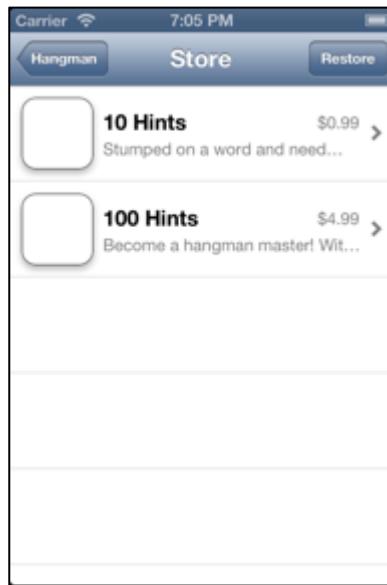
```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    HMStoreListViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];

    IAPPProduct *product = _products[indexPath.row];

    cell.titleLabel.text = product.skProduct.localizedTitle;
    cell.descriptionLabel.text =
        product.skProduct.localizedDescription;
    [_priceFormatter setLocale:product.skProduct.priceLocale];
    cell.priceLabel.text = [_priceFormatter
        stringFromNumber:product.skProduct.price];

    return cell;
}
```

And you're done! Build and run, and now when you go to the store you will see the list of products!



Congratulations – you've just made your first visual step toward In-App Purchase App Store Domination, mwuhahaha!

Showing the Detail View

Now that you have a solid framework in place, showing the detail view will be easy as pie. First open **HMStoreDetailViewController.h** and modify it to add a property for the product to display:

```
@class IAPPProduct;

@interface HMStoreDetailViewController : UIViewController

@property (nonatomic, strong) IAPPProduct * product;

@end
```

Then switch to **HMStoreDetailViewController.m** and add some imports:

```
#import "IAPPProduct.h"
#import <StoreKit/StoreKit.h>
```

And add a price formatter, similar to how you did in **HMStoreListViewController**:

```
@implementation HMStoreDetailViewController {
    NSNumberFormatter * _priceFormatter;
}
```

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

    self.view.backgroundColor = [UIColor
        colorWithPatternImage:[UIImage imageNamed:@"bg.png"]];

    _priceFormatter = [[NSNumberFormatter alloc] init];
    [_priceFormatter
        setFormatterBehavior:NSNumberFormatterBehavior10_4];
    [_priceFormatter
        setNumberStyle:NSNumberFormatterCurrencyStyle];

}
```

Finally, add the code to populate the user interface from the product:

```
- (void)refresh {
    self.title = _product.skProduct.localizedTitle;
    self.titleLabel.text = _product.skProduct.localizedTitle;
    self.descriptionTextView.text =
        _product.skProduct.localizedDescription;
    [_priceFormatter setLocale:_product.skProduct.priceLocale];
    self.priceLabel.text = [_priceFormatter
        stringFromNumber:_product.skProduct.price];
    self.versionLabel.text = @"Version 1.0";

    if (_product.allowedToPurchase) {
        self.navigationItem.rightBarButtonItem =
            [[UIBarButtonItem alloc] initWithTitle:@"Buy"
                style:UIBarButtonItemStyleBordered target:self
                action:@selector(buyTapped:)];
        self.navigationItem.rightBarButtonItem.enabled = YES;
    } else {
        self.navigationItem.rightBarButtonItem = nil;
    }

    self.pauseButton.hidden = YES;
    self.resumeButton.hidden = YES;
    self.cancelButton.hidden = YES;
}

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    self.statusLabel.hidden = YES;
```

```
[self refresh];  
}
```

Most of this is pretty straightforward, so there's not much to discuss – let's go straight to trying it out! Switch to **HMStoreListViewController.m** and add this import to the top of the file:

```
#import "HMStoreDetailViewController.h"
```

Then add the following new methods:

```
#pragma mark - Table view delegate  
  
- (void)tableView:(UITableView *)tableView  
didSelectRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    [self performSegueWithIdentifier:@"PushDetail"  
        sender:indexPath];  
}  
  
#pragma mark - Segues  
  
- (void)prepareForSegue:(UIStoryboardSegue *)segue  
sender:(id)sender {  
    if ([segue.identifier isEqualToString:@"PushDetail"]) {  
        HMStoreDetailViewController * detailViewController =  
            (HMStoreDetailViewController *)  
                segue.destinationViewController;  
        NSIndexPath * indexPath = (NSIndexPath *)sender;  
        IAPPProduct *product = _products[indexPath.row];  
        detailViewController.product = product;  
    }  
}
```

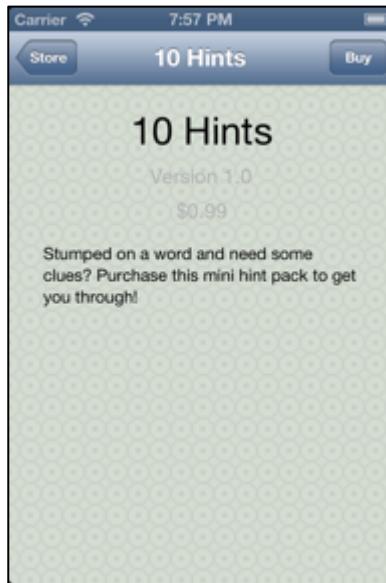
When the user taps a row, you perform the “PushDetail” segue that was previously created in the Storyboard editor (remember, the “Restore” button was hooked up to it?), passing the selected index path as a parameter.

Then in `prepareForSegue:sender:`, you know the destination view controller is the `HMStoreDetailViewController`, so you set the appropriate product to display based on the passed-in index path.

Finally, remove the implementation of `restoreTapped:` (just the code inside the method, not the method itself), since you don't need that test code anymore:

```
- (void)restoreTapped:(id)sender {  
}
```

That's it! Compile and run, and now when you tap an In-App Purchase, you see a detail screen with more information:

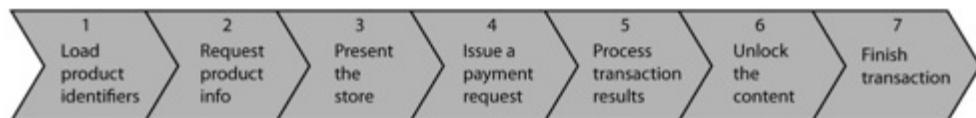


You might notice that shiny new "Buy" button in the upper right – tapping it doesn't do anything yet, but I bet you can guess what's coming next! ☺

Buying a consumable product

It's the moment you've been waiting for – allowing the user to buy a product!

In this section you will implement the remaining four steps of the process: issuing a payment request, processing the transaction results, unlocking the content, and finishing the transaction.



Feels good to be in the final leg of the race, eh? Ahh, I can imagine the money rolling in already. ☺

To keep things simple as you work on this, you're going to set this rule for your app:

"When you buy a product, you can't buy another product with the same product identifier until the first one completes."

Although this isn't strictly necessary with In-App Purchases, it's reasonable from a user's perspective and will make designing your app and user interface a lot simpler.

So to enforce this, open **IAPPProduct.h** and add a new property:

```
@property (nonatomic, assign) BOOL purchaseInProgress;
```

Then switch to **IAPPProduct.m** and modify **allowedToPurchase** as follows:

```
- (BOOL)allowedToPurchase {
    if (!self.availableForPurchase) return NO;

    if (self.purchaseInProgress) return NO;

    return YES;
}
```

Now the user won't be allowed to purchase something if iTunes Connect says it's not available, or if a purchase is in progress with that product identifier. So far, so good.

Next, you're going to add a new method to **IAPHelper** to allow the user to buy a product. Open **IAPHelper.h** and predeclare the **IAPPProduct** class at the top of the file:

```
@class IAPPProduct;
```

Then declare the new method inside **@interface**:

```
- (void)buyProduct:(IAPPProduct *)product;
```

Next, switch to **IAPHelper.m** and add the implementation as follows:

```
- (void)buyProduct:(IAPPProduct *)product {
    NSAssert(product.allowedToPurchase, @"This product isn't
        allowed to be purchased!");

    NSLog(@"Buying %@", product.productIdentifier);

    product.purchaseInProgress = YES;
    SKPayment * payment = [SKPayment
        paymentWithProduct:product.skProduct];
    [[SKPaymentQueue defaultQueue] addPayment:payment];
}
```

First it checks to make sure the product is allowed to be purchased, and if so it marks the purchase as in-progress, and issues an **SKPayment** to the **SKPaymentQueue**.

That's literally all it takes to let the user send you cold, hard cash. It's almost as if you can hear your users saying:



However, if you're letting your users give you money, you better give them something good in return! (After all, you aren't the Greek government.)

So you need to add some code to identify when a payment "transaction" has finished, and process it accordingly.

Doing so is easy. First, modify the `IAPHelper` class extension in **IAPHelper.m** to mark the class as implementing the `SKPaymentTransactionObserver`:

```
@interface IAPHelper () <SKProductsRequestDelegate,  
SKPaymentTransactionObserver>
```

Then modify your `initWithProducts:` method as follows:

```
- (id) initWithProducts:(NSMutableDictionary *)products {  
    if ((self = [super init])) {  
        _products = products;  
        [[SKPaymentQueue defaultQueue]  
         addTransactionObserver:self];  
    }  
    return self;  
}
```

Now when the `IAPHelper` is initialized, it will make itself the transaction observer of the `SKPaymentQueue`. In other words, Apple will tell you when somebody purchased something!

There's one really important thing to understand about this. It could happen that a user starts a purchase (and gets charged for it), but before Apple can respond with success or failure, the user suddenly loses network connection (or terminates your

app). The user will still expect to receive what they purchased though, or lots of rage faces will ensue!

Luckily, Apple has a solution for this in mind. The idea is that Apple will keep track of any purchase transactions that haven't yet been fully processed by your app, and will notify the transaction observer about them. But for this to work well, you should register your class as a transaction observer as early as possible in your app initialization.

To do this, switch to **HAppDelegate.m** and add this import:

```
#import "HMIAPHelper.h"
```

And then add the following line at the beginning of `application:didFinishLaunchingWithOptions:`:

```
[HMIAPHelper sharedInstance];
```

Now, as soon as your app launches it will create the singleton `HMIAPHelper`. This means the `initWithProducts:` method you just modified will be called, which registers itself as the transaction observer. So you will be notified about any transactions that were never quite finished.

You still have to implement the `SKPaymentTransactionObserver` protocol, so switch back to **IAPHelper.m** and add the following:

```
- (void)paymentQueue:(SKPaymentQueue *)queue
updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction * transaction in transactions) {
        switch (transaction.transactionState) {
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
            default:
                break;
        }
    };
}
```

Actually, this is the only required method of the protocol. It gives you a list of transactions that have been updated, and all you have to do is loop through them

and do different things based on their state. To keep the code clean, you call different methods if the transaction has completed, failed, or successfully restored.

Completed and failed makes sense, but what about restored? Remember back in the “In-App Purchase Types” section, I mentioned that there should be a way for users to restore their non-consumable purchases. This is important if the user has the same app on multiple devices (or deletes it and reinstalls it) and wants to get access to their prior purchases. Later you’ll add a way for users to do this in the app, but for now just know when the user restores their purchases, this is where it will come through.

Next for the important stuff – implementing the `completeTransaction`, `restoreTransaction`, and `failedTransaction` methods. Add these to the file:

```
- (void)completeTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"completeTransaction...");
    [self provideContentForTransaction:transaction
        productIdentifier:transaction.payment.productIdentifier];
}

- (void)restoreTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"restoreTransaction...");
    [self provideContentForTransaction:transaction
        productIdentifier:
        transaction.originalTransaction.payment.productIdentifier];
}

- (void)failedTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"failedTransaction...");
    if (transaction.error.code != SKErrorPaymentCancelled)
    {
        NSLog(@"Transaction error: %@", transaction.error.localizedDescription);
    }

    IAPProduct * product =
        _products[transaction.payment.productIdentifier];
    [self notifyStatusForProductIdentifier:
        transaction.payment.productIdentifier
        string:@"Purchase failed."];
    product.purchaseInProgress = NO;
    [[SKPaymentQueue defaultQueue]
        finishTransaction: transaction];
```

```
}
```

`completeTransaction` and `restoreTransaction` do the same thing – they call a helper function to provide the given content (which you'll write next).

`failedTransaction` is a bit different though. It calls a helper method to notify the user that the purchase failed (which you'll also write next), marks the purchase as no longer in progress, and finishes the transaction.

Note: It is very important to call `finishTransaction`, or StoreKit will not know you've finished processing it, and will continue delivering the transaction to your app each time it launches!

Finally, add the remaining methods to the file:

```
- (void)notifyStatusForProductIdentifier:  
    (NSString *)productIdentifier string:(NSString *)string {  
    IAPPProduct * product = _products[productIdentifier];  
    [self notifyStatusForProduct:product string:string];  
}  
  
- (void)notifyStatusForProduct:(IAPPProduct *)product  
    string:(NSString *)string {  
  
}  
  
- (void)provideContentForTransaction:  
    (SKPaymentTransaction *)transaction  
    productIdentifier:(NSString *)productIdentifier {  
  
    IAPPProduct * product = _products[productIdentifier];  
  
    [self provideContentForProductIdentifier:productIdentifier];  
    [self notifyStatusForProductIdentifier:productIdentifier  
        string:@"Purchase complete!"];  
  
    product.purchaseInProgress = NO;  
    [[SKPaymentQueue defaultQueue] finishTransaction:  
        transaction];  
  
}  
  
- (void)provideContentForProductIdentifier:  
    (NSString *)productIdentifier {
```

```
}
```

The first method, `notifyStatusForProductIdentifier`, just finds the appropriate product and then calls the next method, `notifyStatusForProduct`, which is empty! What gives?

Well, you will implement that in the subclass (`HMIAPHelper`), because what you do with that message is app-dependent.

`provideContentForTransaction` is similar to `failedTransaction` in that it marks the purchase as complete and finishes the transaction. But before that, it also calls the `provideContentForProductIdentifier`, which is also empty! Again, that will be implemented in the subclass.

So let's get all this subclass business out of the way. Open **HMIAPHelper.m** and add the following imports to the top of the file:

```
#import "HMContentController.h"
#import "JSNotifier.h"
#import <StoreKit/StoreKit.h>
```

You might be wondering what `JSNotifier.h` is. It's a handy helper class written by Jonah Siegle that temporarily pops up a message at the bottom of the screen. It's good for alerts, error messages and the like. You'll be using it to display the "notifications" mentioned earlier.

Next add the following methods to the bottom of the file:

```
- (void)provideContentForProductIdentifier:
(NSString *)productIdentifier {
    if ([productIdentifier
        isEqualToString:@"com.razeware.hangman.tenhints"]) {
        int curHints =
            [HMContentController sharedInstance].hints;
        [[HMContentController sharedInstance] setHints:
            curHints + 10];
    } else if ([productIdentifier
        isEqualToString:@"com.razeware.hangman.hundredhints"]) {
        int curHints =
            [HMContentController sharedInstance].hints;
        [[HMContentController sharedInstance] setHints:
            curHints + 100];
    }
}

- (void)notifyStatusForProduct:(IAPPProduct *)product
string:(NSString *)string {
```

```
NSString * message = [NSString stringWithFormat:@"%@: %@",
    product.skProduct.localizedTitle, string];
JSNotifier *notify =
    [[JSNotifier alloc] initWithTitle:message];
[notify showFor:2.0];
}
```

As you can see, `provideContentForProductIdentifier:` simply increases the number of available hints based on the product identifier. (Remember to change the product identifiers here to your own identifiers.) And as mentioned, `notifyStatusForProduct:` uses the `JSNotifier` library to pop up an alert with the message.

The final step is to call your new `buyProduct:` method in the store detail view controller. So open **HMStoreDetailViewController.m** and add this import to the top of the file:

```
#import "HMIAPHelper.h"
```

Then replace the current empty implementation of `buyTapped` with the following:

```
- (void)buyTapped:(id)sender {
    NSLog(@"Buy tapped!");
    [[HMIAPHelper sharedInstance] buyProduct:self.product];
}
```

Time to try this out and purchase a product!

Of course, to buy a product, you'll need an account on the App Store sandbox. If you don't already have a sandbox account, log onto [iTunes Connect](#) and select **Manage Users**. Choose **Test User**, and then **Add New User**. Fill out the form and click **Save**.



Next, if you're testing on an actual device make sure you're logged out from the App Store. Go to **Settings\iTunes & App Stores**, and log out if you are logged in.

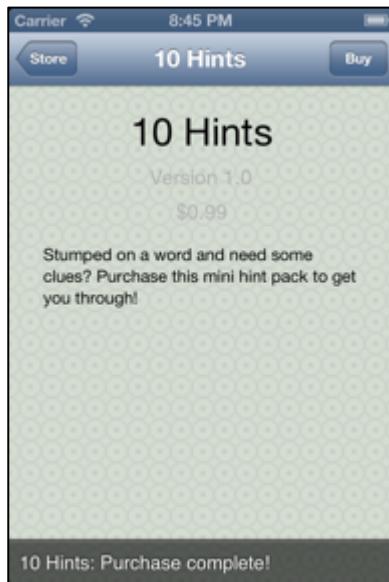
Finally, time to test! Build and run, open up a product, and tap the Buy button. A popup should appear, like this:



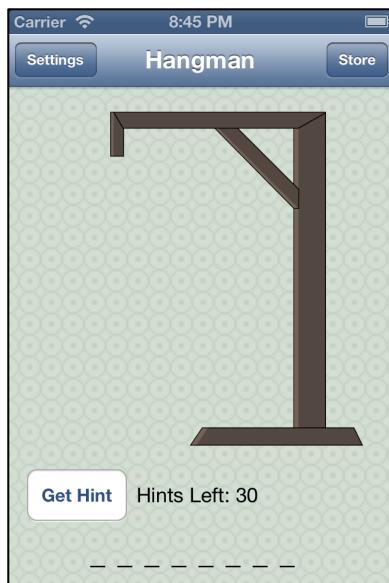
Tap **Buy**, and another dialog should appear (since you just verified you're not already signed in):



Tap "Use Existing Apple ID" and enter the login information for your Test User that you created with iTunes Connect. A few minutes later, you should see a popup showing that your purchase is complete!



You can verify this by going back to the main game screen, where you should see that your number of hints has increased:



Feel free to play around with this a bit more – in fact, purchase as many hints as you like, because on the sandbox all purchases are free. It's kind of like being Warren Buffet for a day!

Buying a non-consumable product

Awesome – you've added the ability for the users to buy additional hints (and as often as they'd like!), so you're on your way to the big bucks already.

But, you started thinking... you bet you could earn even more money if you sell additional packs of words! You know some of your users are gluttons for punishment and want more difficult words, and you know some of your users are geeks among geeks – in other words, iOS devs!

Since you have wisely designed your word list functionality to be file-based (in fact, it's simply a single .plist file), it won't take much time to whip a few more word lists together. You decide to create a new "Hard Words" plist and an "iOS Words" plist, with the goal of turning them into some new In-App Purchases for sale in the store.

There are five steps to make this all happen:

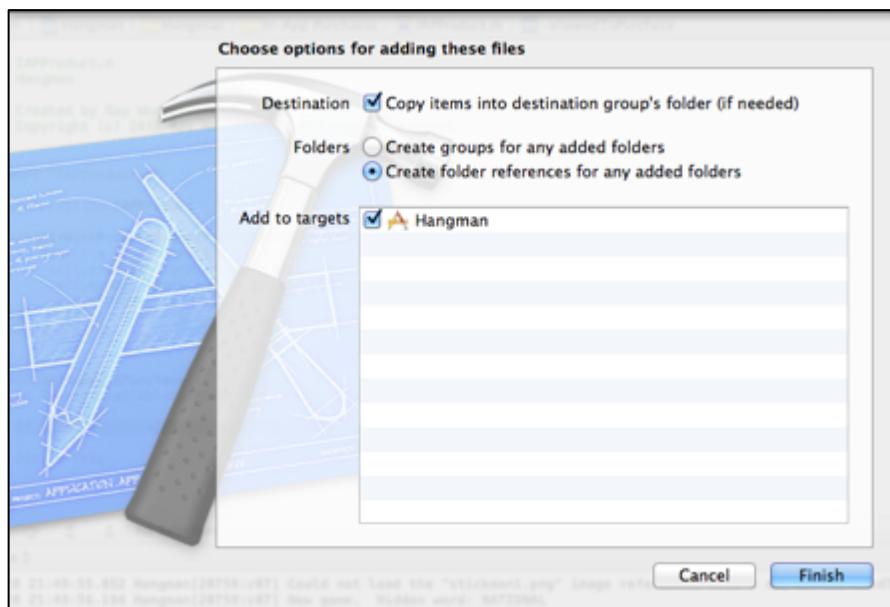
1. Testing the Words
2. Registering the In-App Purchases
3. Modifying the Code
4. Restoring Purchases
5. Finishing Touches

Let's not delay – there are stickmen to hang!

Testing the words

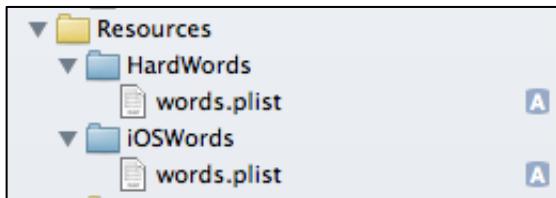
Before doing anything with In-App Purchases, the first step is to make sure that the content you're providing actually works.

So let's add the new word lists to your app. I've already created these files for you – you'll find them in the resources for this chapter, in a directory called **ExtraWords**. Select the **HardWords** and **iOSWords** folders, drag them into the **Resources** group of your Xcode project, and this popup will appear:



Make sure that **Copy items into destination group's folder (if needed)** is checked, **Create folder references for any added folders** is selected, and that the **Hangman** target is checked, and click Finish.

At this point, your Resources folder in the Project Navigator should look like the following:



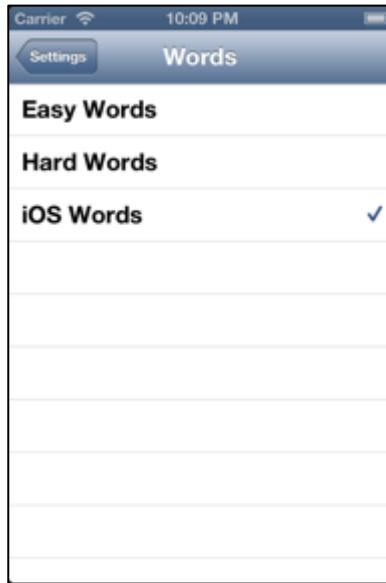
Note: As a reminder, the blue folders represent subdirectories that will be added to your Application Bundle, instead of the default behavior that copies the files to the root of the bundle. This way, each words.plist will be safely contained within its own directory, and there will be no name conflicts. For more details on this design decision, see the “Design Considerations” section earlier in this chapter.

Now that you have the new word lists in your bundle, try them out. Open **HMContentController.m** and add the following lines to `init`, after the other call to `unlockWordsWithDirURL`:

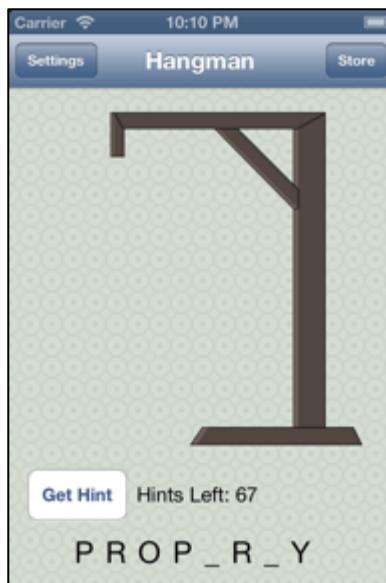
```
[self unlockWordsWithDirURL:[resourceURL  
    URLByAppendingPathComponent:@"HardWords"]];  
[self unlockWordsWithDirURL:[resourceURL  
    URLByAppendingPathComponent:@"iOSWords"]];
```

Here you unlock the content when the app first starts up, every time. Obviously you want to eventually make these In-App Purchases instead, but this is a good way to test them out and make sure they work.

Build and run your app, go to Settings, and change the word list to iOS Words.



Then test out your geek cred by playing a game – but this time, a mistake isn't a compiler error, it's death! ☺



Registering the In-App Purchases

Now that you know they are working, delete the lines of code you just added to **HMContentController.m** to test out the HardWords and iOSWords.

It's time to register the new In-App Purchases in iTunes Connect! The process is very similar to what you did before in the "Getting Started" section, except you will be making **non-consumable** purchases this time.

Open [iTunes Connect](#) and click **Manage Your Applications**. Find your Hangman app in the list, open it, and click **Manage In-App Purchases**. Click **Create New**, choose **Non-Consumable\Select**, and enter the following information:

The screenshot shows the 'In-App Purchase Summary' section. It includes fields for 'Reference Name' (HardWords) and 'Product ID' (com.raywenderlich.hangman.hardwords). Below these, there's a 'Pricing and Availability' section with options for 'Cleared for Sale' (Yes selected), 'Price Tier' (Tier 1), and a 'View Pricing Matrix' link.

Don't forget to modify your Product ID so that it's based on your own Bundle Identifier – for example, yours might look like com.yourdomain.hangman.hardwords.

Scroll down to the **In-App Purchase Details\Language** section and click **Add Language**. Fill out the popup as follows:

The 'Add Language' dialog box contains fields for 'Language' (English), 'Display Name' (Hard Words), and 'Description' (A set of over 100 new words for Hangman – all at insane level difficulty! Test your might – or stump your friends!). At the bottom are 'Cancel' and 'Save' buttons.

Click **Save**, and then scroll down to the bottom of the page and click **Save** again. Now repeat for iOS Words – here is the relevant info for both sections:

In-App Purchase Summary

Enter a reference name and a product ID for this In-App Purchase. You must also add at least one language, along with a display name and a description in that language.

Reference Name: iOSWords [?](#)

Product ID: com.raizewares.hangman.iOSwords [?](#)

Pricing and Availability

Enter the pricing and availability details for this In-App Purchase below.

Cleared for Sale Yes No

Price Tier [Tier 1](#) [View Pricing Matrix](#) [?](#)

Add Language

Language English [?](#)

Display Name iOS Words [?](#)

Description Geeks rejoice – expand your game with words related to iOS development! This time, the penalty for entering the wrong keyword isn't an error – it's death! :]

[Cancel](#) [Save](#)

Again, don't forget to modify your Product ID appropriately!

Modifying the code

Good news – you've already made most of the changes you need for non-consumable In-App Purchases already, thanks to the work you did earlier with consumables. There are three main differences, however:

1. You have to keep a record of whether the user has purchased the product so that when the user restarts your app, you unlock the content (or not) appropriately.
2. A given user can purchase a given non-consumable product only once, rather than unlimited times as with consumable products.
3. Users want to be able to access the words they bought on their other devices.

Let's start by just getting things working and addressing the first two items – you'll deal with restoring purchases later.

First, open **IAPProduct.h** and add a new variable to keep track of whether an IAPProduct has been purchased before:

```
@property (nonatomic, assign) BOOL purchase;
```

And in **IAPPProduct.m**, add a case in `allowedToPurchase` to return `NO` if the product has been purchased already:

```
- (BOOL)allowedToPurchase {
    if (!self.availableForPurchase) return NO;

    if (self.purchaseInProgress) return NO;

    if (self.purchase) return NO;

    return YES;
}
```

Believe it or not, you don't have to make any changes to `IAPHelper` – you already have a firm foundation there already!

You do have to make a few changes to `HMIAPHelper`, though. First, add a new method to unlock a set of words with a given product identifier and directory:

```
- (void)unlockWordsForProductIdentifier:(NSString *)productIdentifier directory:(NSString *)directory {
    // 1
    IAPPProduct * product = self.products[productIdentifier];
    product.purchase = YES;

    // 2
    [[NSUserDefaults standardUserDefaults] setBool:YES
        forKey:productIdentifier];
    [[NSUserDefaults standardUserDefaults] synchronize];

    // 3
    NSURL * resourceURL = [NSBundle mainBundle].resourceURL;
    [[HMContentController sharedInstance]
        unlockWordsWithDirURL:[resourceURL
            URLByAppendingPathComponent:directory]];
}
```

There are three main steps here:

1. This bit of code finds the product for this product identifier, and marks it as purchased.
2. The simplest way to keep track of whether something has been purchased or not is to set an `NSUserDefaults`, as shown here.
3. Finally, the content is unlocked by calling the method in `HMContentController`.

Now that you have this handy method, let's make use of it. First add the following statements to the end of the if/else chain in `provideContentForProductIdentifier`:

```
else if ([productIdentifier  
 isEqualToString:@"com.razeware.hangman.hardwords"]) {  
     [self unlockWordsForProductIdentifier:  
      @"com.razeware.hangman.hardwords" directory:@"HardWords"];  
 } else if ([productIdentifier  
 isEqualToString:@"com.razeware.hangman.ioswords"]) {  
     [self unlockWordsForProductIdentifier:  
      @"com.razeware.hangman.ioswords" directory:@"iOSWords"];  
 }
```

Remember, this method gets called after a purchase transaction successfully completes. When this occurs, you simply call your new method to unlock the content.

As always, don't forget to replace these product identifiers with your own.

There's one other place you need to call your new unlock method – on app startup, if the `NSUserDefaults` flag you set to indicate whether the content is available is there.

Replace your `init` method in **HMIAPHelper.m** with the following:

```
- (id)init {  
     IAPPProduct * tenHints = [[IAPPProduct alloc]  
     initWithProductIdentifier:  
      @"com.razeware.hangman.tenhints"];  
     IAPPProduct * hundredHints = [[IAPPProduct alloc]  
     initWithProductIdentifier:  
      @"com.razeware.hangman.hundredhints"];  
 // 1  
     IAPPProduct * hardWords = [[IAPPProduct alloc]  
     initWithProductIdentifier:  
      @"com.razeware.hangman.hardwords"];  
     IAPPProduct * iosWords = [[IAPPProduct alloc]  
     initWithProductIdentifier:  
      @"com.razeware.hangman.ioswords"];  
     NSMutableDictionary * products = @{@"  
         tenHints.productIdentifier": tenHints,  
         hundredHints.productIdentifier": hundredHints,  
         hardWords.productIdentifier": hardWords,  
         iosWords.productIdentifier": iosWords} mutableCopy];  
     if ((self = [super initWithProducts:products])) {  
 // 2  
         if ([[NSUserDefaults standardUserDefaults]
```

```
        boolForKey:@"com.razeware.hangman.hardwords"]) {  
    [self unlockWordsForProductIdentifier:  
        @"com.razeware.hangman.hardwords"  
        directory:@"HardWords"];  
}  
if ([[NSUserDefaults standardUserDefaults]  
    boolForKey:@"com.razeware.hangman.ioswords"]) {  
    [self unlockWordsForProductIdentifier:  
        @"com.razeware.hangman.ioswords"  
        directory:@"iOSWords"];  
}  
}  
return self;  
}
```

There are two changes here:

1. The new non-consumable product identifiers are added to the list of `IAPPProducts`. This way, the `IAPHelper` superclass will add these to the list of products in its `SKProductsRequest`.
2. On startup, it checks to see if the flags that say the word lists have been unlocked are set, and unlocks them if appropriate.

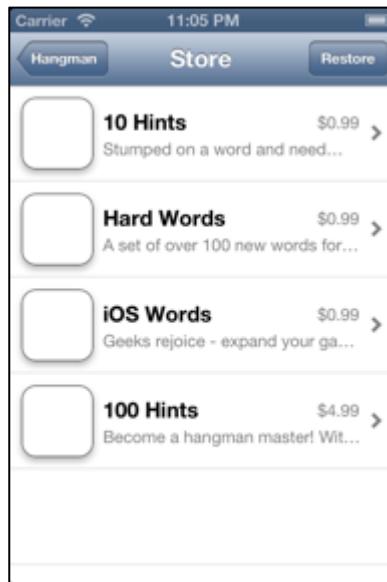
As always, don't forget to replace these product identifiers with your own.

You're almost ready to try this out! Open **HMStoreListViewController.m** and inside `tableView:cellForRowAtIndexPath:`, replace the line that sets the `cell.priceLabel.text` with the following:

```
if (product.purchase) {  
    cell.priceLabel.text = @"Installed";  
} else {  
    cell.priceLabel.text = [_priceFormatter  
        stringFromNumber:product.skProduct.price];  
}
```

This makes it so that when a user looks at a product in the store list, if it's already been purchased it will be marked as Installed rather than showing the price, as a user would expect. Of course, this will never happen for non-consumables, because the new purchase flag on `IAPPProduct` is never set to `YES` for those.

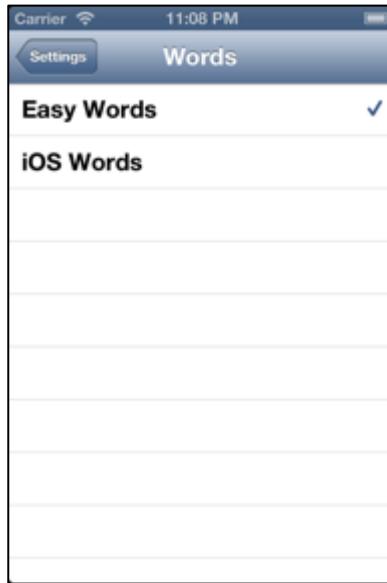
Build and run the project and go to the store. You should see your new non-consumable In-App Purchases in the list:



Open iOS Words, and purchase it. You should see a popup that shows the purchase as successful:



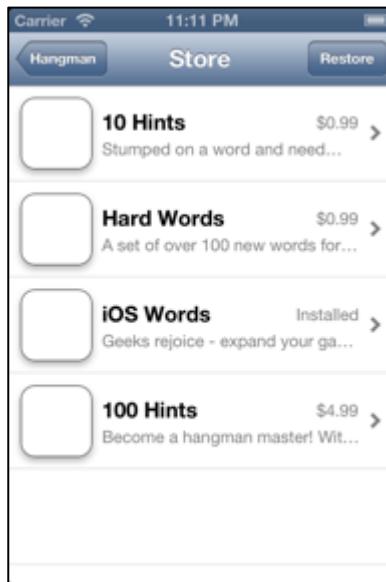
Go back to the main game, then **Settings\Words**. You should see your new purchase in the list. (You might have noticed that the iOS Words item is not immediately marked as “installed” as you move through the store listing. If you come back to the listing, it’ll be marked correctly – you’ll fix this later on.)



Feel free to select your new purchase and enjoy the game!



Finally, try shutting down the app and starting it back up again, and open up the store. You should see your purchase marked as installed:



Restoring purchases

If you have non-consumable In-App Purchases in your app, Apple says you must implement the ability for users to restore purchases they have made in the past – or risk rejection. Don't worry – it's almost trivial to implement, so it's a no-brainer to add!

First, open **IAPHelper.h** and declare a new method in the `@interface`:

```
- (void)restoreCompletedTransactions;
```

And add the implementation in **IAPHelper.m**:

```
- (void)restoreCompletedTransactions {
    [[SKPaymentQueue defaultQueue]
     restoreCompletedTransactions];
}
```

This one-liner gets StoreKit to connect to the App Store, figures out every non-consumable that the user has purchased in the past, and eventually calls your `paymentQueue:updatedTransactions` method with the `SKPaymentTransactionStateRestored` case. Since you already handle this in `paymentQueue:updatedTransactions:` (and unlock the content when this occurs), that's all you have to do. It doesn't get much simpler than that!

Let's hook up the GUI to this, and with a nice alert view, too. Open **HMStoreListViewController.m** and add a new class extension to mark the class as implementing `UIAlertViewDelegate`:

```
@interface HMStoreListViewController() <UIAlertViewDelegate>
```

```
@end
```

Then replace `restoreTapped:` with the following (and add a new method):

```
- (void)restoreTapped:(id)sender {
    NSLog(@"Restore tapped!");
    UIAlertView *alertView = [[UIAlertView alloc]
        initWithTitle:@"Restore Content"
        message:@"Would you like to check for and restore any
previous purchases?"
        delegate:self
        cancelButtonTitle:@"Cancel"
        otherButtonTitles:@"OK", nil];
    alertView.delegate = self;
    [alertView show];
}

#pragma mark - UIAlertViewDelegate

- (void)alertView:(UIAlertView *)alertView
didDismissWithButtonIndex:(NSInteger)buttonIndex {
    if (buttonIndex == alertView.firstOtherButtonIndex) {
        [[HMIAPHelper sharedInstance]
            restoreCompletedTransactions];
    }
}
```

Delete the app from your device, then build and run it again, go to the store and tap Restore, and then OK. Your purchases should be restored, and you will see a popup notification about this.

Of course, you'll notice the list view does not refresh correctly – but you'll fix that next!

Finishing touches

Before you move on, there are two finishing touches you should add to the current workflow.

First, you may have noticed that after you purchase a non-consumable product, the Buy button remains on the page. If you tap the Buy button, the app will crash, because it has an assert to make sure the product is allowed to be purchased (and of course it's not, because you already purchased it).

The Buy button remains on the page because right now the store details view controller has no knowledge of any changes that occur on the product, such as a product being purchased. There are multiple ways you could notify the details view

controller about this (notifications, delegates, and more), but in this case you're going to use Key-Value Observing (KVO).

If you're new to KVO it might sound scary, but it's quite simple, really. KVO allows you to "pay attention" to any changes on any property of an object. So you'll just "pay attention" to when the `purchaseInProgress` or `purchase` fields on the product change, so you can refresh when that occurs.

Let's see how this looks in code. Go to **HMStoreDetailViewController.m** and replace `viewWillAppear` with the following (and add two new methods):

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    selfStatusLabel.hidden = YES;
    [self refresh];
    [self.product addObserver:self
        forKeyPath:@"purchaseInProgress" options:0 context:nil];
    [self.product addObserver:self
        forKeyPath:@"purchase" options:0 context:nil];
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self.product removeObserver:self
        forKeyPath:@"purchaseInProgress" context:nil];
    [self.product removeObserver:self
        forKeyPath:@"purchase" context:nil];
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
    context:(void *)context {
    [self refresh];
}
```

When the view controller appears, it starts "paying attention" to the product, and when it disappears it stops. When a change occurs, `observeValueForKeyPath:ofObject:change:context` is called – and here it simply refreshes itself.

The second finishing touch is similar – the store list controller needs to know about changes to the products it is watching and respond accordingly. Open up **HMStoreListViewController.m** and add a new instance variable to keep track of whether it's observing its products:

```
@implementation HMStoreListViewController {
    NSArray * _products;
```

```
    NSNumberFormatter * _priceFormatter;
    BOOL _observing;
}
```

Then add the following code to the file:

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

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self removeObservers];
}

#pragma mark - KVO

- (void)addObservers {
    if (_observing || _products == nil) return;
    _observing = TRUE;
    for (IAPPProduct * product in _products) {
        [product addObserver:self
            forKeyPath:@"purchaseInProgress" options:0 context:nil];
        [product addObserver:self forKeyPath:@"purchase" options:0
            context:nil];
    }
}

- (void)removeObservers {
    if (!_observing) return;
    _observing = FALSE;
    for (IAPPProduct * product in _products) {
        [product removeObserver:self
            forKeyPath:@"purchaseInProgress" context:nil];
        [product removeObserver:self forKeyPath:@"purchase"
            context:nil];
    }
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
    context:(void *)context {
    IAPPProduct * product = (IAPPProduct *)object;
```

```
int row = [_products indexOfObject:product];
NSIndexPath * indexPath = [NSIndexPath indexPathForRow:row
    inSection:0];
[self.tableView reloadRowsAtIndexPaths:@[indexPath]
    withRowAnimation:UITableViewRowAnimationNone];
}

- (void)setProducts:(NSArray *)products {
    [self removeObservers];
    _products = products;
    [self addObservers];
}
```

This includes methods to add and remove observers on all the products in the array. When the observer fires, it reloads the appropriate row in the table view. The observers are registered/deregistered whenever the list of products changes.

Also, `viewWillAppear` and `viewWillDisappear` register/unregister the observers. And when the view first appears, it refreshes all the products in the table view.

Oh, and one final thing: modify the `reload` method to use the new `setProducts` method:

```
- (void)reload {
    [self setProducts:nil];
    [self.tableView reloadData];
    [[HMIAPHelper sharedInstance]
    requestProductsWithCompletionHandler:^(BOOL success, NSArray
    *products) {
        if (success) {
            [self setProducts:products];
            [self.tableView reloadData];
        }
        [self.refreshControl endRefreshing];
    }];
}
```

Build and run, and now after you buy a non-consumable product, the detail view controller will notice what's going on and make the Buy button disappear. And if you go back to the list view, it will show the item as installed. Similarly, if you restore purchases, the list view pays attention and updates itself correctly.

My, aren't you attentive? To celebrate your newfound attention skills, check out this YouTube video and see if you can pass the test!

<http://www.youtube.com/watch?v=vJG698U2Mvo>

Receipt validation

When you make an In-App Purchase, you can't 100% trust that the response that says "everything went OK" really came from Apple without using a technique called "receipt validation."

When you make an In-App Purchase, Apple sends you back a special piece of data called a "receipt." This is a private piece of data that records cryptographically-signed information about the transaction. The idea is that for your app to be secure, you shouldn't blindly trust that a purchase completed – you should send the receipt to a special "receipt validation" server that Apple has set up to double-check that everything is OK.

The dangers of not performing receipt validation were proven pretty spectacularly recently. A Russian hacker developed an easy-to-install In-App Purchase hack that allows users to receive almost any In-App Purchase for free – at least, if the app isn't doing proper receipt validation.

The hack is a classic "man in the middle" attack. You configure DNS records so your device is routed to the "hack" servers rather than Apple, and you configure a fake certificate on your device so the "hack server" you're connected to is trusted. Then, whenever you make a request to make an In-App Purchase, the request goes to the "hack server" instead. The hack server will always say "done and paid for!" so the app will unlock the content without knowing it's been had – for free!

FYI, this hack will no longer work on iOS 6. However, there are other variants attackers could employ in the future. So it's still a good idea to use receipt validation.

Note: For more information on the In-App Purchase hack discussed above, check out this great article:

http://www.macworld.com/article/1167677/hacker_exploits_ios_flaw_for_free_in_app_purchases.html

The no-server solution

Apple's official recommendation to perform receipt validation is to connect to your own server, which then connects to Apple's servers to validate the receipts.

For a number of reasons, this is more secure than connecting to Apple directly, and you'll be taking this approach in the next chapter. But in this chapter, the focus is on the simplest and easiest way to implement In-App Purchases. Right now, it would be a major pain to have to set up your own server just to validate receipts.

A lot of other developers feel the same way, and so they wrote code to connect to Apple's validation server directly rather than going through an intermediate server

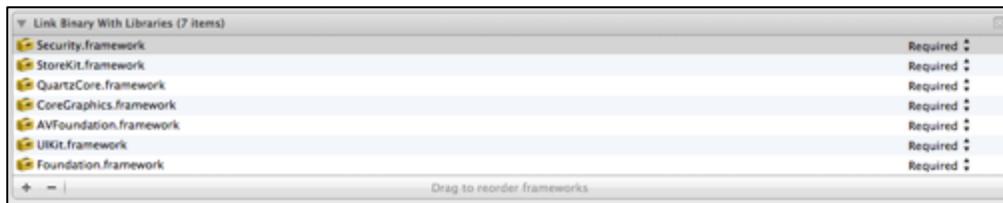
(despite Apple's recommendations). It became so common that Apple provided some sample code demonstrating a fairly secure way to verify receipts by connecting to Apple's servers directly.

In this section, you're going to integrate Apple's provided code into the app to validate receipts before unlocking the purchases.

Note: The original code was missing some pieces (such as the base64 routines) as well as some logic (like returning results to a caller), which I added in. Also, I cannot vouch 100% for the robustness of the Apple code, as it looks a little thrown together, so use at your own risk!

Inside the resources for this chapter, you will find a folder called **VerificationController**. Drag this folder into your **libs** folder, make sure **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, and the **Hangman** target is checked, and click **Finish**.

This code requires the Security framework, so let's add it to your project. To do this, click on your **Hangman** project root in the Project Navigator and then the **Hangman** target. Select the **Build Phases** tab, scroll down to the **Link Binary with Libraries** section, expand it if necessary, and click the **+** button. Select **Security.framework**, and click **Add**. At this point, your list of libraries should look like the following:



Then switch to **IAPHelper.m** and add the following import to the top of the file:

```
#import "VerificationController.h"
```

And add the following method:

```
- (void)validateReceiptForTransaction:  
    (SKPaymentTransaction *)transaction {  
  
    IAPProduct * product =  
        _products[transaction.payment.productIdentifier];  
    VerificationController * verifier =  
        [VerificationController sharedInstance];
```

```
[verifier verifyPurchase:transaction
    completionHandler:^(BOOL success) {
        if (success) {
            NSLog(@"Successfully verified receipt!");
            [self provideContentForTransaction:transaction
                productIdentifier:
                transaction.payment.productIdentifier];
        } else {
            NSLog(@"Failed to validate receipt.");
            product.purchaseInProgress = NO;
            [[SKPaymentQueue defaultQueue]
                finishTransaction: transaction];
        }
    }];
}
```

This method simply calls Apple's (somewhat modified) code to verify the transactions, and either provides the content or not, based on the results.

The final step is to call this new method from `completeTransaction` and `restoreTransaction`, instead of providing the content right away. Replace those two methods with the following:

```
- (void)completeTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"completeTransaction...");
    [self validateReceiptForTransaction:transaction];
}

- (void)restoreTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"restoreTransaction...");
    [self validateReceiptForTransaction:transaction];
}
```

Build and run, and your app should work as usual – but now with a great deal more security!

Note: If you are going to use this verification controller in your app, remember that it is currently configured to use the sandbox server (see the `verifyPurchase:completionHandler:` method). You will need to switch this to production before you release your app.

Where to go from here?

Congratulations! You now have hands-on experience adding both consumable and non-consumable In-App Purchases to your app, and have made a fun and extensible Hangman game along the way!

The best part is that the `IAPHelper` class that you've been creating is completely reusable in your own apps, so you can start integrating In-App Purchases into your apps right away – and hopefully rake in the big bucks. ☺

In this chapter, you hard-coded the purchases directly into your app. This is definitely the easiest way to get things working, and is well suited for a large portion of apps.

However, if you have an app that makes heavy use of In-App Purchases and you want to frequently add new purchases, having to update your app every time you add a new purchase can be annoying. You need another way.

That's exactly what you'll learn in the next chapter:

- How to use iOS 6's new Hosted Content feature to download your In-App Purchases over the network.
- How to get rid of all the hard-coded product info that's currently in your app and move it to a web server.
- How to perform receipt validation through your own server.
- How to use iOS 6's new `SKStoreProductViewController` to allow users to purchase content from the App Store without leaving your app.
- And much more!

If this sounds right up your alley, keep reading, but if the simple implementation that you learned about in this chapter is sufficient for your needs, I wish you the best of luck with In-App Purchases and hope it earns you a lot of money!

And I only ask for a 10% cut (just kidding!). ☺

Chapter 10: Intermediate In-App Purchases

By Ray Wenderlich

Welcome back to our epic journey through the world of In-App Purchases!

At this point, you should be pretty comfortable with the basics: registering In-App Purchases in iTunes Connect, retrieving product info, displaying the list of products, making purchases, and handling both consumable and non-consumable purchases.

However, up to this point you've been focusing on doing things the quickest and easiest way – hard-coding the purchases directly into your app bundle. This technique, although good for simple apps, doesn't leverage the full potential of the In-App Purchase system.

The best way to use In-App Purchases is to download the contents from a server instead of bundling them with your app. This approach has several benefits:

- **No app update required.** By retrieving your In-App Purchases from a server, you can add new purchases without having to do an app update. This is handy if you are adding (or tweaking) In-App Purchases frequently, or if you want non-programmer types to be able to add new In-App Purchases without your involvement.
- **Reduced binary size.** At the time of writing this chapter, if your app is over 50 MB, it cannot be downloaded from the App Store using an over-the-air connection (like your cellular connection). So if your app is over 50 MB, you are probably losing money. I can't tell you how many times I've tried to download an app while on the road, and it failed because the app was too big, and I never tried to download it again from home (hence the developers lost the sale). One good way to reduce binary size is to remove the optional components from your app bundle, and have users download them on demand with In-App Purchases instead.
- **Increased Security.** Since the In-App Purchases are not pre-included in your app bundle, it becomes more difficult for a hacker to simply patch your app to get free access to your In-App Purchases. Also, you can implement additional security measures by using your own server for receipt validation.

In this chapter, you're going to take the Hangman game that you worked on in the previous chapter and make it completely server-based. It will get its product information from your own server and support downloadable In-App Purchases.

In the process, you'll learn about some killer new In-App Purchase capabilities introduced in iOS 6:

- **Hosted Content** is a new feature in iOS 6 whereby Apple hosts your downloads for you. This saves you a ton of implementation time, and it also saves you from having to pay hosting fees for the downloads!
- **SKStoreProductViewController** is a new feature in iOS 6 allowing you to display the iTunes Store and sell items from directly within your app. This is a great way to allow your users to purchase related content without having to leave your app.

By the end of this chapter, you will know how to make an extremely flexible and robust server-based implementation you can use in your own apps. In short, you will become an In-App Purchase boss!



Moving to your own server

Note: If you skipped the last chapter and are starting this topic here, you can find the Hangman project we've been working on in the resources for this chapter named **HangmanCh10Starter**. Open it up and take a look around, and make sure you understand everything. If you're uncertain about something, flip back to the previous chapter – it's all explained there.

To have full control over the app, you should create a new app for yourself in iTunes Connect with your own In-App Purchases, and modify the product identifiers/Bundle ID in the app appropriately.

If you are confused about how to do this, refer back to the previous chapter.

Before you begin, open up **HMIAPHelper.m** to refresh your memory with what's going on in there. You'll see that the list of products to load is hard-coded in the `init` method:

```
- (id) init {
    IAPPProduct * tenHints = [[IAPPProduct alloc]
        initWithProductIdentifier:
        @"com.razeware.hangman.tenhints"];
    IAPPProduct * hundredHints = [[IAPPProduct alloc]
        initWithProductIdentifier:
        @"com.razeware.hangman.hundredhints"];
    IAPPProduct * hardWords = [[IAPPProduct alloc]
        initWithProductIdentifier:
        @"com.razeware.hangman.hardwords"];
    IAPPProduct * iosWords = [[IAPPProduct alloc]
        initWithProductIdentifier:
        @"com.razeware.hangman.ioswords"];
    NSMutableDictionary * products = @{
        tenHints.productIdentifier: tenHints,
        hundredHints.productIdentifier: hundredHints,
        hardWords.productIdentifier: hardWords,
        iosWords.productIdentifier: iosWords} mutableCopy];
    if ((self = [super initWithProducts:products])) {
        // ...
    }
    return self;
}
```

Also, there's a ton of code with logic hard-coded to specific In-App Purchases, like this:

```
- (void) provideContentForProductIdentifier:
(NSString *) productIdentifier {
    if ([productIdentifier
        isEqualToString:@"com.razeware.hangman.tenhints"]) {
        int curHints =
            [HMContentController sharedInstance].hints;
        [[HMContentController sharedInstance] setHints:
            curHints + 10];
    } else if ([productIdentifier
        isEqualToString:@"com.razeware.hangman.hundredhints"]) {
        int curHints = [HMContentController
            sharedInstance].hints;
        [[HMContentController sharedInstance] setHints:curHints
            + 100];
    }
}
```

```
// . . .  
}
```

If you're trying to support downloadable In-App Purchases, all of this hard-coded logic is not good. The whole point is that you want to be able to add new purchases without having to update your app, but with hard-coded lists like this, it's impossible.

So in this section, you are going to move the list of products off to your own server, along with some additional information about each of the products, like their icon and what they should unlock.

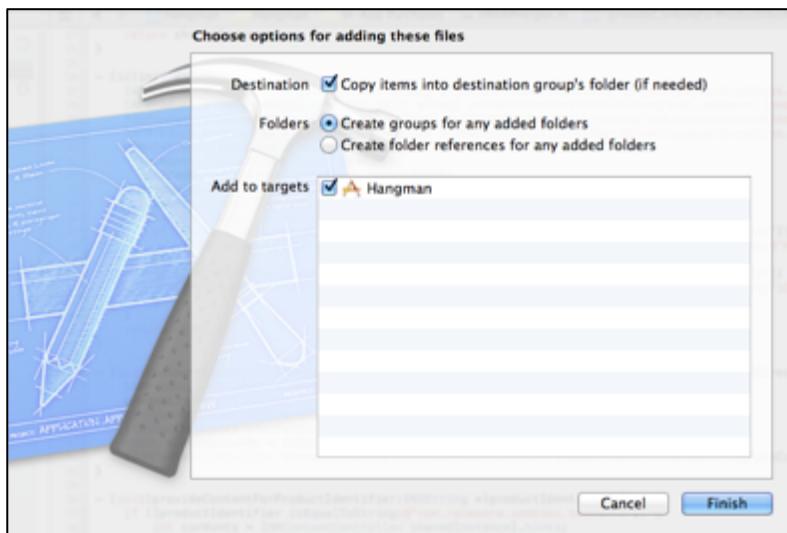
This will involve a fair bit of refactoring of the current code. If you are eager to get straight to the shiny new iOS 6 stuff, feel free to jump ahead to the next section, "Hosted Content" – the resources for this chapter includes the refactored project that you can use to continue.

But if you're curious about how to get this working and want to follow along with converting Hangman to a server-based app, read on!

Moving product info to a file

The first step is to move the list of products away from code and into a file. At first, you'll just put the file in your app's bundle, but eventually you'll store it on your own server.

Inside the resources for this chapter, you will find a folder called **IAPInfo**. Drag this folder into your Xcode project, make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, the **Hangman** target is checked, and click **Finish**.



Take a look inside the folder. You'll see several icons for the In-App Purchases – finally, you'll get to fill in those blank spots in the store! ☺

You'll also see a file called **productInfos.plist** that contains the information about the products that was previously hard-coded into the app. Open it to take a look:

| | | |
|----------------------|-------------------|--------------------------------|
| Root | Array | (4 items) |
| Item 0 | Dictionary | (6 items) |
| productIdentifier | String | com.razeware.hangman.hardwords |
| icon | String | icon_hardwords.png |
| consumable | Boolean | NO |
| consumableIdentifier | String | |
| consumableAmount | Number | 0 |
| bundleDir | String | HardWords |
| Item 1 | Dictionary | (6 items) |
| Item 2 | Dictionary | (6 items) |
| productIdentifier | String | com.razeware.hangman.tenhints |
| icon | String | icon_10hints.png |
| consumable | Boolean | YES |
| consumableIdentifier | String | com.razeware.hangman.hints |
| consumableAmount | Number | 10 |
| bundleDir | String | |
| Item 3 | Dictionary | (6 items) |

The property list is an array of dictionaries. Each dictionary represents information about a single In-App Purchase and contains the following entries:

- **productIdentifier**: The identifier of the In-App Purchase you set up in iTunes Connect.
- **icon**: The name of the icon to associate with the In-App Purchase in the store. Now that you have a method to associate additional information with a product identifier, you can tag extra-useful info like this to your products. You might want to create additional entries for screenshots to display in your store, a URL to a video demonstrating the In-App Purchase, an HTML page with more info, or whatever you may desire.
- **consumable**: A Boolean to indicate whether this is a consumable or non-consumable In-App Purchase type, because you will handle them differently.
- **consumableIdentifier**: To avoid hard-coding logic into your app, you're going to adopt a very simple way of handling consumables. Each consumable will be stored as an integer in `NSUserDefaults` (like you're already doing for hints in this project). This key indicates the name of the `NSUserDefaults` key for the identifier. In this example, both "10 Hints" and "100 Hints" use the same "com.razeware.hangman.hints" key. If you have other integer-based `NSUserDefaults` you want to modify with In-App Purchases, you can see that these would be extremely easy to add!
- **consumableAmount**: The amount by which to modify the previously mentioned `NSUserDefaults` key when the user makes the purchase. I bet you can imagine how easy adding a "50 hints" entry would be now!
- **bundleDir**: If the purchase is non-consumable and the content is embedded in the app bundle, this contains the subdirectory of the bundle where the content

resides. For example, the “Hard Words” pack is in the “HardWords” bundle subdirectory.

Before you move on, look through this file and replace all of the product identifiers with your own (matching what you set up in iTunes Connect). Also, make sure that whatever you put for the `consumableIdentifier` for “10 Hints” and “100 Hints” matches what **HMContentController.m** is using (in the `hints` and `setHints:` methods).

Reading product info

Now that you have this amazing file chock-full of useful info about your products, it’s time to add some code to read it!

Control-click on the **In-App Purchases** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **IAPPProductInfo**, make it a subclass of **NSObject**, and click **Next** and finally **Create**.

Open **IAPPProductInfo.h** and replace it with the following:

```
@interface IAPPProductInfo : NSObject

- (id)initFromDict:(NSDictionary *)dict;

@property (nonatomic, strong) NSString * productIdentifier;
@property (nonatomic, strong) NSString * icon;
@property (nonatomic, assign) BOOL consumable;
@property (nonatomic, strong) NSString * consumableIdentifier;
@property (nonatomic, assign) int consumableAmount;
@property (nonatomic, strong) NSString * bundleDir;

@end
```

This is a fairly simple class – you pass it a dictionary that contains information about the product, and it pulls out all of the information into properties.

Next, switch to **IAPPProductInfo.m** and replace it with the following:

```
#import "IAPPProductInfo.h"

@implementation IAPPProductInfo

- (id)initFromDict:(NSDictionary *)dict {
    if ((self = [super init])) {
        self.productIdentifier = dict[@"productIdentifier"];
        self.icon = dict[@"icon"];
        self.consumable = [dict[@"consumable"] boolValue];
```

```
    self.consumableIdentifier =
        dict[@"consumableIdentifier"];
    self.consumableAmount =
        [dict[@"consumableAmount"] intValue];
    self.bundleDir = dict[@"bundleDir"];
}
return self;
}

@end
```

This simply pulls out the information from the dictionary according to the format you set up in **productInfos.plist**.

Next, add a property to **IAPPProduct** to store this new class. Open **IAPPProduct.h** and pre-declare the class at the top of the file:

```
@class IAPPProductInfo;
```

Then add a new property:

```
@property (nonatomic, strong) IAPPProductInfo * info;
```

Then switch to **IAPPProduct.m** and add a new condition to `allowedToPurchase`. Now if you don't have information about a purchase, purchasing isn't allowed.

```
- (BOOL)allowedToPurchase {
    if (!self.availableForPurchase) return NO;

    if (self.purchaseInProgress) return NO;

    if (!self.info) return NO;

    if (self.purchase) return NO;

    return YES;
}
```

Why not? Remember that the **IAPPProductInfo** class now contains really important information you need to know in order to process a purchase correctly – like if the product is consumable or not, and if it is consumable, which `NSUserDefaults` to increment and by how much, etc. Without this info, you wouldn't know what to do!

You might notice that the new check for `info` was added above the existing check for `purchase`. While this isn't relevant now, it will become important when you next change things in this method.

Next, open **IAPHelper.h** and delete the `initWithProducts:` declaration. The subclass will no longer be passing a hard-coded list of products – now the `IAPHelper` will read the information from the property list itself.

Switch to **IAPHelper.m** and add this import to the top of the file:

```
#import "IAPPProductInfo.h"
```

Then replace `initWithProducts:` with the following:

```
- (id)init {
    if ((self = [super init])) {
        _products = [NSMutableDictionary dictionary];

        NSURL * productInfosURL = [[NSBundle mainBundle]
            URLForResource:@"productInfos" withExtension:@"plist"];
        NSArray * productInfosArray = [NSArray
            arrayWithContentsOfURL:productInfosURL];
        for (NSDictionary * productInfoDict in
            productInfosArray) {
            IAPPProductInfo * info = [[IAPPProductInfo alloc]
                initFromDict:productInfoDict];
            [self addInfo:info
                forProductIdentifier:info.productIdentifier];
        };

        [[SKPaymentQueue defaultQueue]
            addTransactionObserver:self];
    }
    return self;
}
```

This gets the path to `productInfos.plist` in the app bundle and converts it into an `NSArray` with the built-in property list deserialization. It then loops through the array, and creates an `IAPPProductInfo` for each.

It then calls a helper method to add the new product info object to the products dictionary, so that code gets added next:

```
- (IAPPProduct *)addProductForProductIdentifier:
(NSString *)productIdentifier {
    IAPPProduct * product = _products[productIdentifier];
    if (product == nil) {
        product = [[IAPPProduct alloc]
            initWithProductIdentifier:productIdentifier];
        _products[productIdentifier] = product;
    }
}
```

```
    return product;
}

- (void)addInfo:(IAPPProductInfo *)info
forProductIdentifier:(NSString *)productIdentifier {

    IAPPProduct * product = [self
        addProductForProductIdentifier:productIdentifier];
    product.info = info;

}
```

This simply adds the product to the dictionary if it's not already there, and sets the info on the new product.

Next, you have to make some changes to the way purchases are handled. Again, the basic idea is "out with the hard-coding!" and "in with the sweet new file-based method!"

Delete `provideContentForProductIdentifier:`, as you won't need it anymore. Then replace `provideContentForTransaction:productIdentifier:` with the following:

```
- (void)provideContentForTransaction:
(SKPaymentTransaction *)transaction
productIdentifier:(NSString *)productIdentifier {

    IAPPProduct * product = _products[productIdentifier];

    if (product.info.consumable) {
        [self
            purchaseConsumable:product.info.consumableIdentifier
            forProductIdentifier:productIdentifier
            amount:product.info.consumableAmount];
    } else {
        NSURL * bundleURL = [[NSBundle mainBundle].resourceURL
            URLByAppendingPathComponent:product.info.bundleDir];
        [self purchaseNonconsumableAtURL:bundleURL
            forProductIdentifier:productIdentifier];
    }

    [self notifyStatusForProductIdentifier:productIdentifier
        string:@"Purchase complete!"];

    product.purchaseInProgress = NO;
    [[SKPaymentQueue defaultQueue] finishTransaction:
        transaction];
```

```
}
```

This looks at the product info to see if it's a consumable purchase or not. If it's consumable, it calls a helper method that will increment the appropriate `NSUserDefaults`. If it's non-consumable, it calls a helper method that will eventually unlock the appropriate item. The rest is the same as before.

Next, add the helper methods called from the above method:

```
- (void)purchaseConsumable:(NSString *)consumableIdentifier  
forProductIdentifier:(NSString *)productIdentifier  
amount:(int)consumableAmount {  
    int previousAmount = [[NSUserDefaults standardUserDefaults]  
        integerForKey:consumableIdentifier];  
    int newAmount = previousAmount + consumableAmount;  
    [[NSUserDefaults standardUserDefaults] setInteger:newAmount  
        forKey:consumableIdentifier];  
    [[NSUserDefaults standardUserDefaults] synchronize];  
}  
  
- (void)provideContentWithURL:(NSURL *)URL {  
}  
  
- (void)purchaseNonconsumableAtURL:(NSURL *)nonLocalURL  
forProductIdentifier:(NSString *)productIdentifier {  
    [self provideContentWithURL:nonLocalURL];  
}
```

As you can see, the consumable case has become nice and generic, so it will be easy for you to reuse for any type of consumable item. As for the non-consumable case, it calls an empty method! This is because unlocking a directory of arbitrary contents is app-specific, so you'll have to leave that to the `HMIAPHelper`.

Speaking of which, `HMIAPHelper` is now greatly simplified. Open `HMIAPHelper.m` and delete the `init`, `unlockWordsForProductIdentifier:directory`, and `provideContentForProductIdentifier:` methods. Ahh – all of that horrible hard-coding is now gone. Time to party!



And in return, just add the following small method:

```
- (void)provideContentWithURL:(NSURL *)URL {
    [[HMContentController sharedInstance]
        unlockContentWithDirURL:URL];
}
```

The beauty of the way the code is organized is that to unlock content, you just have to call `unlockContentWithDirURL` on `HMContentController`. It will look inside the directory to see if it contains a `words.plist` (in which case it's treated as a words list) or a `themes.plist` (in which case it's treated as a theme). Pretty nice, eh?

Two more small changes and it will be time to test this out. Open **HMStoreListViewController.m** and add this import to the top of the file:

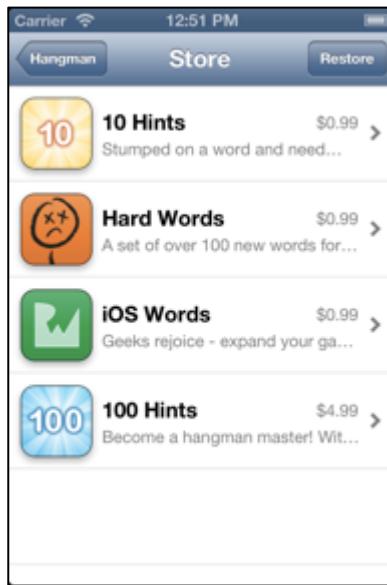
```
#import "IAPProductInfo.h"
```

Then inside `tableView:cellForRowAtIndexPath:`, add this line before the call to `return cell`:

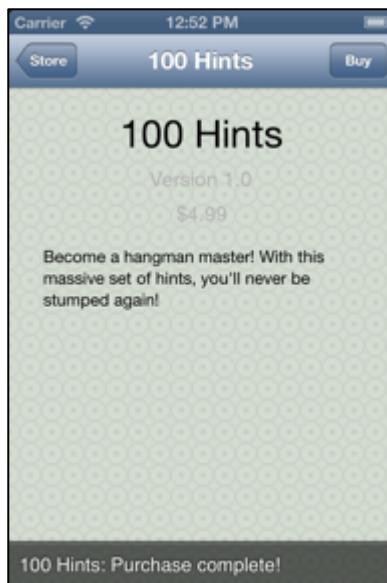
```
cell.iconImageView.image = [UIImage
    imageNamed:product.info.icon];
```

This sets the icon on the row to whatever has been configured in `productInfos.plist`.

And that's it! Build and run, and go to the store. The first thing you'll notice are the beautiful icons:



Then open a product and purchase it, and it will work just as before, but in a much more elegant file-based manner!



Recording purchases

So far so good, but if you play around with the app, you'll eventually notice that you broke something during this refactoring.

If you buy a non-consumable purchase, it is no longer marked as purchased after you buy it in the store. Even worse, when you shut down the app and start it back up again, the app won't remember that you purchased it, and you'll have to do it all over again. You don't want to be hunted down by a mob of angry users, so you'd better fix this. ☺

In the last chapter, you solved this by having a different `NSUserDefault` flag set for each non-consumable purchase to indicate whether it was purchased or not. Then on startup, you looked through each of these flags and unlocked the content if it was available.

This would still work, but it has some limitations that will come to haunt you later. If you only store a Boolean, you'll know that something has been purchased – but you won't necessarily know where the files have been stored, or what version of the purchase was downloaded. Instead of storing just one piece of information about a purchase, it's a lot more helpful to store a class of information that can be extended as necessary.

So to lay a good foundation that will make your life easier later, you're going to create a new class to store information about a purchase each time one is made. You will save the purchase information after each purchase, and on startup, you can look for previous purchases and restore them.

Control-click on the **In-App Purchases** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **IAPPProductPurchase**, make it a subclass of **NSObject**, and click **Next** and finally **Create**.

Open **IAPPProductPurchase.h** and replace it with the following:

```
@interface IAPPProductPurchase : NSObject <NSCoding>

- (id)initWithProductIdentifier:(NSString *)productIdentifier
    consumable:(BOOL)consumable
    timesPurchased:(int)timesPurchased
    libraryRelativePath:(NSString *)libraryRelativePath
    contentVersion:(NSString *)contentVersion;

@property (nonatomic, strong) NSString * productIdentifier;
@property (nonatomic, assign) BOOL consumable;
@property (nonatomic, assign) int timesPurchased;
@property (nonatomic, strong) NSString * libraryRelativePath;
@property (nonatomic, strong) NSString * contentVersion;

@end
```

This is similar to `IAPPProductInfo`, as it's mainly storing a lot of information about the purchase. A few things to point out:

- This implements `NSCoding`, which will allow you to easily serialize/deserialize the class to/from `NSData` (and hence a file).
- No matter where the directory of contents originally comes from (the app bundle directory or somewhere else), you are going to copy the contents into the app's

library directory. This way, everything is in one known place (and it resolves some technical challenges involving changing bundle and app URLs).

Next switch to **IAPPProductPurchase.m** and replace the contents with the following:

```
#import "IAPPProductPurchase.h"

static NSString *const kProductIdentifierKey =
    @"ProductIdentifier";
static NSString *const kConsumableKey = @"Consumable";
static NSString *const kTimesPurchasedKey = @"TimesPurchased";
static NSString *const kLibraryRelativePathKey =
    @"LibraryRelativePath";
static NSString *const kContentVersionKey = @"ContentVersion";

@implementation IAPPProductPurchase

- (id)initWithProductIdentifier:(NSString *)productIdentifier
    consumable:(BOOL)consumable
    timesPurchased:(int)timesPurchased
    libraryRelativePath:(NSString *)libraryRelativePath
    contentVersion:(NSString *)contentVersion {
    if ((self = [super init])) {
        self.productIdentifier = productIdentifier;
        self.consumable = consumable;
        self.timesPurchased = timesPurchased;
        self.libraryRelativePath = libraryRelativePath;
        self.contentVersion = contentVersion;
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)aDecoder {
    NSString * productIdentifier = [aDecoder
        decodeObjectForKey:kProductIdentifierKey];
    BOOL consumable = [aDecoder
        decodeBoolForKey:kConsumableKey];
    int timesPurchased = [aDecoder
        decodeIntForKey:kTimesPurchasedKey];
    NSString * libraryRelativePath = [aDecoder
        decodeObjectForKey:kLibraryRelativePathKey];
    NSString * contentVersion = [aDecoder
        decodeObjectForKey:kContentVersionKey];
    return [self initWithProductIdentifier:productIdentifier
        consumable:consumable
        timesPurchased:timesPurchased
        libraryRelativePath:libraryRelativePath
        contentVersion:contentVersion];
}
```

```

        libraryRelativePath:libraryRelativePath
        contentVersion:contentVersion];
}

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:self.productIdentifier
        forKey:kProductIdentifierKey];
    [aCoder encodeBool:self.consumable
        forKey:kConsumableKey];
    [aCoder encodeInt:self.timesPurchased
        forKey:kTimesPurchasedKey];
    [aCoder encodeObject:self.libraryRelativePath
        forKey:kLibraryRelativePathKey];
    [aCoder encodeObject:self.contentVersion
        forKey:kContentVersionKey];
}

@end

```

This contains a basic initializer, and the implementation of the `NSCoding` protocol to implement serialization and deserialization.

Next, move on to `IAPPProduct` and remove the old Boolean flag you were using to keep track of whether a product had been purchased, and replace it with this new class. Open **IAPPProduct.h** and add the following class declaration to the top of the file:

```
@class IAPPProductPurchase;
```

Then replace the declaration of the `purchase` property with the following:

```
@property (nonatomic, strong) IAPPProductPurchase * purchase;
```

Switch to **IAPPProduct.m**. Add the following import to the top of the file:

```
#import "IAPPProductInfo.h"
```

Then modify the check for `purchase` in `allowedToPurchase` as follows:

```

- (BOOL)allowedToPurchase {
    if (!self.availableForPurchase) return NO;

    if (self.purchaseInProgress) return NO;

    if (!self.info) return NO;

    if (!self.info.consumable && self.purchase) {

```

```
        return NO;
    }

    return YES;
}
```

Now you are tracking purchase information on consumable as well as non-consumable purchases, so you have to make sure the product isn't consumable before disallowing a purchase due to a prior purchase.

Do you see why the check for `info` was placed before the final check? You need to make sure the info is there before trying to check its `consumable` property. The way the code is structured now, if `info` is `nil`, you never get to the next check.

Now for the bulk of the changes. Open **IAPHelper.m** and add the following to the top of the file:

```
#import "IAPPurchase.h"

static NSString *const IAPHelperPurchasesPlist =
@"purchases.plist";
```

This imports the new class you just made and keeps track of the filename that will store the purchase info.

Next, add a bunch of helper methods:

```
// 1
- (NSString *)libraryPath {
    NSArray * libraryPaths =
        NSSearchPathForDirectoriesInDomains(NSLibraryDirectory,
        NSUserDomainMask, YES);
    return libraryPaths[0];
}

// 2
- (NSString *)purchasesPath {
    return [[self libraryPath]
        stringByAppendingPathComponent:IAPHelperPurchasesPlist];
}

// 3
- (void)addPurchase:(IAPPurchase *)purchase
forProductIdentifier:(NSString *)productIdentifier {

    IAPProduct * product = [self
        addProductForProductIdentifier:productIdentifier];
```

```
    product.purchase = purchase;

}

// 4
- (IAPPProductPurchase *)purchaseForProductIdentifier:
(NSStrong * )productIdentifier {
    IAPPProduct * product = _products[productIdentifier];
    if (!product) return nil;

    return product.purchase;
}
```

Let's go over each method:

1. A helper method to return the path of the app's library directory. Note that you are using the library directory rather than the documents directory by choice. The documents directory is meant for documents that you want the user to be able to modify (if you're using iTunes File Sharing or iCloud), and the library directory is meant for private app files. You want to keep these files private, so you choose the library directory.
2. A helper method to return the full path where information about the app purchases is stored.
3. A helper method to get an `IAPPProduct` for a product identifier (using the helper method to create it if it doesn't exist already), and set the `IAPPProductPurchase` on it appropriately.
4. A helper method to return the information about a previous purchase for a product identifier – or nil if no such purchase (or product) exists.

You'll be using these helper methods as you continue along the path to make your life easier. Next add the routines to load and save the purchase info:

```
- (void)loadPurchases {

    // 1
    NSString * purchasesPath = [self purchasesPath];
    NSArray * purchasesArray = [NSKeyedUnarchiver
        unarchiveObjectWithFile:purchasesPath];
    for (IAPPProductPurchase * purchase in purchasesArray) {
        // 2
        if (purchase.libraryRelativePath) {
            NSString * localPath = [[self libraryPath]
                stringByAppendingPathComponent:
                purchase.libraryRelativePath];
            NSURL * localURL = [NSURL fileURLWithPath:localPath
                isDirectory:YES];
        }
    }
}
```

```
        [self provideContentWithURL:localURL];
    }
    // 3
    [self addPurchase:purchase
        forProductIdentifier:purchase.productIdentifier];
    NSLog(@"Loaded purchase for %@ (%@)",
        purchase.productIdentifier, purchase.contentVersion);
}
}

- (void)savePurchases {

    // 1
    NSString * purchasesPath = [self purchasesPath];
    NSMutableArray * purchasesArray = [NSMutableArray array];
    for (IAPPProduct * product in _products.allValues) {
        if (product.purchase) {
            [purchasesArray addObject:product.purchase];
        }
    }
    // 2
    BOOL success = [NSKeyedArchiver
        archiveRootObject:purchasesArray toFile:purchasesPath];
    if (!success) {
        NSLog(@"Failed to save purchases to %@", purchasesPath);
    }
}
}
```

The `loadPurchases` method will be called on app startup. It does the following:

1. Unarchives the saved `purchases.plist` file, getting a list of `IAPPProductPurchases`.
2. It then gets the full path to each purchase by appending the saved `libraryRelativePath` to the library directory, and calls `provideContentWithURL:` to actually unlock the content.
3. It then uses the helper method to add an `IAPPProduct` for this to the `products` array, or update it if it's already there. Note this is an important design consideration, because it takes care of the case where a user purchased a product that is no longer available.

The `savePurchases` method will be called whenever the user makes a new purchase. It does the following:

1. Loops through all of the products. Any that have purchase information, it adds to an array of `IAPPProductPurchases`.
2. Saves the array out to `purchases.plist`.

Now that you have all of the pieces in place, time to use it! First, add the call to `loadPurchases` in `init`, right after creating the dictionary of products:

```
- (id)init {
    if ((self = [super init])) {

        _products = [NSMutableDictionary dictionary];
        [self loadPurchases];
        // ...
    }
    return self;
}
```

Then add this code to record the purchase to the bottom of `purchaseConsumable:forProductIdentifier:amount:`

```
IAPPurchase * previousPurchase = [self
    purchaseForProductIdentifier:productIdentifier];
if (previousPurchase) {
    previousPurchase.timesPurchased++;
} else {
    IAPPurchase * purchase = [[IAPPurchase alloc]
        initWithProductIdentifier:productIdentifier consumable:YES
        timesPurchased:1 libraryRelativePath:@"""
        contentVersion:@""];
    [self addPurchase:purchase
        forProductIdentifier:productIdentifier];
}
[self savePurchases];
```

Basically, if there's a previous purchase it bumps the number of times purchased up by one. Otherwise it creates a new purchase and saves it out.

One final step in this file – but it's a big one. Take a deep breath, grab your favorite caffeinated beverage, turn on some groovy tunes, and get ready to code like there's no tomorrow! Or just make wise use of copy/paste. ☺

Replace `purchaseNonconsumableAtURL:forProductIdentifier:` with the following:

```
- (void)purchaseNonconsumableAtURL:(NSURL *)nonLocalURL
    forProductIdentifier:(NSString *)productIdentifier {

    NSError * error = nil;
    BOOL success = FALSE;
    BOOL exists = FALSE;
    BOOL isDirectory = FALSE;
```

```
// 1
NSString * libraryRelativePath =
    nonLocalURL.lastPathComponent;
NSString * localPath = [[self libraryPath]
    stringByAppendingPathComponent:libraryRelativePath];
NSURL * localURL = [NSURL fileURLWithPath:localPath
    isDirectory:YES];
exists = [[NSFileManager defaultManager]
    fileExistsAtPath:localPath isDirectory:&isDirectory];

// 2
if (exists) {
    BOOL success = [[NSFileManager defaultManager]
        removeItemAtURL:localURL error:&error];
    if (!success) {
        NSLog(@"Couldn't delete directory at %@: %@", localURL,
            error.localizedDescription);
    }
}

// 3
NSLog(@"Copying directory from %@ to %@", nonLocalURL,
    localURL);
success = [[NSFileManager defaultManager]
    copyItemAtURL:nonLocalURL toURL:localURL error:&error];
if (!success) {
    NSLog(@"Failed to copy directory: %@", error.localizedDescription);
    [self notifyStatusForProductIdentifier:productIdentifier
        string:@"Copying failed."];
    return;
}

NSString * contentVersion = @"";
// 4
[self provideContentWithURL:localURL];

// 5
IAPPurchase * previousPurchase = [self
    purchaseForProductIdentifier:productIdentifier];
if (previousPurchase) {
    previousPurchase.timesPurchased++;

// 6
```

```
NSString * oldPath = [[self libraryPath]
    stringByAppendingPathComponent:
    previousPurchase.libraryRelativePath];
success = [[NSFileManager defaultManager]
    removeItemAtPath:oldPath error:&error];
if (!success) {
    NSLog(@"Could not remove old purchase at %@", oldPath);
} else {
    NSLog(@"Removed old purchase at %@", oldPath);
}

// 7
previousPurchase.libraryRelativePath =
    libraryRelativePath;
previousPurchase.contentVersion = contentVersion;
} else {
    IAPPProductPurchase * purchase =
        [[IAPPProductPurchase alloc]
            initWithProductIdentifier:productIdentifier
            consumable:NO timesPurchased:1
            libraryRelativePath:libraryRelativePath
            contentVersion:contentVersion];
    [self addPurchase:purchase
        forProductIdentifier:productIdentifier];
}

[self notifyStatusForProductIdentifier:productIdentifier
    string:@"Purchase complete!"];

// 8
[self savePurchases];

}
```

Wow, a lot of code here! It's important to understand it though, so let's go over this section-by-section.

1. The URL that is passed in (`nonLocalURL`) is probably not within the library directory (right now it's a directory in the application bundle). So the first task is to get a directory similar to the one passed in, but inside the library directory. To do this, it strips off the last part of the URL (i.e. the directory name) and appends it to the path of the library directory.
2. If that directory already exists, it deletes it.
3. It then copies the directory from its current location (right now, inside the application bundle) to the library directory.

4. It then unlocks the content as usual.
5. It looks to see if there's a previous purchase for this product identifier. If not, it records a new one.
6. If there is a purchase, it finds the place where the old version of this product identifier was saved and removes it. Note that this step is still necessary, despite step #2, because the old version could have been stored in a different directory name.
7. It then updates the stored purchase information with the new directory name, content version, and times purchased.
8. It saves the new purchase information to disk.

One final change. Open **HMStoreListViewController.m** and add the following import to the top of the file:

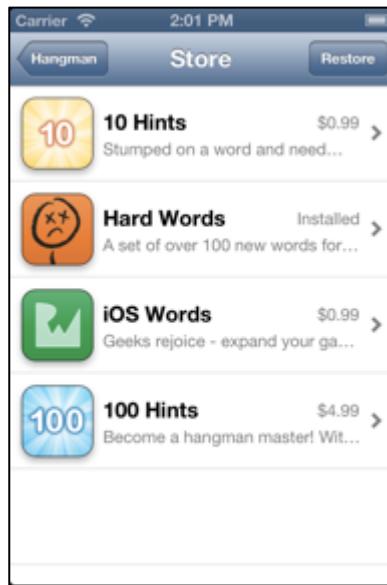
```
#import "IAPPProductPurchase.h"
```

And in `tableView:cellForRowAtIndexPath:`, modify the lines that set the `cell.priceLabel.text` as follows:

```
if (!product.purchase.consumable && product.purchase) {  
    cell.priceLabel.text = @"Installed";  
} else {  
    cell.priceLabel.text = [_priceFormatter  
        stringFromNumber:product.skProduct.price];  
}
```

If a product has a purchase recorded, this makes sure it's non-consumable before marking it as installed.

Finally, you're done! Build and run, and now when you purchase a product, it will be marked as installed – even if you restart the app!



Finalizing the move

Now that you have the groundwork in place and are properly reading your product information from a file instead of hard-coding it, you can move the file off to your own web server!

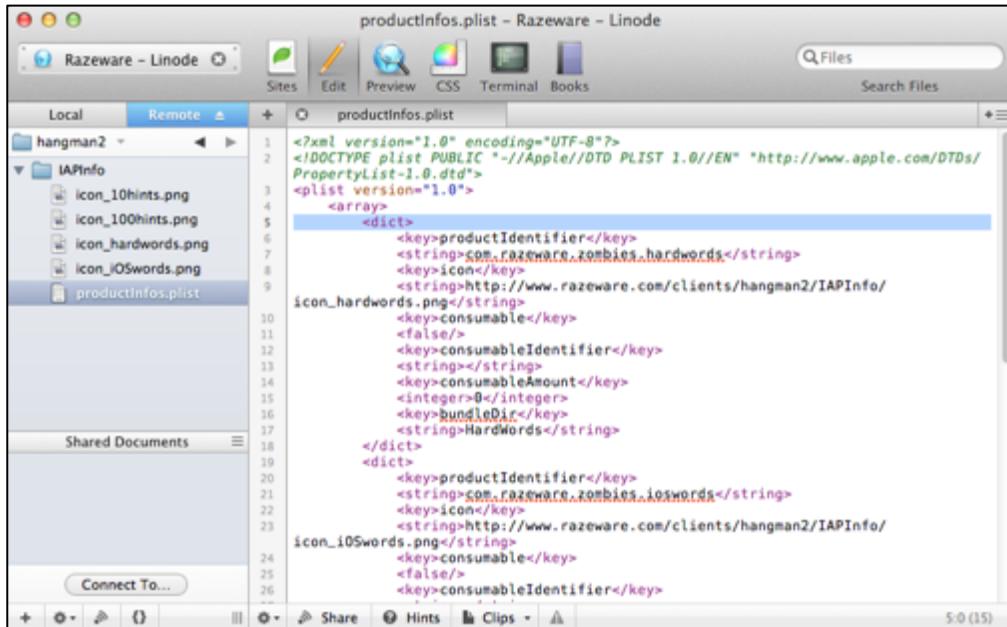
Note: To go through this section, you will need access to a web server. If you don't have your own web server to test with, you can set up your own test server on your Mac. Setting up a web server is beyond the scope of this chapter, but for more information you can check out the following project, which makes it pretty easy to run a web server with MySQL and PHP (which you'll need later) on your Mac:

<http://www.apachefriends.org/en/xampp-macosx.html>

The first step is to upload your **IAPInfo** directory (containing `productInfos.plist` and the icons) to your web server.

You will also have to update **productInfos.plist** so that the `icon` fields contain full URLs to your icons (instead of local URLs like they do now). You can do this before or after uploading the file – whichever is easier for you.

I usually prefer to edit it directly on the web server with my editor of choice (Coda), as you can see here:



Once you've uploaded the files, go back to Xcode and delete the **IAPInfo** folder from your project. You might want to choose "Remove References" instead of "Move to Trash" so the old files stay around in case you need them later.

Now it's time for the code changes. Open **IAPHelper.m** and add the following to the top of the file:

```

#import "AFNetworking.h"
#import "AFHTTPClient.h"
#import "AFHTTPRequestOperation.h"

static NSString *const IAPServerDataURL =
    @"https://www.razeware.com";
static NSString *const IAPServerProductsURL =
    @"/clients/hangman2/IAPInfo/productInfos.plist";

```

Here you import some header files for AFNetworking, a popular and easy-to-use networking library that you're going to use to simplify your networking code.

You then declare the URLs for where your productInfos.plist is found. Note it's split into two parts: a base URL, and a sub URL. Replace those with wherever you stored your productInfos.plist.

Also, add a new instance variable to the `@implementation` to keep track of whether the products have been loaded:

```

@implementation IAPHelper {
    SKProductsRequest * _productsRequest;
    RequestProductsCompletionHandler _completionHandler;
    BOOL _productsLoaded;
}

```

```
}
```

Next, add this new method to the file, which is the main workhorse:

```
- (void)loadProductsWithCompletionHandler:(void (^)(  
    BOOL success, NSError * error))completionHandler {  
  
    // 1  
    for (IAPPProduct * product in _products.allValues) {  
        product.info = nil;  
        product.availableForPurchase = NO;  
    }  
  
    // 2  
    NSURL * baseUrl = [NSURL URLWithString:IAPServerBaseUrl];  
    AFHTTPClient * httpClient = [[AFHTTPClient alloc]  
        initWithBaseUrl:baseUrl];  
    NSURL * url = [NSURL URLWithString:IAPServerProductsURL  
        relativeToURL:baseUrl];  
  
    // 3  
    NSMutableURLRequest * request = [NSMutableURLRequest  
        requestWithURL:url  
        cachePolicy:NSURLRequestReloadIgnoringCacheData  
        timeoutInterval:60];  
  
    // 4  
    AFHTTPRequestOperation *operation = [httpClient  
        HTTPRequestOperationWithRequest:request  
        success:^(AFHTTPRequestOperation *operation, id  
        responseObject) {  
  
        // 5  
        NSData * productInfosData = [operation responseData];  
        NSError * error = nil;  
        NSArray * productInfosArray =  
            [NSPropertyListSerialization  
                propertyListWithData:productInfosData  
                options:NSPropertyListImmutable format:NULL  
                error:&error];  
        if (productInfosArray == nil) {  
            completionHandler(FALSE, error);  
        } else {  
  
            // 6  
            for (NSDictionary * productInfoDict in
```

```
productInfosArray) {
    IAPProductInfo * info = [[IAPProductInfo alloc]
        initFromDict:productInfoDict];
    [self addInfo:info
        forProductIdentifier:info.productIdentifier];
}

// 7
if (!productsLoaded) {
    _productsLoaded = YES;
    [[SKPaymentQueue defaultQueue]
        addTransactionObserver:self];
}

// 8
completionHandler(TRUE, nil);
}

} failure:^(AFHTTPRequestOperation *operation,
NSError *error) {
    completionHandler(FALSE, error);
}];
[httpClient enqueueHTTPRequestOperation:operation];

}
```

There's a lot to discuss here, so let's go over it section-by-section.

1. Loops through all of the products and “resets” them by clearing out any previous info and marking them as not available for purchase.
2. Determines the full URL to productInfos.plist by combining `IAPServerBaseURL` and `IAPServerProductsURL`.
3. Sets the URL request to ignore cached data. This is important, because the default behavior is to load a cached version of a previous request if one is available. But while you are debugging, you don't want a cached version – you always want the most up-to-date version from the web server. This forces the library to make a new request to the web server each time. In production, you may wish to turn this off and leave the caching on for decreased load on your web server.
4. Uses AFNetworking to retrieve the file. The nice thing about AFNetworking is that it's as simple as firing this request, and it will give you the results back in completion blocks (or a failure in a failure block).
5. AFNetworking gives you the data from the web server in the `responseData` property on the `AFHTTPRequestOperation`. Here you take that and deserialize it into an array, using property list serialization.

6. Same as before, here you loop through the array and store the information.
7. If this is the first time the list of products has been loaded, it starts the transaction observer. Note that the transaction observer won't be loaded right away anymore, it has to wait until the first time you get the product info. This is because if you hear about a transaction but don't have the info yet, it does you no good, because you need the info in order to know how to resolve the transaction. (For example, which `NSUserDefaults` to update and how many times, etc.)
8. Finally, calls your passed-in completion handler block, letting the caller know it's done.

Just a bit of wrap-up left. Replace the `init` method with the following much-simplified version:

```
- (id) init {
    if ((self = [super init])) {
        _products = [NSMutableDictionary dictionary];
        [self loadPurchases];
        [self loadProductsWithCompletionHandler:^(BOOL success,
            NSError *error) {
        }];
    }
    return self;
}
```

This removes the code that loaded the old built-in `productInfos.plist` and registered the transaction observer immediately, since that is now done in `loadProductsWithCompletionHandler:`.

The final change in this file is to modify `requestProductsWithCompletionHandler:` so it looks like the following:

```
- (void) requestProductsWithCompletionHandler:
    (RequestProductsCompletionHandler)completionHandler {

    _completionHandler = [completionHandler copy];

    [self loadProductsWithCompletionHandler:^(BOOL success,
        NSError *error) {

        NSMutableSet * productIdentifiers = [NSMutableSet
            setWithCapacity:_products.count];
        for (IAPPProduct * product in _products.allValues) {
            if (product.info) {
                product.availableForPurchase = NO;
                [productIdentifiers
                    addObject:product.productIdentifier];
            }
        }
        completionHandler(success, nil);
    }];
}
```

```
        }
    }

    _productsRequest = [[SKProductsRequest alloc]
        initWithProductIdentifiers:productIdentifiers];
    _productsRequest.delegate = self;
    [_productsRequest start];

}];

}
```

There are two main changes here:

1. All of the code is now put inside a call to `loadProductsWithCompletionHandler`. This way, before refreshing the list of available In-App Purchases from Apple, you first ask your server what it thinks should be available.
2. A slight modification is made so that you only ask Apple for info about the products you have info about. This way, if you take an item out from your `productInfos.plist`, it will no longer show up in your store.

The final change you need to make before testing is to modify the store list view controller so it can display icons that are on the web, instead of being bundled within the app. Luckily, AFNetworking makes this extremely easy as well.

Open **HMStoreListViewController.m** and add this to the top of the file:

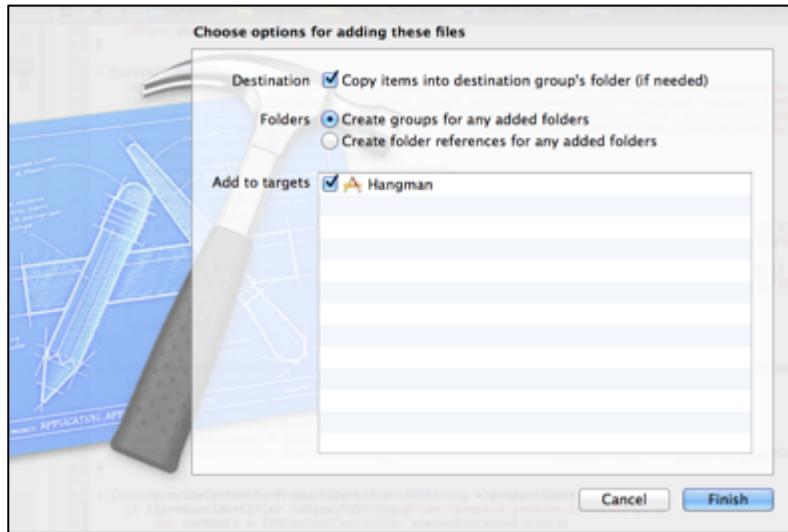
```
#import "UIImageView+AFNetworking.h"
```

This imports a class extension to `UIImageView`, provided by AFNetworking, that makes loading an image from a URL into an image view very easy. Try it out by replacing the line that sets the `cell.iconImageView.image` in `tableView:cellForRowAtIndexPath:` with the following:

```
[cell.iconImageView setImageWithURL:
    [NSURL URLWithString:product.info.icon]
placeholderImage:
    [UIImage imageNamed:@"icon_placeholder.png"]];
```

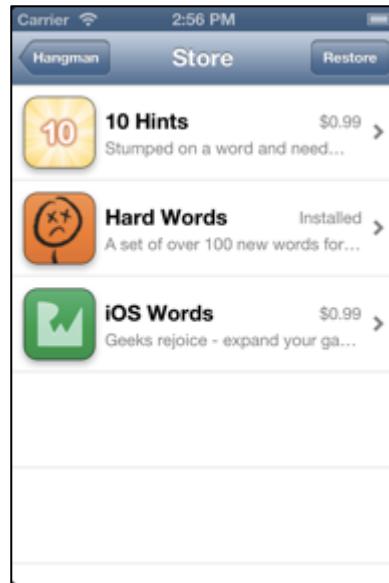
Pretty easy, eh? It will automatically load the image at the specified URL in the background. And while it's loading, it will set the image to a placeholder image.

Of course, for this to work you need a placeholder image. In the resources for this chapter, you will find a folder called **PlaceholderIcon**. Inside is **icon_placeholder.png**. Drag it into your project's **Resources** folder, make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, the **Hangman** target is checked, and click **Finish**.

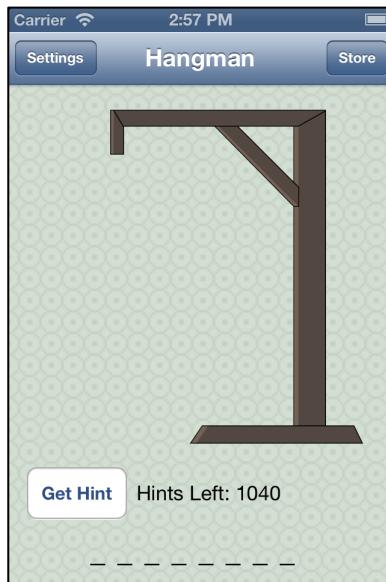


And that's it – build and run your app, and go to the store. After a moment, your products will load as usual – and if you watch very carefully, you might notice the placeholder icons flash before the full icons are loaded.

It might be fun to prove to yourself that you actually have full control over what's in there now. Open your productInfos.plist file and remove the 100 Hints entry. Then, without shutting down your app, pull to refresh the table view. The 100 Hints item immediately disappears!



For even more fun, put the entry back, but change the number of hints that "100 Hints" gives you to 1000. Refresh your table view so it appears again, and then purchase your product. Holy hints, Batman!



Reaping the benefits

To prove to you that all your hard work was worth it, you're going to add a new In-App Purchase to your app and see how easy it's become. And the best part is, you'll do it all dynamically, without updating any code!

In fact, as you work on this section you can just leave Hangman running on your device or simulator. When you're done, you can refresh the table view, your new product will appear, and you can purchase it. ☺

Open [iTunes Connect](#) and click **Manage Your Applications**. Find your Hangman app in the list, open it, and click **Manage In-App Purchases**. Click **Create New**, choose **Consumable\Select**, and enter the following information:

The screenshot shows the 'In-App Purchase Summary' screen in iTunes Connect. The product details are as follows:

| | |
|-----------------|---------------------------------|
| Reference Name: | 50 Hints |
| Product ID: | com.razeware.hangman.fiftyhints |
| Type: | Consumable |
| Apple ID: | 548665465 |

Pricing and Availability

The pricing and availability details for this In-App Purchase are shown below.

Cleared for Sale: Yes

| Price Tier | Price Effective Date | Price End Date |
|------------|----------------------|----------------|
| Tier 3 | Existing | None |

Don't forget to modify your Product ID so that it's based on your own Bundle Identifier – yours might look like com.yourdomain.hangman.fiftyhints, for example.

Scroll down to the **In-App Purchase Details\Language** section and click **Add Language**. Fill out the popup as follows:



Click **Save**, and then scroll down to the bottom of the page and click **Save** again. Done with iTunes Connect!

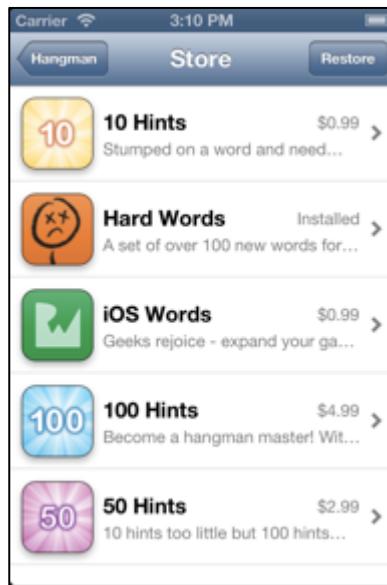
Now edit **productInfos.plist** on your web server and add a new dictionary to the array, similar to the following:

```
<dict>
    <key>productIdentifier</key>
    <string>com.razeware.hangman.fiftyhints</string>
    <key>icon</key>
    <string>http://www.razeware.com/clients/hangman2/IAPInfo/icon_50hints.png</string>
    <key>consumable</key>
    <true/>
    <key>consumableIdentifier</key>
    <string>com.razeware.hangman.hints</string>
    <key>consumableAmount</key>
    <integer>50</integer>
    <key>bundleDir</key>
    <string></string>
</dict>
```

Of course, remember to modify the `productIdentifier` and `consumableIdentifier` based on what you set up for your project, and to modify the URL for the icon.

Finally, inside the resources for this project you will find a folder called **50HintsIcon**. Inside is a file called **icon_50hints.png**. Upload this to your web server next to the other icons.

That's it! Go back to your app (which hopefully is still running from before) and refresh your table view. Voila – your new In-App Purchase appears, with no update required!



As you can see, through this refactoring you have given yourself a ton of power and flexibility. You can add new products very easily without having to wait for app review cycles or modifying code, which allows you to iterate faster and tweak your purchase options and prices based on user metrics and feedback!

Hosted Content

Finally, you are ready to try out one of the cool new features in iOS 6 with In-App Purchases: Hosted Content!

Hosted Content is a way for you to host your In-App Purchases as downloadable content on Apple's servers. It's much easier than setting up your own infrastructure, and best of all, it's free!

To mark an In-App Purchase as using Hosted Content, you simply check the box when registering it in iTunes Connect:

Hosting Content with Apple

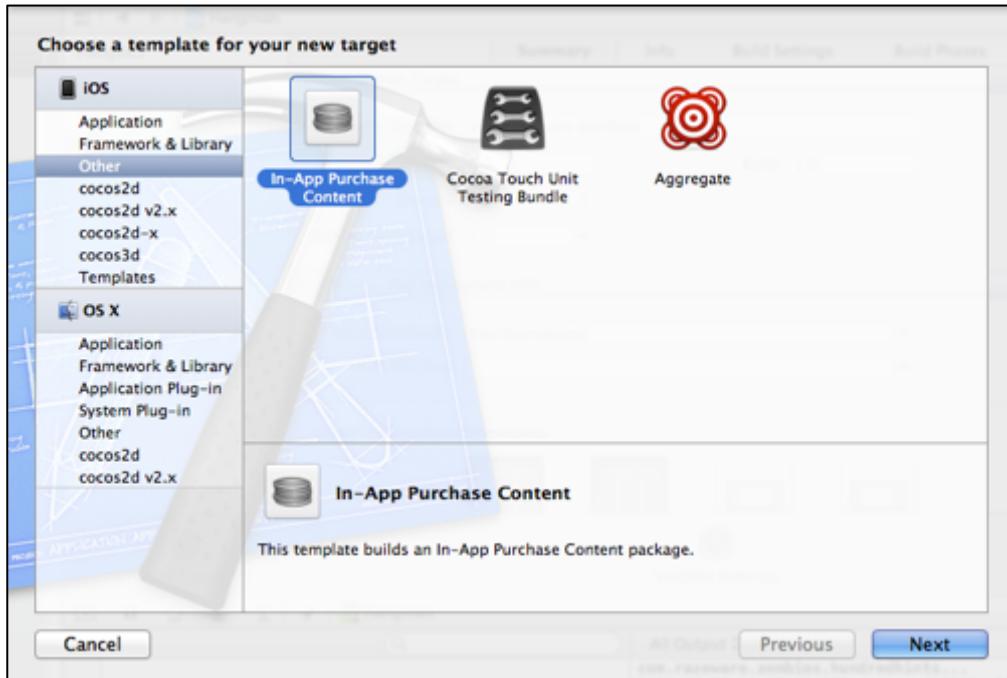
Select if you want Apple to host your In-App Purchase content. If you select yes, you must upload your content to Apple before sending the In-App Purchase for review.

Hosting Content with Apple Yes No

You then need to upload your content to Apple in a particular format. Basically, it's a zip file with two important items inside:

- A file called **ContentInfo.plist** with some metadata about your In-App Purchase
 - for now, just the version and product identifier.
- A directory called **Contents** with whatever your In-App Purchase needs – property lists, images, movies, whatever.

Even though this zip file would be pretty simple to make, Apple has made it even easier for you by integrating the process into Xcode. You simply add a new In-App Purchase target into your app:



Then you can drag whatever files you need into the target and use Xcode's Archive capability to archive it to the Organizer. From there, you upload it directly to iTunes Connect.

Once you've marked your In-App Purchase as using Hosted Content and uploaded it to iTunes Connect, you just need to make sure your code can process the downloads. There are three main steps:

- 1. Check for downloads (and start them if they exist).** When a transaction completes, you should check the `SKPaymentTransaction`'s new downloads property to see if there are any downloads for the transaction. If there are, you should start them up. But don't finish the transaction quite yet!
- 2. Receive progress updates.** In iOS 6, `SKPaymentTransactionObserver` has a new `paymentQueue:updatedDownloads` callback that will notify you as downloads progress. You can use this to display nice updates in your GUI.
- 3. Process and unlock the content.** When the download completes, you should move it into a safe directory (by default, the downloads will be stored in the `Caches` directory, and hence eventually deleted unless you move them somewhere safe), and then unlock the content as usual.

In this section, you are going to try out Hosted Content for yourself by adding a cool new feature to Hangman – downloadable themes!

Note: If you skipped over the previous section and are just starting here, you can find the project where you left it off in the **HangmanCh10Refactored** folder in the resources for this chapter.

Note that it is currently set up to use a) my Bundle Identifier and b) In-App Purchases set up by me, and c) information about the products hosted on my own web server.

To have full control over the app, you should create a new app for yourself in iTunes Connect with your own In-App Purchases, and modify the product identifiers/Bundle ID in the app appropriately.

Also, you should upload the **Hangman\IAPInfo** folder to your own web server and modify the **productInfos.plist** inside to match the In-App Purchase product identifiers you set up.

If you are confused about how to do this, refer back to the previous sections.

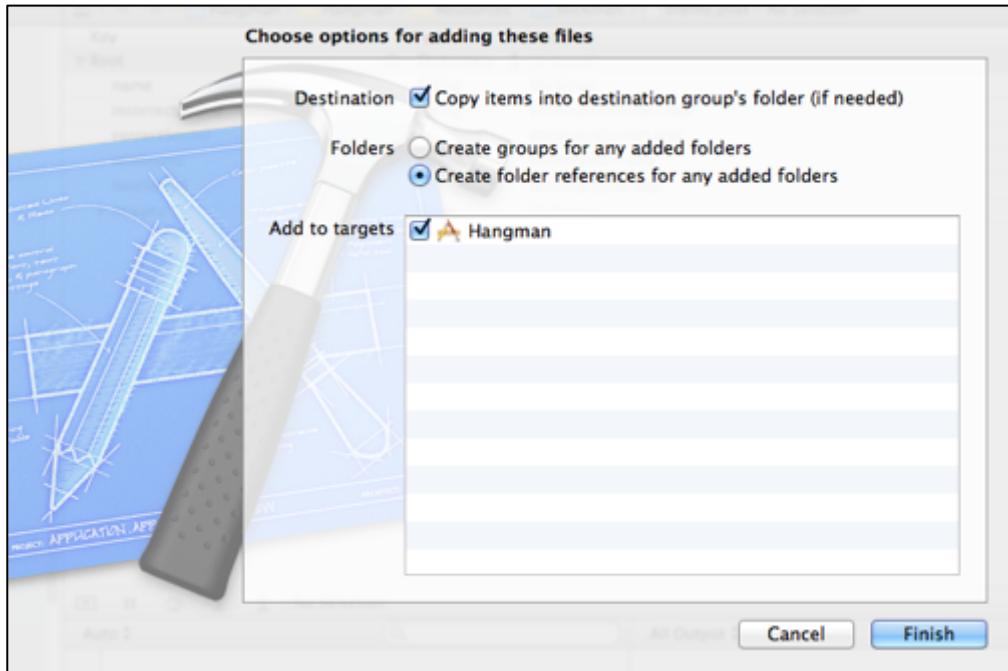
Testing the themes

Since you have wisely designed the theme system in Hangman to be file-based instead of code-based, adding a new theme is as simple as creating a theme.plist and the associated images and sound effects.

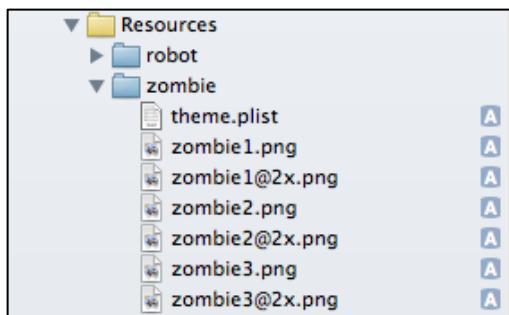
So you ask your favorite artist and musician to make some new themes for Hangman for you, and they come back with two cool themes: a Robot and a Zombie theme. After all, as you can tell from the App Store, you can't lose if you have robots and zombies in your app! ☺

As always, the first step before making In-App Purchases is to test out your content and make sure it actually works, so let's try them out.

In the resources for this chapter, you will find a folder called **Themes**. Drag the **robot** and **zombie** folders inside into the **Resources** folder of your Xcode project. Make sure that **Copy items into destination group's folder (if needed)** is checked, **Create folder references for any added folders** is selected, the Hangman target is checked, and click Finish.



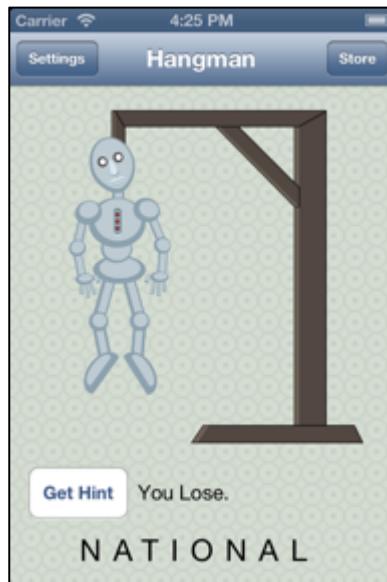
At this point, your Project Navigator should look like this (note the blue folders):



Since the code has been designed to load themes from a directory, testing this is simple. Open **HMContentController.m** and add the following lines to the `init` method, after the call to unlock the Stickman theme:

```
[self unlockThemeWithURL:[resourceURL
    URLByAppendingPathComponent:@"robot"]];
[self unlockThemeWithURL:[resourceURL
    URLByAppendingPathComponent:@"zombie"]];
```

That's it! Build and run the app, choose a new theme, and try it out!



I wonder if Google would be upset about you hanging Androids? 😊

Registering the In-App Purchases

Now that you're confident that the new In-App Purchases are working, it's time to register them on iTunes Connect. You've done this several times now, so you should be an old hand at this.

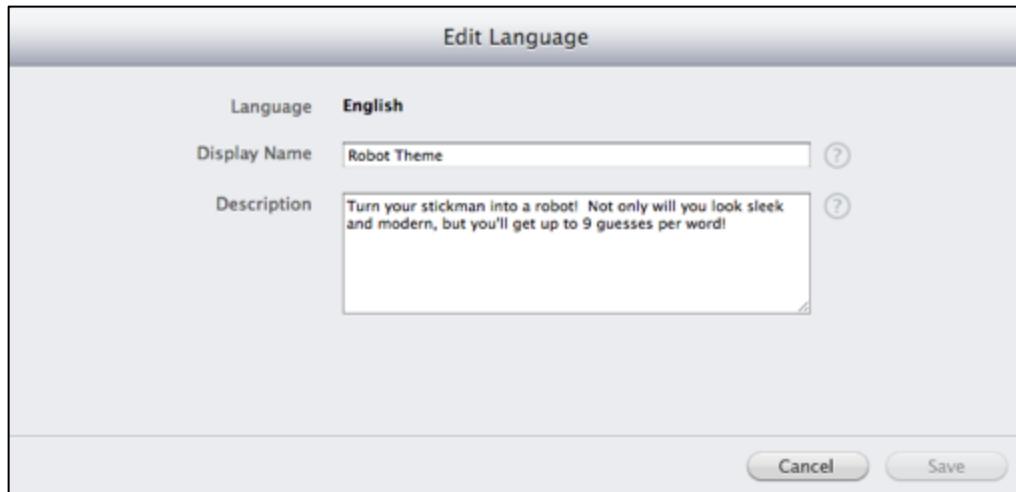
Log on to [iTunes Connect](#) and click **Manage Your Applications**. Find your Hangman app in the list, open it, and click **Manage In-App Purchases**. Click **Create New**, choose **Non-Consumable\Select**, and enter the following information:

The screenshot shows the 'In-App Purchase Summary' page for a product named 'Robot Theme'. The product ID is 'com.razeware.hangman.robot'. The type is listed as 'Non-Consumable'. The Apple ID is '544840291'. Under the 'Pricing and Availability' section, it says 'Cleared for Sale Yes'. A table below shows the price tier configuration: Tier 1 has an effective date of 'Existing' and an end date of 'None'.

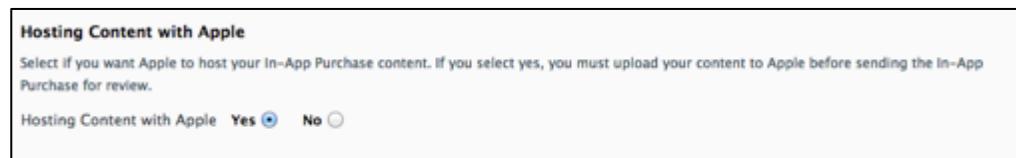
| Price Tier | Price Effective Date | Price End Date |
|------------|----------------------|----------------|
| Tier 1 | Existing | None |

Don't forget to modify your Product ID so that it's based on your own Bundle Identifier – yours might look like com.yourdomain.hangman.robot, for example.

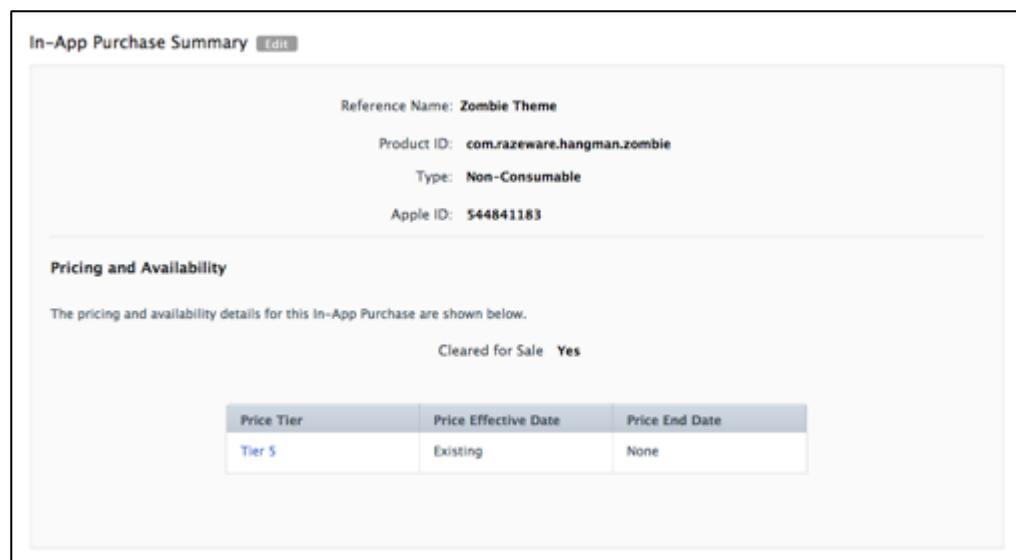
Scroll down to the **In-App Purchase Details\Language** section and click **Add Language**. Fill out the popup as follows:



And most importantly, scroll down and mark this In-App Purchase as having Hosted Content by selecting the radio button:



Then scroll down to the bottom of the page and click **Save**. Now repeat for the Zombie Theme – here is the relevant info for both sections.





Again, don't forget to mark it has having Hosted Content, and to modify your Product ID appropriately!

At this point, your list of In-App Purchases should look like this:

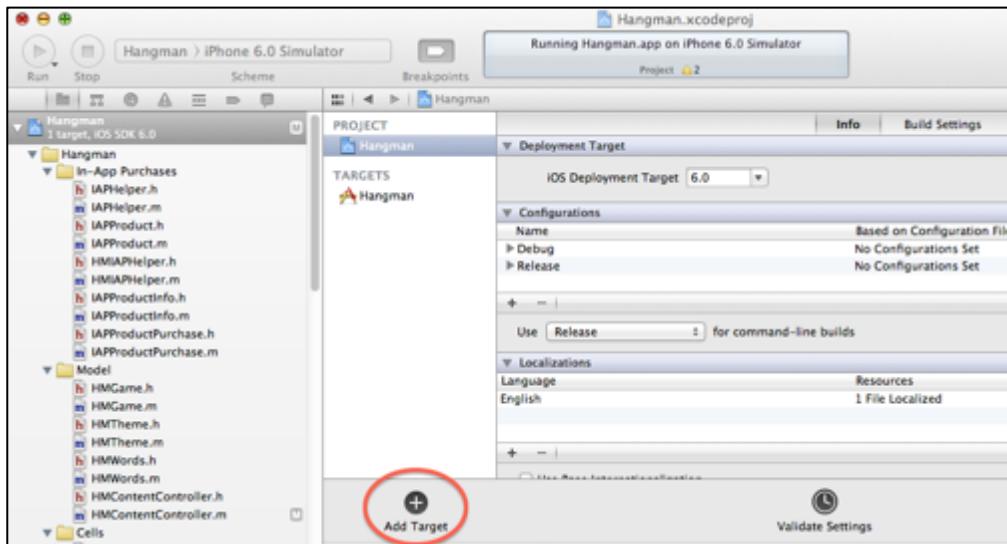
| 7 In-App Purchases | | | | | |
|--------------------|--------------------------------|----------------|-----------|--------------------------|--|
| Reference Name | Product ID | Type | Apple ID | Status | |
| 10 Hints | com.razewa....zombies.tenhints | Consumable | 548953153 | 🔴 Waiting for Screenshot | |
| 100 Hints | com.razewa...bles.hundredhints | Consumable | 548953202 | 🔴 Waiting for Screenshot | |
| 50 Hints | com.razewa...ombies.fiftyhints | Non-Consumable | 549014672 | 🔴 Waiting for Screenshot | |
| Hard Words | com.razewa...zombies.hardwords | Non-Consumable | 548964850 | 🔴 Waiting for Screenshot | |
| iOSWords | com.razewa....zombies.ioswords | Non-Consumable | 548965706 | 🔴 Waiting for Screenshot | |
| Robot Theme | com.razeware.zombies.robot | Non-Consumable | 549019395 | 🟡 Waiting for Upload | |
| Zombie Theme | com.razeware.zombies.zombie | Non-Consumable | 549019397 | 🟡 Waiting for Upload | |

Note that the ones you just set up are in the "Waiting for Upload" state. This means the App Store is ready for you to send it the content for your In-App Purchases – so let's do that now!

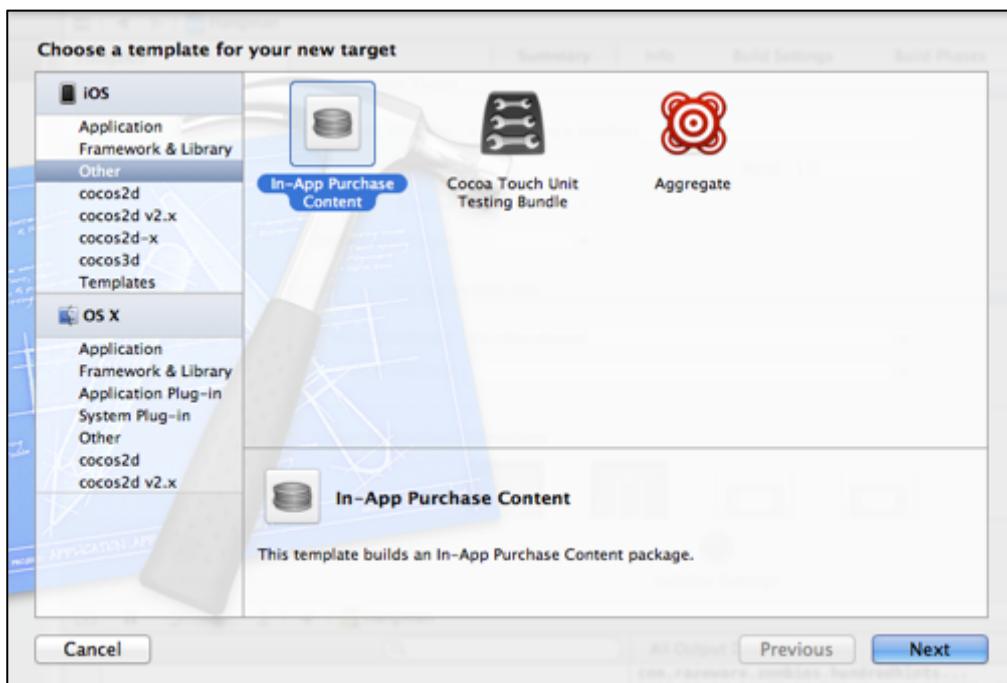
Creating Hosted Content packages

To create the packages of files to upload to the App Store, you have to use the new In-App Purchase target built into Xcode.

Try it out. In the Project Navigator, select the Hangman project and click the "Add Target" button:



Then select **iOS\Other\In-App Purchase Content**:



Name the target **robot** and click **Finish**. Don't worry too much about the Company Identifier or Bundle Identifier for now, you can change them later.

Next, add another new target following the same steps, but name it **zombie** this time.

At this point, if you scroll down in the Project Navigator, you should see two folders representing what will be included in the robot and zombie In-App Purchase content targets:



Right now there's not much – each one just has a single file, **ContentInfo.plist**.

Open **ContentInfo.plist** for the **robot** target, and you'll see the following:

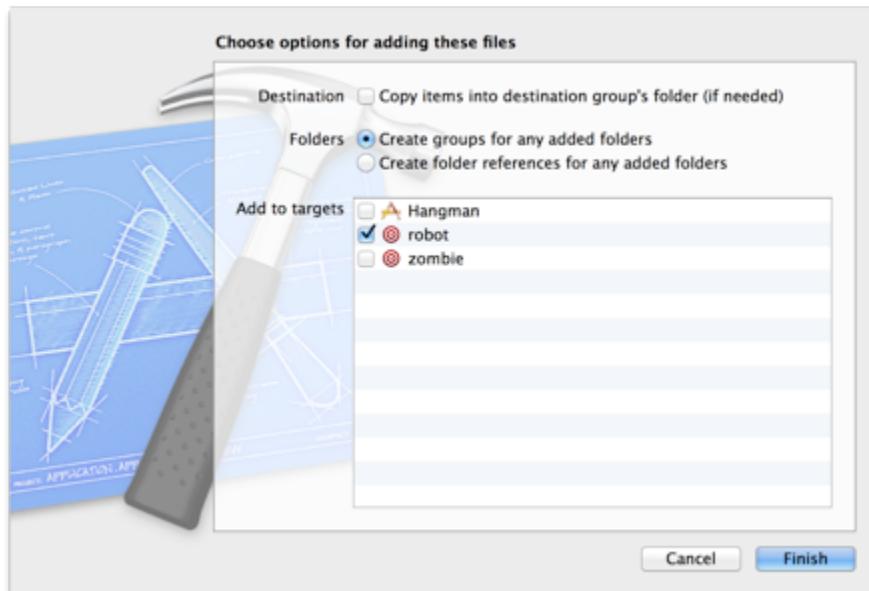
| Key | Type | Value |
|-----------------------------|------------|--|
| ▼ Information Property List | Dictionary | (2 items) |
| ContentVersion | String | 1.0 |
| IAPPProductIdentifier | String | com.razeware.\${PRODUCT_NAME}rfc1034identifier |

This is the file that the App Store and StoreKit will read to find out metadata about your hosted content. Right now it just contains two values – the version number, and the product identifier.

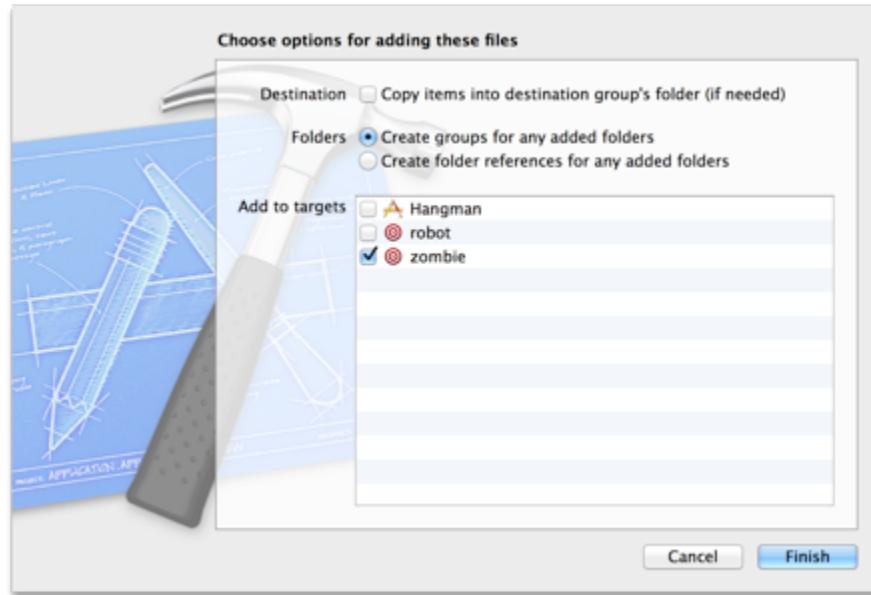
But this product identifier isn't right for us. Double click it and replace it with whatever identifier you set up in iTunes Connect. For me, it is com.razeware.hangman.robot.

Next, repeat this for **ContentInfo.plist** in the **zombie** target, replacing the product identifier there as well.

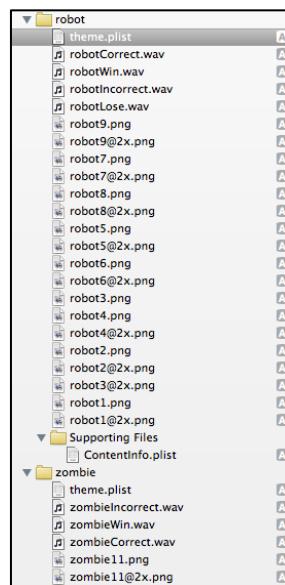
The next step is to add the files for the In-App Purchases to each target. Open up your Finder window and find the **Resources\robot** folder in your project directory. Select all of the files inside, and drag them to the yellow **robot** folder in your Xcode Project Navigator. Make sure **Copy items into destination group's folder (if needed)** is **not** checked (no reason to make an additional copy) and the **robot** target is checked, and click **Finish**.



Repeat this for the **Resources\zombie** folder in Finder, dragging them all to the yellow **zombie** folder and making sure the **zombie** target is checked when you add them.



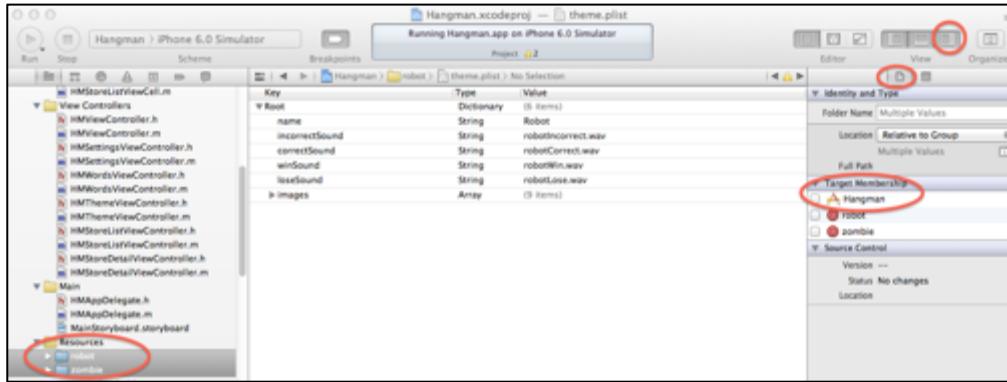
At this point, your robot and zombie folders should look something like this:



Now that you've added all of these files to downloadable content, there is no longer any need to embed them in your application bundle. However, you might need to add them back in for testing, so what you'll do is keep them in your project, but just remove them from the Hangman target.

So select your **blue** robot and zombie folders (under Resources), and bring up the Utilities panel. In the first tab (the File Inspector), uncheck the box next to

Hangman. The files will no longer be included in your project, but if you ever want to test them again, all you have to do is re-check this box.



Finally, comment the lines from **HMContentController.m** that unlock the robot and zombie themes:

```
//[self unlockThemeWithURL:[resourceURL
//    URLByAppendingPathComponent:@"robot"]];
//[self unlockThemeWithURL:[resourceURL
//    URLByAppendingPathComponent:@"zombie"]];
```

Delete the app off your simulator or device, do a clean build, and start it up again. The themes should no longer be there. But just like the zombies and robots they are, they will be back from the grave soon!

Uploading the packages

At this point you have made two file packages containing all the necessary files, and tested to make sure they work. Now it's time to upload the file packages to the App Store.

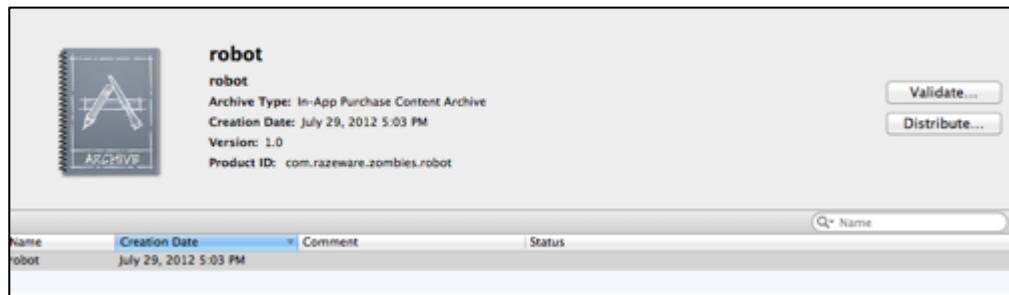
To do this, in the toolbar in Xcode switch your active target to **robot** and the target device to **iOS Device** (or if you have your device connected, select your device name):



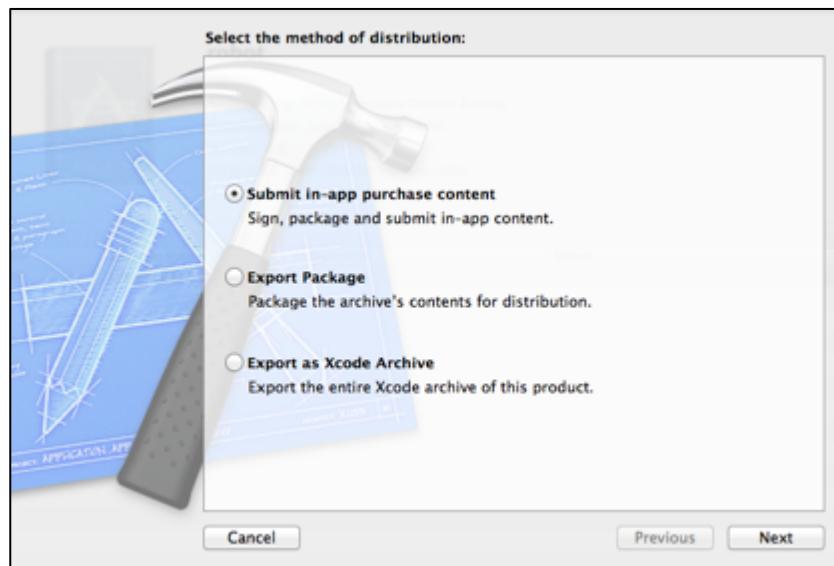
Then go to **Product\Archive** to build it to submit to the App Store.

Note: If the active target is set to use a simulator, the **Archive** option under the Product menu will be grayed out. So be sure to select a device.

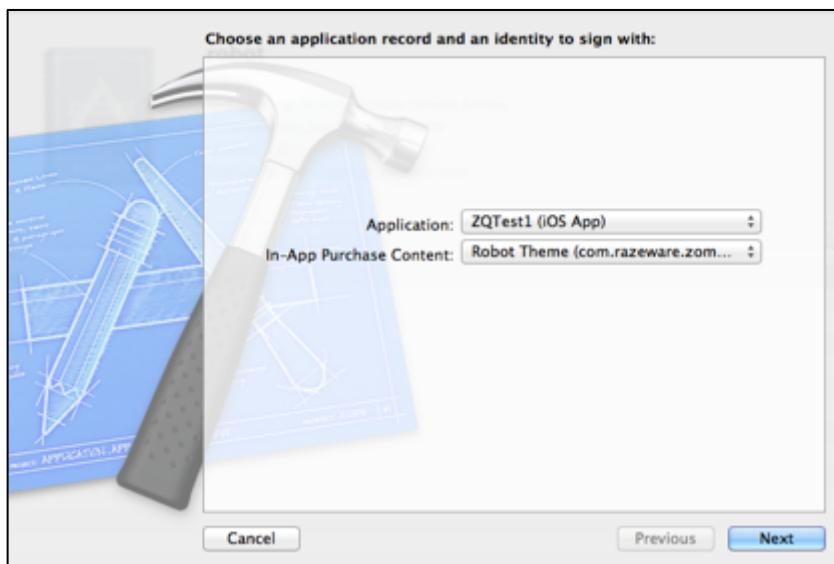
You will see it appear in Xcode's Organizer:



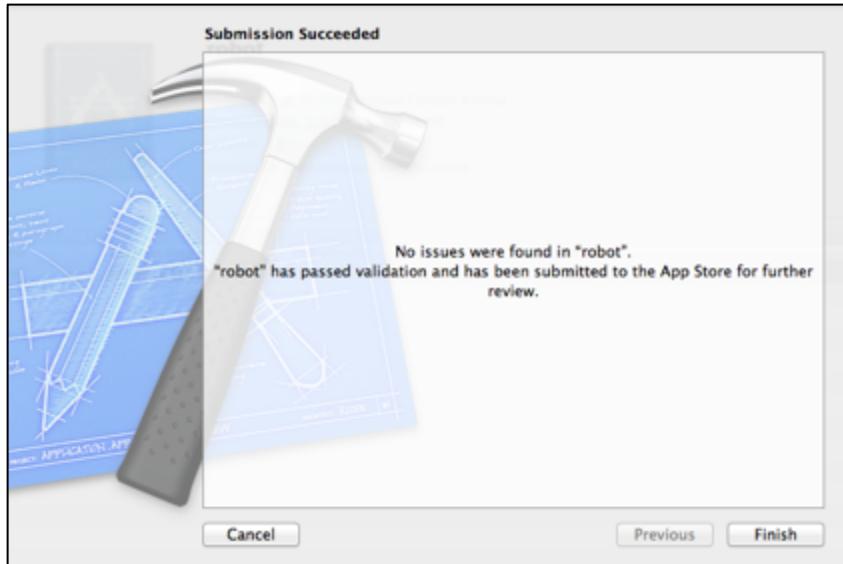
Click **Distribute**, and **Submit in-app purchase content**, then **Next**:



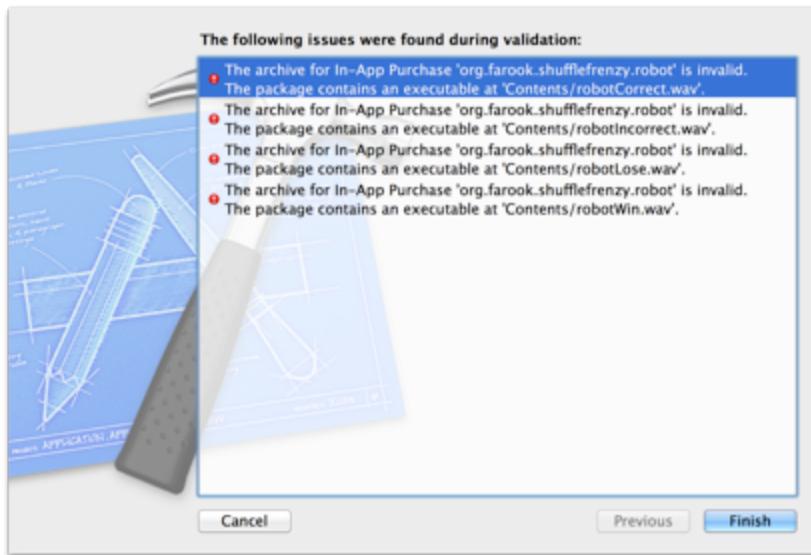
If a popup appears asking for Keychain Access, click Always Allow. Enter in your login information, and click **Next**. Choose the appropriate app and In-App Purchase that you set up in iTunes Connect:



You should see a progress indicator showing your package being uploaded, and then the following screen when it completes:



If you do get an error at this stage, it might be something like this:



The above is usually an indication that the mentioned files have the “executable” bit set for their file permissions. All you really need to do is modify the file permissions for the files mentioned so that there is no executable permission, create a new archive in Xcode, and then upload it to iTunes Connect again. Simple. ☺

Now go back to iTunes Connect, and click on the entry for your robot In-App Purchase. You should see an entry showing that it has your package ready to go!

Hosting Content with Apple

Details for your delivered package are shown below.



robot.pkg

1.0

07/29/2012 02:09 PM

Note: It sometimes takes a little while for your uploaded packages to show up in iTunes Connect. If you don't see it, just wait a few minutes and refresh the page.

Next, repeat these steps for the **zombie** target, and verify that it successfully uploads in iTunes Connect.

Hosting Content with Apple

Details for your delivered package are shown below.



zombie.pkg

1.0

07/29/2012 02:15 PM

What manner of evil are you about to unleash upon the world? ☺

Downloading your content

Now you have all the pieces in place: your In-App Purchases are registered as hosted content, the packages are made and uploaded to the store. It's time to write some code to download this content!

First, you need to modify `IAPPProduct` to store information about the current download and its progress. So open `IAPPProduct.h` and add this to the top of the file:

```
@class SKDownload;
```

`SKDownload` is the StoreKit class that represents a download. You'll learn more about what it contains later in the chapter.

Next, add two properties to the file:

```
@property (nonatomic, strong) SKDownload * skDownload;
@property (nonatomic, assign) float progress;
```

And that's it for `IAPPProduct`. Now open `IAPHelper.m` and modify `provideContentForTransaction:productIdentifier:` to the following:

```
- (void)provideContentForTransaction:  
    (SKPaymentTransaction *)transaction  
    productIdentifier:(NSString *)productIdentifier {  
  
    IAPPProduct * product = _products[productIdentifier];  
  
    if (transaction.downloads) {  
  
        product.skDownload = transaction.downloads[0];  
        if (transaction.downloads.count > 1) {  
            NSLog(@"Unexpected number of downloads!");  
        }  
        [[SKPaymentQueue defaultQueue]  
            startDownloads:transaction.downloads];  
  
    } else {  
  
        // Put the code from before here  
  
    }  
}
```

This is the method that gets called after a successful receipt validation. Here you check to see if it has any downloads, and if so, start the downloads by calling `startDownloads:` on `SKPaymentQueue`.

Note that the API says there are downloads (plural) even though you can only upload one download in iTunes Connect. This is because the API was designed to be future-proof. However, for now there's only one download per In-App Purchase, so you double-check that this is true here and log a warning if not.

Now that you have queued a download, you will start to receive updates as the download progresses and eventually completes. To receive these updates, you have to implement the `paymentQueue:updatedDownloads:` callback from the `SKPaymentTransactionObserver`, so do that next:

```
- (void)paymentQueue:(SKPaymentQueue *)queue  
updatedDownloads:(NSArray *)downloads {  
  
    // 1  
    SKDownload * download = [downloads objectAtIndex:0];  
    SKPaymentTransaction * transaction = download.transaction;  
    SKPayment * payment = transaction.payment;  
    NSString * productIdentifier = payment.productIdentifier;  
    IAPPProduct * product = _products[productIdentifier];
```

```
// 2
product.progress = download.progress;

// 3
 NSLog(@"Download state: %d", download.downloadState);
if (download.downloadState == SKDownloadStateFinished) {

    // 4
    [self purchaseNonconsumableAtURL:download.contentURL
        forProductIdentifier:productIdentifier];
    product.purchaseInProgress = NO;
    [[SKPaymentQueue defaultQueue] finishTransaction:
        transaction];

} else if (download.downloadState ==
SKDownloadStateFailed) {

    // 5
    NSLog(@"Download failed.");
    [self notifyStatusForProductIdentifier:productIdentifier
        string:@"Download failed."];
    product.purchaseInProgress = NO;
    [[SKPaymentQueue defaultQueue] finishTransaction:
        transaction];

} else if (download.downloadState ==
SKDownloadStateCancelled) {

    // 6
    NSLog(@"Download cancelled.");
    [self notifyStatusForProductIdentifier:productIdentifier
        string:@"Download cancelled."];
    product.purchaseInProgress = NO;
    [[SKPaymentQueue defaultQueue] finishTransaction:
        transaction];

} else {
    // 7
    NSLog(@"Download for %@: %0.2f complete",
        productIdentifier, product.progress);
}
}
```

There's a good bit of code here and it's an important method, so let's go over it section-by-section.

1. Pulls out a bunch of information you care about from the first download (again, the assumption is there is only one). At the end of this chain, you have found your `IAPProduct` class, which you'll need to get and store information about the product.
2. An `SKDownload` stores the progress of how far the download has progressed in a property called `progress`, that ranges from 0 to 1. It also can be -1 before the download has started. Here you update the `progress` field on the `IAPProduct` class with this value. This is just to make watching for progress updates easier with Key-Value Observing, as you'll see later.
3. An `SKDownload` stores its current state in a property called `downloadState`. It's important to check this to see if the download has finished or failed, as shown here.
4. If the download is finished, it automatically unzips the package into a directory, and you can access that directory with the `contentURL` property on the `SKDownload`. The only problem is that the directory it gives you is in the `Caches` directory, so unless you copy it somewhere safe, it will eventually be deleted. Luckily, that is not a problem for you, because earlier you made sure that `purchaseNonconsumableAtURL:forProductIdentifier:` copies any contents given to it to the `libraries` directory. So you can use that same method you've been using all along to unlock the purchase, and then finish up the transaction. Note that you know the purchase is non-consumable, because Hosted Downloads are not currently supported for consumables.
5. If the download fails, it finishes the transaction and flashes an alert.
6. The behavior is similar if the download is cancelled. A download can only be cancelled by user interaction, and that can only happen if you specifically add a button to let the user do this, as you'll see later.
7. Otherwise, print out the current status of the download. This will be useful for debugging purposes – you can simply look at the Console log to see how the downloads are coming along.

There's one final change to make in this file. Inside `purchaseNonconsumableAtURL:forProductIdentifier:`, replace the line that sets `contentVersion` to an empty string with the following:

```
// 1
NSString * contentVersion = @"";
NSURL * contentInfoURL = [localURL
    URLByAppendingPathComponent:@"ContentInfo.plist"];
exists = [[NSFileManager defaultManager]
    fileExistsAtPath:contentInfoURL.path
    isDirectory:&isDirectory];
if (exists) {
    // 2
    NSDictionary * contentInfo = [NSDictionary
        dictionaryWithContentsOfURL:contentInfoURL];
```

```
contentVersion = contentInfo[@"ContentVersion"];
NSString * contentsPath = [libraryRelativePath
    stringByAppendingPathComponent:@"Contents"];
// 3
NSString * fullContentsPath = [[self libraryPath]
    stringByAppendingPathComponent:contentsPath];
if ([[NSFileManager defaultManager]
    fileExistsAtPath:fullContentsPath]) {
    libraryRelativePath = contentsPath;
    localPath = [[self libraryPath]
        stringByAppendingPathComponent:libraryRelativePath];
    localURL = [NSURL URLWithString:localPath
        isDirectory:YES];
}
}
```

This is a slight tweak to this method so that it can deal with downloaded file packages. Up until now, you've been assuming that the directory passed to this method contains the contents you're trying to unlock – i.e. words.plist, or themes.plist and its associated files.

However, remember that a hosted content package is a directory with a special file inside (ContentInfo.plist), and a subdirectory called Contents with your stuff inside. So you need to detect this case, and rework the paths when this happens.

Let's go over it section-by-section:

1. Checks to see if this directory contains ContentInfo.plist. If it does, you know this is a hosted download.
2. Pulls out the content version from ContentInfo.plist so it can be stored on the IAPPProductPurchase. This will be handy later when you want to check if a product has been updated since it was purchased. It also checks to see if the Contents subdirectory exists (it should).
3. If the Contents Directory exists, it updates the URL to work with from then on to be the Contents directory, where your stuff actually resides.

It's almost time to test this out! You have no more code changes to make, but you do need to update your productInfos.plist file on your server to add entries for these new purchases, and upload the new icons.

First open your productInfos.plist file on your server and add the new purchase entries, similar to the following:

```
<dict>
    <key>productIdentifier</key>
    <string>com.razeware.hangman.robot</string>
    <key>icon</key>
```

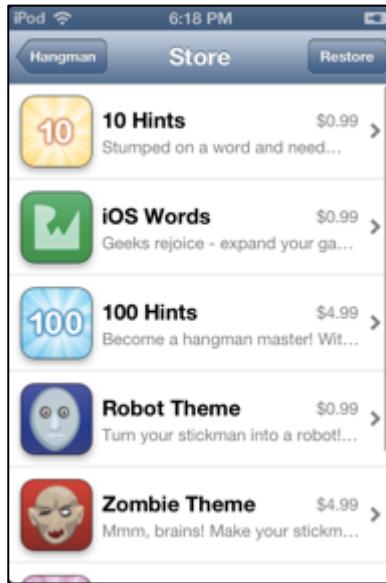
```
<string>http://www.razeware.com/clients/hangman2/IAPInfo/icon_robo
t.png</string>
    <key>consumable</key>
    <false/>
    <key>consumableIdentifier</key>
    <string></string>
    <key>consumableAmount</key>
    <integer>0</integer>
    <key>bundleDir</key>
    <string></string>
</dict>
<dict>
    <key>productIdentifier</key>
    <string>com.razeware.hangman.zombie</string>
    <key>icon</key>
<string>http://www.razeware.com/clients/hangman2/IAPInfo/icon_zomb
ie.png</string>
    <key>consumable</key>
    <false/>
    <key>consumableIdentifier</key>
    <string></string>
    <key>consumableAmount</key>
    <integer>0</integer>
    <key>bundleDir</key>
    <string></string>
</dict>
```

Of course, you should change the product identifiers and URLs to your own values.

Next, in the resources for this chapter you will find a directory called **ThemeIcons** with the icons for these new In-App Purchases. Copy them to your web server alongside the other icons.

And guess what – it's finally time to test this out! Be sure to run the test on a real device, as Hosted Content is not fully supported on the Simulator at the time of writing this chapter.

Build and run the app, and you should see your new In-App Purchases in the store:



Choose an item and purchase it. If you look in your Console log, you'll see debug messages showing the download progress:

```
2012-07-29 18:19:04.186 Hangman[4581:997] Download for com.razeware.zombies.zombie: 0.81 complete
2012-07-29 18:19:04.706 Hangman[4581:997] Download state: 1
2012-07-29 18:19:04.707 Hangman[4581:997] Download for com.razeware.zombies.zombie: 0.95 complete
2012-07-29 18:19:05.301 Hangman[4581:997] Download state: 1
2012-07-29 18:19:05.302 Hangman[4581:997] Download for com.razeware.zombies.zombie: 0.95 complete
2012-07-29 18:19:05.956 Hangman[4581:997] Download state: 1
2012-07-29 18:19:05.957 Hangman[4581:997] Download for com.razeware.zombies.zombie: 0.95 complete
2012-07-29 18:19:06.272 Hangman[4581:997] Download state: 1
2012-07-29 18:19:06.274 Hangman[4581:997] Download for com.razeware.zombies.zombie: 1.00 complete
2012-07-29 18:19:06.327 Hangman[4581:997] Download state: 3
2012-07-29 18:19:06.336 Hangman[4581:997] Copying directory from file://localhost/private/var/mobile/Applications/
B1E002D7-670F-4695-B9E4-19D7A454E246/Library/Caches/C7841DCA-3656-4931-BAFB-ABC1C67D1847.zip/ to file://localhost/var/mobile/Applications/
B1E002D7-670F-4695-B9E4-19D7A454E246/Library/C7841DCA-3656-4931-BAFB-ABC1C67D1847.zip/
```

You should then see a popup showing the successful download:



Go ahead and enjoy your first Hosted Content In-App Purchase!



If you've made it this far, you've got brains – so watch out for those zombies!

Displaying progress

The Hosted Content downloads are functional at this point, but it's a bit annoying from a user's perspective because you can't see how the download is progressing.

As you know, `SKDownload` gives you a progress field you can use to display something to the user – you just haven't added any user interface code to take advantage of this yet. So let's do that now!

Open **HMStoreDetailViewController.m** and add these imports to the top of the file:

```
#import "IAPPProductPurchase.h"
#import "IAPPProductInfo.h"
#import "PrettyBytes.h"
```

Next, modify `viewWillAppear` and `viewWillDisappear` to the following:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    selfStatusLabel.hidden = YES;
    [self refresh];
    [self.product addObserver:self
        forKeyPath:@"purchaseInProgress" options:0 context:nil];
    [self.product addObserver:self forKeyPath:@"purchase"
        options:0 context:nil];
    [self.product addObserver:self forKeyPath:@"progress"
        options:0 context:nil];
```

```

    }

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self.product removeObserver:self
        forKeyPath:@"purchaseInProgress" context:nil];
    [self.product removeObserver:self forKeyPath:@"purchase"
        context:nil];
    [self.product removeObserver:self forKeyPath:@"progress"
        context:nil];
}

```

The only difference is now you are watching to see when the `progress` field on `IAPPProduct` changes as well.

Now for the big change – making your `refresh` method a lot smarter. Start by replacing the line that sets the version label to the following:

```

if (_product.skProduct.downloadable) {
    int numBytes = [_product.skProduct.downloadContentLengths[0]
        integerValue];
    NSString * numBytesString = prettyBytes(numBytes);
    self.versionLabel.text = [NSString
        stringWithFormat:@"Version %@ (%@)",
        _product.skProduct.downloadContentVersion,
        numBytesString];
} else {
    self.versionLabel.text = @"Version 1.0";
}

```

This demonstrates two cool properties on `SKProduct`:

1. `downloadContentLengths`: This is an array of lengths for each of the downloads, in bytes. Again, your assumption is that there is only one, so you pull this up and “beautify” it into a nice string like “2.25 MB” using a helper library.
2. `downloadContentVersion`: This is the most recent version of the download that is available. Note that it might be newer than the one on your device – you might have installed it a few days ago, but then an update came out!

So if the product has a download, here you pull out this information and display it to the user so they can easily see the latest version of the product that is available, and how big it is.

Next add the following code to the bottom of `refresh`:

```

if (self.product.purchaseInProgress) {
    selfStatusLabel.hidden = NO;
    self.progressView.hidden = NO;
}

```

```
if (self.product.skDownload) {
    self.pauseButton.hidden = NO;
    self.resumeButton.hidden = NO;
    self.cancelButton.hidden = NO;
    switch (self.product.skDownload.downloadState) {
        case SKDownloadStateActive: {
            if (self.product.skDownload.timeRemaining >= 0) {
                selfStatusLabel.text = [NSString
                    stringWithFormat:@"Active (%0.2fs
remaining)...",
                    self.product.skDownload.timeRemaining];
            } else {
                selfStatusLabel.text = @"Active...";
            }
        }
        break;
        case SKDownloadStateWaiting: {
            if (self.product.skDownload.timeRemaining >= 0) {
                selfStatusLabel.text = [NSString
                    stringWithFormat:@"Waiting (%0.2fs
remaining)...",
                    self.product.skDownload.timeRemaining];
            } else {
                selfStatusLabel.text = @"Waiting...";
            }
        }
        break;
        case SKDownloadStateFinished:
            selfStatusLabel.text = @"Download Finished.";
            break;
        case SKDownloadStateFailed:
            selfStatusLabel.text = @"Download failed.";
            break;
        case SKDownloadStatePaused:
            selfStatusLabel.text = @"Paused.";
            break;
        case SKDownloadStateCancelled:
            selfStatusLabel.text = @"Cancelled";
            break;
        default:
            selfStatusLabel.text = @"Unexpected state!";
            break;
    }
    self.progressView.progress = self.product.progress;
} else {
```

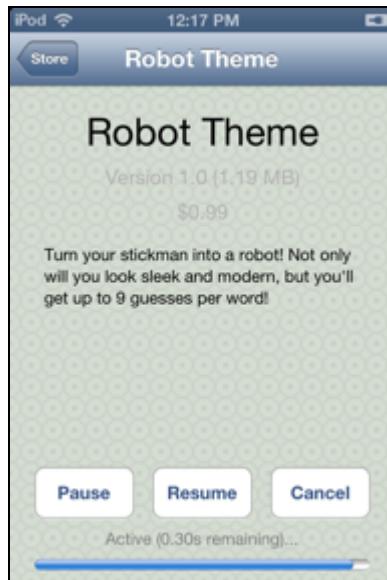
```
        selfStatusLabel.text = @"Installing...";  
        self.progressView.progress = self.product.progress;  
    }  
} else if (!self.product.purchase.consumable &&  
           self.product.purchase) {  
    selfStatusLabel.hidden = NO;  
    self.progressView.hidden = YES;  
    if (self.product.skProduct.downloadContentVersion &&  
        (![self.product.skProduct.downloadContentVersion  
         isEqualToString:self.product.purchase.contentVersion])) {  
        selfStatusLabel.text = @"Update Available, Please  
Restore";  
    } else {  
        selfStatusLabel.text = @"Installed";  
    }  
} else if (self.product.info.consumable) {  
    selfStatusLabel.hidden = NO;  
    self.progressView.hidden = YES;  
    int newValue = [[NSUserDefaults standardUserDefaults]  
                  integerForKey:self.product.info.consumableIdentifier];  
    selfStatusLabel.text = [NSString stringWithFormat:@"Current  
value: %d", newValue];  
} else {  
    selfStatusLabel.hidden = YES;  
    self.progressView.hidden = YES;  
}
```

There's a lot of code here, but most of it is pretty simple. I won't explain everything, but I will point out a few of the highlights:

- If a purchase is in progress and there's an `skDownload`, this will display its current status. It displays the current state, and if the download is in the "active" or "waiting" state, it shows the download progress.
- `SKDownload` also includes a handy method called `timeRemaining`. This is the estimated number of seconds that it will take to finish downloading the package. Here you display it to the user.
- This view controller contains a progress view and cancel/resume/pause buttons that are usually hidden. This code reveals them and updates the progress view when appropriate.
- This code compares the content version of the purchase to the latest version that is available (from `skProduct`). If there's a newer version available, it tells the user about it.
- Finally, if the user is looking at a non-consumable, it shows the current value for the non-consumable for convenience.

Look through the code and make sure you understand what it does. Then build and run and download a product. (If you've already downloaded both products, just delete the app off your device and re-install.)

This time, you should see additional information and a nice status bar as the product downloads!



Now that the detail view controller is updating properly, you might as well update the list view controller. Open **HMStoreListViewController.m** and replace the lines in `tableView:cellForRowAtIndexPath:` that set the price label with the following:

```
if (product.purchaseInProgress) {
    cell.priceLabel.text = @"Installing...";
} else if (!product.purchase.consumable && product.purchase) {
    if (product.skProduct.downloadContentVersion &
        ![product.skProduct.downloadContentVersion
        isEqualToString:product.purchase.contentVersion]) {
        cell.priceLabel.text = @"Update";
    } else {
        cell.priceLabel.text = @"Installed";
    }
} else if (product.allowedToPurchase) {
    cell.priceLabel.text = [_priceFormatter
        stringFromNumber:product.skProduct.price];
} else {
    NSLog(@"Unexpected product state!");
    cell.priceLabel.text = @"";
}

if (product.skDownload.downloadState == SKDownloadStateActive) {
```

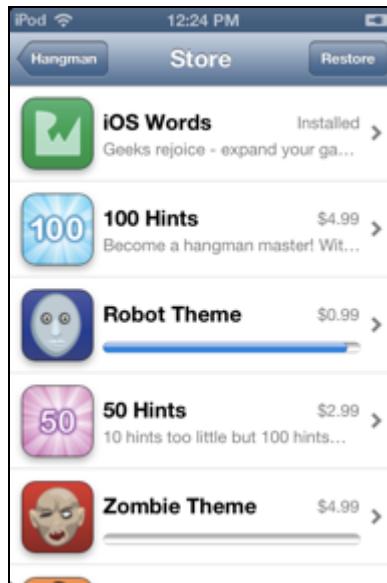
```
    cell.descriptionLabel.hidden = YES;
    cell.progressView.hidden = NO;
    cell.progressView.progress = product.progress;
} else {
    cell.descriptionLabel.hidden = NO;
    cell.progressView.hidden = YES;
}
```

This is very similar logic to what you did in the detail controller. Also modify `addObservers` and `removeObservers` to watch/stop watching the progress variable:

```
- (void)addObservers {
    if (_observing || _products == nil) return;
    _observing = TRUE;
    for (IAPPProduct * product in _products) {
        [product addObserver:self
            forKeyPath:@"purchaseInProgress" options:0 context:nil];
        [product addObserver:self forKeyPath:@"purchase"
            options:0 context:nil];
        [product addObserver:self forKeyPath:@"progress"
            options:0 context:nil];
    }
}

- (void)removeObservers {
    if (!_observing) return;
    _observing = FALSE;
    for (IAPPProduct * product in _products) {
        [product removeObserver:self
            forKeyPath:@"purchaseInProgress" context:nil];
        [product removeObserver:self forKeyPath:@"purchase"
            context:nil];
        [product removeObserver:self forKeyPath:@"progress"
            context:nil];
    }
}
```

Go ahead and test it out now – a good way to do this is to delete and reinstall the app, and then restore all purchases.



Pausing, Resuming, and Canceling

You might be wondering about those Pause/Resume/Cancel buttons. Right now they don't do anything, but implementing them is really easy. Let's give it a shot.

Open **IAPHelper.h** and pre-declare three methods:

```
- (void)pauseDownloads:(NSArray *)downloads;
- (void)resumeDownloads:(NSArray *)downloads;
- (void)cancelDownloads:(NSArray *)downloads;
```

Then open **IAPHelper.m** and implement them as follows:

```
- (void)pauseDownloads:(NSArray *)downloads {
    [[SKPaymentQueue defaultQueue] pauseDownloads:downloads];
}

- (void)resumeDownloads:(NSArray *)downloads {
    [[SKPaymentQueue defaultQueue] resumeDownloads:downloads];
}

- (void)cancelDownloads:(NSArray *)downloads {
    [[SKPaymentQueue defaultQueue] cancelDownloads:downloads];
}
```

Pretty easy, eh? To call them, open **HMSStoreDetailViewController.m** and replace the `pauseTapped:`, `resumeTapped:`, and `cancelTapped:` methods as follows:

```
- (IBAction)pauseTapped:(id)sender {
    [[HMIAPHelper sharedInstance]
```

```
    pauseDownloads:@[self.product.skDownload]];
}

- (IBAction)resumeTapped:(id)sender {
    [[HMIAPHelper sharedInstance]
        resumeDownloads:@[self.product.skDownload]];
}

- (IBAction)cancelTapped:(id)sender {
    [[HMIAPHelper sharedInstance]
        cancelDownloads:@[self.product.skDownload]];
}
```

Build and run, and download a product, but try pausing and resuming it as it goes along.

Unfortunately, at the time of writing this chapter, if you pause and resume a download it will cause the download to fail! Hopefully this will be fixed in a future update – but at least you know what do when it's fixed. ☺

Updates to Hosted Content

You might be curious about how to get updates to Hosted Content over to your users. Here's the basic process:

1. You make a new version of your content and upload it to iTunes Connect with an increased ContentVersion number in your ContentInfo.plist.
2. Then the user needs to restore their purchases through the “Restore” button you added. (Yes, unfortunately at the time of writing this chapter, there is no way to upgrade an individual purchase without restoring all.)
3. As long as your code is smart enough to handle re-downloading a new version of the old content, you're good to go. And lucky for you, your code already deals with this situation. ☺

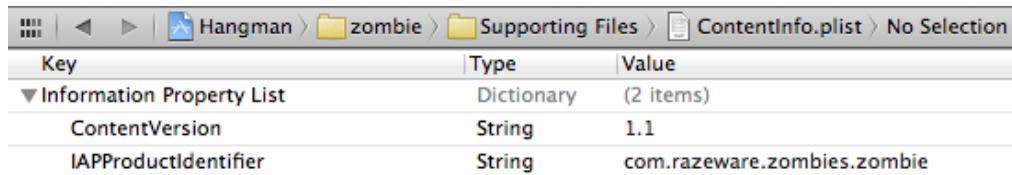
Let's try this out. In the resources for this chapter there is a folder called **ZombieAlt**. This represents a new version of your zombie In-App Purchase, where the Zombie sports a party hat! Hey, zombies like to party too!



Before you begin, make sure you have the current zombie theme downloaded and installed on your device. After all, you're trying to simulate an upgrade!

Next, copy the new **ZombieAlt** files and paste them into your **zombie** directory in Finder, overwriting your current files. In Xcode, go inside your yellow **zombie** directory and verify that the correct files appear.

Then open **zombie\Supporting Files\ContentInfo.plist** and modify the ContentVersion to a higher version number:



| ContentInfo.plist | | |
|---------------------------|------------|-----------------------------|
| Key | Type | Value |
| Information Property List | Dictionary | (2 items) |
| ContentVersion | String | 1.1 |
| IAPPProductIdentifier | String | com.razeware.zombies.zombie |

Next, follow the same process you did earlier in this chapter to upload the new update to iTunes Connect, and verify the update appears there.

Switch back to your app, and select the Zombie product from the store. Note that it can take a few minutes for the App Store to synchronize, and in the meantime there is a chance your product might not show up in the list.

When it does show up, you should see a message that says “Update Available, Please Restore” (or, if you are on the store list screen, simply “Update” next to the zombie theme):



Go back to the list screen and restore your purchases via the Restore button. After a few moments your purchases will be restored, and if you select the Zombie theme you will see the new party hat!



Congratulations, you now know everything there is to know about Hosted Content! You can use your knowledge to add downloadable content to your apps without needing to host it yourself, opening up a lot of options for selling things to your users!

Just please, don't let loose any more zombies. I'm running out of ammo!

Selling iTunes content

iOS 6 has introduced a neat new feature that lets you sell iTunes Store products right within your app, with a built-in view controller. If you have an app that allows users to buy music, movies, books, or more from iTunes, this might be right up your alley!

This feature is especially handy for apps that have content on the App Store made by the owner of the app. For example, an app for a musician could sell the musician's tracks directly inside the app.

Note: Unfortunately, at the time of writing this chapter it seems there is no way to send in an affiliate identifier so you can get the ~10% referral credit that you would normally get through the [iTunes Affiliate](#) program. I wouldn't be surprised if this is added sometime soon, though.

In the Hangman app, you don't have any content of your own to sell in the iTunes Store. However, one thing this game could really use is some groovy tunes. And all this talk about zombies has got me hankering for the good old Castlevania soundtrack. So you're going to add the ability for users to buy Castlevania music directly within your app that they can then play in the background as they game!

You'll start with the easiest possible way to get this working (showing a predetermined item), and then you'll learn an even cooler way (searching for items).

Note: This section is completely optional, so if you don't think this is something that would be useful for your apps, feel free to skip at any point to the next section, "Validating Receipts on your Server."

Showing a predetermined item

Before you start coding, you should figure out what song you want to display.

Well, one time I attended this really cool concert called [Video Games Live](#) and it had this amazing rendition of Castlevania that left me spellbound in geek heaven. So let's use that one!

If you search for "site:itunes.apple.com video games live castlevania" in Google, you'll find the following page:

| Name | Artist | Time | Price | Action |
|---------------------------------|-----------------------------|------|--------|--------------------------------|
| 1 Kingdom Hearts (Live) | Crouch End Festival Chro... | 3:39 | \$1.29 | View in iTunes |
| 2 Warcraft Suite (Live) | Crouch End Festival Chro... | 5:05 | \$1.29 | View in iTunes |
| 3 Myst Medley (Live) | Crouch End Festival Chro... | 6:04 | \$1.29 | View in iTunes |
| 4 Medal of Honor (Live) | Crouch End Festival Chro... | 5:43 | \$1.29 | View in iTunes |
| 5 Civilization IV Medley (Live) | Crouch End Festival Chro... | 4:55 | \$1.29 | View in iTunes |
| 6 Medley: Tetris (Live) | Martin Leung | 1:09 | \$1.29 | View in iTunes |
| 7 God of War Montage (Live) | Crouch End Festival Chro... | 3:35 | \$1.29 | View in iTunes |
| 8 Advent Rising Suite (Live) | Crouch End Festival Chro... | 6:27 | \$1.29 | View in iTunes |
| 9 Tron Montage (Live) | Crouch End Festival Chro... | 5:18 | \$1.29 | View in iTunes |
| 10 Halo Suite (Live) | Crouch End Festival Chro... | 6:38 | \$1.29 | View in iTunes |
| 11 Castlevania (Live) | Crouch End Festival Chro... | 4:44 | \$1.29 | View in iTunes |
| 12 Falling (Live) | Tommy Tallarico | 2:21 | \$1.29 | View in iTunes |

This is the one I was talking about! For now, you'll just link to the entire album. But to do this, you need an ID. How do you get that?

If you look at the URL in your browser, you'll see something like this:

<http://itunes.apple.com/ca/album/video-games-live-vol.-1/id286201200>

Aha! The link contains the ID of the item you wish to display: 286201200. That's what you need to pass to Apple's new `SKStoreProductViewController`!

Let's try this out. You're going to create a new row in your store that displays the `SKStoreProductViewController` with this item, if the user were to select that row.

Open **HMStoreListViewController.m**, add an import to the top of the file, and mark the class as implementing the `SKStoreProductViewControllerDelegate`:

```
#import "MBProgressHUD.h"

@interface HMStoreListViewController() <UIAlertViewDelegate,
SKStoreProductViewControllerDelegate>
@end
```

You're importing `MBProgressHUD.h`, which is a handy helper library good for displaying modal "Loading" messages when long-running operations take place, such as starting up this view controller. ☺

Also, you mark the class as implementing the `SKStoreProductViewControllerDelegate`. This way it can notify you when the user is done shopping.

Next, modify `tableView:numberOfRowsInSection:` to return an extra row for the music store:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return _products.count + 1;
}
```

Similarly, modify `tableView:cellForRowAtIndexPath:` to fill in the new row appropriately:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    HMStoreListViewController *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];

    if (indexPath.row < _products.count) {

        // Put old code here . . .

    } else {

        cell.iconImageView.image = [UIImage
imageNamed:@"icon_music.png"];
        cell.titleLabel.text = @"Game Music";
        cell.descriptionLabel.text = @"Get some great music for
the game!";
    }
}
```

```
    cell.priceLabel.text = @"Various";
    cell.descriptionLabel.hidden = NO;
    cell.progressView.hidden = YES;

}

return cell;
}
```

And finally, modify `tableView:didSelectRowAtIndexPath:` to call a new method when the new row is tapped:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (indexPath.row < _products.count) {
        [self performSegueWithIdentifier:@"PushDetail"
            sender:indexPath];
    } else {
        [self showMusicStore];
    }
}
```

The only part left is the fun stuff – displaying the new view controller! Add these methods next:

```
#pragma mark - Music Store

- (void)showMusicStore {

    // 1
    MBProgressHUD * hud = [MBProgressHUD
        showHUDAddedTo:self.view animated:YES];
    hud.labelText = @"Loading...";

    // 2
    SKStoreProductViewController * viewController =
        [[SKStoreProductViewController alloc] init];
    viewController.delegate = self;

    // 3
    NSDictionary * parameters =
        @{@"SKStoreProductParameterITunesItemIdentifier":
            @286201200};
    [viewController loadProductWithParameters:parameters
        completionBlock:^(BOOL result, NSError *error) {
```

```
[MBProgressHUD hideHUDForView:self.view animated:YES];
if (result) {
    // 4
    [self presentViewController:viewController
        animated:YES completion:nil];
} else {
    NSLog(@"Failed to load products: %@", error);
}
};

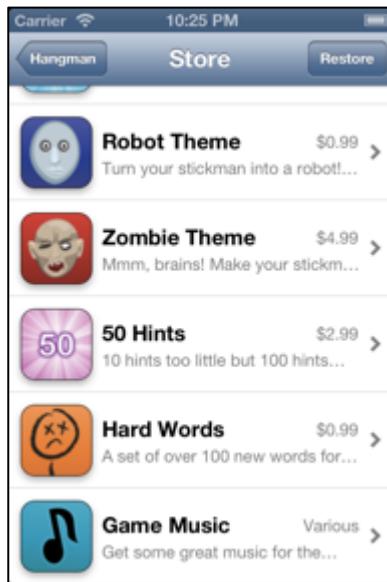
-
(void)productViewControllerDidFinish:(SKStoreProductViewController *)
viewController {
    // 5
    NSLog(@"Finished shopping!");
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

Let's go over this bit-by-bit.

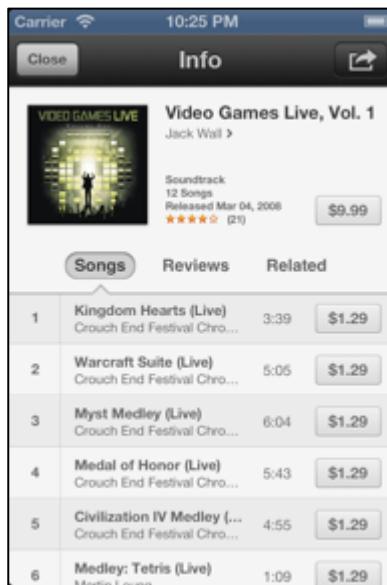
1. Starting up the view controller takes a while because it has to search for the product to display first. So here you start up a loading indicator so that the user isn't left confused as to what's going on.
2. Initialize the view controller and set the current class as its delegate so the current class gets a callback when the user is done.
3. Set up the parameters to pass to the `SKStoreProductViewController`. As of writing this chapter, there is only one parameter you can pass to the view controller – the ID of the item to display. This is the ID that you found from the iTunes URL. The code then passes the dictionary in a call to `loadProductWithParameters:completionBlock:` so that iTunes can look up that ID – this is the part that can take a while.
4. If the results come back successfully, then display the view controller.
5. When the user is done shopping, dismiss the view controller.

One final step – inside the resources of this project, you'll find a folder named **MusicIcon** with a file **icon_music.png** inside. Drag that into your Resources folder and make sure it's added to the Hangman target.

And that's it! Build and run, go to your store and you will see a new Game Music entry:



Tap the entry, and a Loading dialog will display. After a while, the new `SKStoreProductViewController` will appear, allowing the user to buy some groovy tunes from right within your app!



The iTunes search API

If you have a small number of items you're displaying, the above method is great – but if you have a large number of items you want to sell, looking up item IDs manually is a drag. This is where the iTunes Search API comes to the rescue!

The iTunes Search API is a well-documented web service API that you can call to find information about items available on the iTunes Store. It was originally intended for web sites that might want to link to items on iTunes to sell. You can read all about it on the official Search API documentation page:

<http://www.apple.com/itunes/affiliates/resources/documentation/itunes-store-web-service-search-api.html>

Using it is incredibly easy. You just send a GET request to a URL with several key/value pairs, like this:

```
http://itunes.apple.com/search?key1=value1&key2=value2&...
```

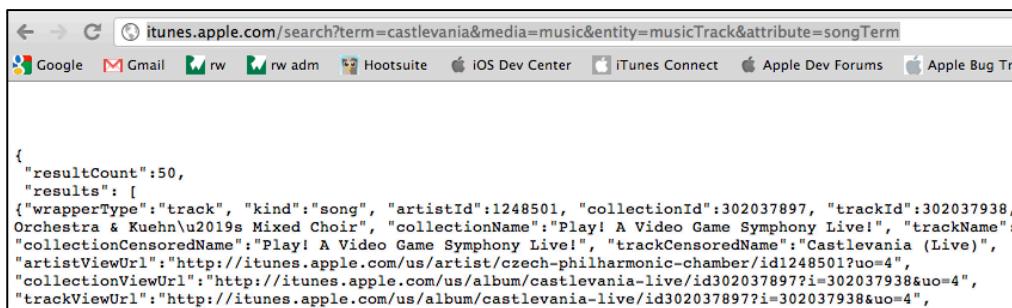
You'll take advantage of this cool API in Hangman by searching for any song that has the word Castlevania in it, and then display them all in a neat list. To do this, you'll use the following parameters:

- **“term” = “castlevania”**: Term is short for “search term” here. In short, it means find me some Castlevania songs, dammit!
- **“media” = “music”**: You can search for different media types in iTunes like movies, podcasts, music, or more. You choose music here.
- **“entity” = “musicTrack”**: Even though you’re searching for music, the API could return different things – an entire album, or the artist for example. All you’re interested in is tracks, so you set that here.
- **“attribute” = “songTerm”**: Your search query could target different parts of the music. You want to search the name of the song.

Putting it all together, you get a URL like this:

<http://itunes.apple.com/search?term=castlevania&media=music&entity=musicTrack&attribute=songTerm>

Put that into your web browser, and you’ll see it works! It returns a bunch of JSON, including cool Castlevania music.

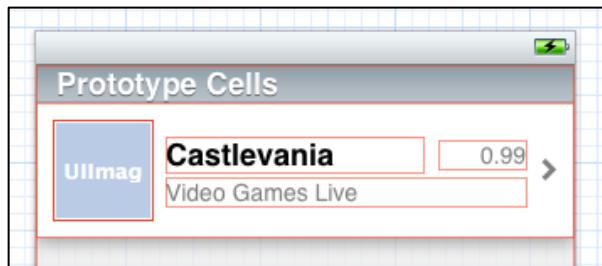


A screenshot of a web browser window displaying the JSON response from the iTunes search API. The URL in the address bar is `http://itunes.apple.com/search?term=castlevania&media=music&entity=musicTrack&attribute=songTerm`. The browser's toolbar includes icons for Google, Gmail, Hootsuite, iOS Dev Center, iTunes Connect, Apple Dev Forums, and Apple Bug Tracker. The JSON response is shown in a code block:

```
{  
  "resultCount":50,  
  "results": [  
    {"wrapperType":"track", "kind":"song", "artistId":1248501, "collectionId":302037897, "trackId":302037938, "collectionName":"Play! A Video Game Symphony Live!", "trackName":"Orchestra & Kuehn\u2019s Mixed Choir", "collectionCensoredName":"Play! A Video Game Symphony Live!", "trackCensoredName":"Castlevania (Live)", "artistViewUrl":"http://itunes.apple.com/us/artist/czech-philharmonic-chamber/id1248501?uo=4", "collectionViewUrl":"http://itunes.apple.com/us/album/castlevania-live/id302037897?i=302037938&uo=4", "trackViewUrl":"http://itunes.apple.com/us/album/castlevania-live/id302037897?i=302037938&uo=4", ...]
```

Pretty awesome! Feel free to look through the above results to see what you get back. You get a lot of great info, such as the all-important ID to send to the store view controller, some art to display, and much more. Let’s put this to good use!

First of all, you need a new view controller for the music store. Open **MainStoryboard.storyboard**, and drag a new table view controller onto your storyboard near the Store Detail View Controller. Set the table view row height to 80, and then lay out a prototype cell similar to the following:



Set the identifier for the table view cell to “Cell” in the Attributes inspector. Also, control-drag from the Store List View Controller to the new table view controller, and select “Push” to create a new segue. Name the new segue **PushMusic** in the Attributes inspector.

Next, you need to make a class for the table view cell. Control-click on the **Cells** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **HMMusicCell**, make it a subclass of **UITableViewCell**, and click **Next** and finally **Create**.

Replace **HMMusicCell.h** with the following:

```
@interface HMMusicCell : UITableViewCell

@property (weak, nonatomic) IBOutlet UIImageView *iconImageView;
@property (weak, nonatomic) IBOutlet UILabel *titleLabel;
@property (weak, nonatomic) IBOutlet UILabel *descriptionLabel;
@property (weak, nonatomic) IBOutlet UILabel *priceLabel;

@end
```

These just contain properties for all of the user interface elements you created.

Open **MainStoryboard.storyboard** again, select the prototype cell you created, and set the class to **HMMusicCell** in the Identity Inspector. Then control-click on the cell and connect all four properties to the correct user interface elements.

There’s one last helper class to make, for information about each track you will display in this table view. Control-click on the **Model** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **HMMusicInfo**, make it a subclass of **NSObject**, and click **Next** and finally **Create**.

Replace **HMMusicInfo.h** with the following:

```
@interface HMMusicInfo : NSObject

@property (nonatomic, assign) int trackId;
@property (nonatomic, strong) NSString * trackName;
@property (nonatomic, strong) NSString * artistName;
```

```
@property (nonatomic, assign) float price;
@property (nonatomic, strong) NSString * artworkURL;

- (id)initWithTrackId:(int)trackId
    trackName:(NSString *)trackName
    artistName:(NSString *)artistName price:(float)price
    artworkURL:(NSString *)artworkURL;

@end
```

This is a simple model class that just stores a bunch of information to display. You will retrieve this info from the search API and store it in this class.

Switch to **HMMusicInfo.m** and add the implementation:

```
#import "HMMusicInfo.h"

@implementation HMMusicInfo

- (id)initWithTrackId:(int)trackId
    trackName:(NSString *)trackName
    artistName:(NSString *)artistName price:(float)price
    artworkURL:(NSString *)artworkURL {
    if ((self = [super init])) {
        self.trackId = trackId;
        self.trackName = trackName;
        self.artistName = artistName;
        self.price = price;
        self.artworkURL = artworkURL;
    }
    return self;
}

@end
```

This is just a simple initializer – not much to mention.

Now, time to make a view controller class for this. Control-click on the **View Controllers** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **HMMusicViewController**, make it a subclass of **UITableViewController**, and click **Next** and finally **Create**.

Before you forget – go back to MainStoryboard.storyboard and set the class of the new table view controller you created to **HMMusicViewController** in the Identity Inspector.

Open **HMMusicViewController.m** and replace it with the following:

```
#import "HMMusicViewController.h"
#import "AFHTTPClient.h"
#import "AFHTTPRequestOperation.h"
#import "HMMusicInfo.h"
#import "HMMusicCell.h"
#import "UIImageView+AFNetworking.h"
#import "MBProgressHUD.h"
#import <StoreKit/StoreKit.h>

@interface HMMusicViewController : UIViewController
<SKStoreProductViewControllerDelegate>
@end

@implementation HMMusicViewController {
    NSMutableArray * _musicInfos;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.refreshControl = [[UIRefreshControl alloc] init];
    [self.refreshControl addTarget:self action:@selector(reload)
        forControlEvents:UIControlEventValueChanged];
    [self reload];
    [self.refreshControl beginRefreshing];
}

- (void)reload {
    _musicInfos = [NSMutableArray array];
    [self.tableView reloadData];
    [self requestMusic];
}

#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
```

```
    return _musicInfos.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    HMMusicCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier
        forIndexPath:indexPath];

    HMMusicInfo * info = [_musicInfos
        objectAtIndex:indexPath.row];

    cell.titleLabel.text = info.trackName;
    cell.descriptionLabel.text = info.artistName;
    cell.priceLabel.text = [NSString stringWithFormat:@"$%0.2f",
        info.price];
    [cell.iconImageView setImageWithURL:[NSURL
        URLWithString:info.artworkURL] placeholderImage:[UIImage
        imageNamed:@"icon_placeholder.png"]];

    return cell;
}

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    HMMusicInfo * info = [_musicInfos
        objectAtIndex:indexPath.row];

    MBProgressHUD * hud = [MBProgressHUD
        showHUDAddedTo:self.view animated:YES];
    hud.labelText = @"Loading...";

    SKStoreProductViewController * viewController =
        [[SKStoreProductViewController alloc] init];
    viewController.delegate = self;
    NSDictionary * parameters =
        @{@"SKStoreProductParameterITunesItemIdentifier": [NSNumber
            numberWithInt:info.trackId]};

    [viewController loadProductWithParameters:parameters
        completionBlock:^(BOOL result, NSError *error) {
            [MBProgressHUD hideHUDForView:self.view animated:YES];
    }];
}
```

```
        if (result) {
            [self presentViewController:viewController
                animated:YES completion:nil];
        } else {
            NSLog(@"Failed to load product: %@", error);
        }
    }];
}

- (void)productViewControllerDidFinish:
    (SKStoreProductViewController *)viewController {
    NSLog(@"Finished shopping!");
    [self dismissViewControllerAnimated:YES completion:nil];
}

@end
```

This looks like a lot of code, but it's pretty standard stuff. Basically, it displays a list of `HMMusicInfo` classes in the table view. When you select a row, it opens up an `SKStoreProductViewController`, just as you did in the last section.

There's one missing method – and that's the new stuff related to the Search API. Add this next (anywhere in the file):

```
- (void)requestMusic {

    // 1
    NSURL * url = [NSURL
        URLWithString:@"http://itunes.apple.com/"];
    AFHTTPClient * client = [[AFHTTPClient alloc]
        initWithBaseURL:url];
    NSDictionary * parameters = @{
        @"term" : @"castlevania",
        @"media" : @"music",
        @"entity" : @"musicTrack",
        @"attribute" : @"songTerm"
    };
    [client getPath:@"/search" parameters:parameters
        success:^(AFHTTPRequestOperation *operation, id
        responseObject) {
        // 2
        NSError * error;
        NSDictionary * searchResults = [NSJSONSerialization
            JSONObjectWithData:operation.responseData options:0
            error:&error];
    }];
}
```

```
if (searchResults == nil) {
    NSLog(@"Failure parsing response: %@", error);
} else {
    // 3
    NSArray * results = searchResults[@"results"];
    for (NSDictionary * result in results) {
        int trackId = [result[@"trackId"] intValue];
        NSString * trackName = result[@"trackName"];
        NSString * artistName = result[@"artistName"];
        float price = [result[@"trackPrice"]
            floatValue];
        NSString * artworkURL = result[@"artworkUrl60"];

        // 4
        HMMusicInfo * musicInfo = [[HMMusicInfo alloc]
            initWithTrackId:trackId trackName:trackName
            artistName:artistName price:price
            artworkURL:artworkURL];
        [_musicInfos addObject:musicInfo];
    }

    // 5
    [self.tableView reloadData];
}
[self.refreshControl endRefreshing];
} failure:^(AFHTTPRequestOperation *operation,
NSError *error) {
    NSLog(@"Error searching for songs: %@", error);
    [self.refreshControl endRefreshing];
}];
```

}

Let's go over this section-by-section.

1. This uses `AFHTTPClient` to make the search request, since it's nice and easy. Although you could construct the URL manually, here you let `AFHTTPClient` build it for you by passing it a dictionary of the keys and values.
2. When the request finishes, your block of code will fire. Here you convert the JSON into an `NSDictionary` using iOS's built-in JSON deserialization capabilities. To learn more about this, check out Chapter 23, "Working with JSON in iOS 5" in [iOS 5 by Tutorials](#).
3. Inside the root dictionary, the "results" key contains an array of more dictionaries – one for each song. This then pulls out the interesting information from the returned JSON. The keys to look for in the results were determined by

looking at the returned JSON in the browser and referring to the Search API guide.

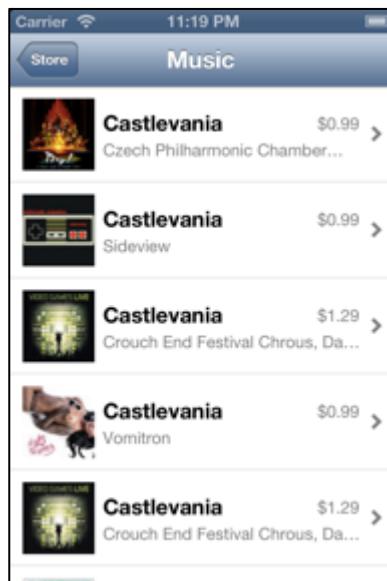
4. Put all of the information into your handy `HMMusicInfo` model class and add it to the list.

5. Reload the table view, and everything in the list will be displayed.

One finishing touch – go to `HMStoreListViewController.m` and modify `tableView:didSelectRowAtIndexPath:` to trigger the new segue rather than calling the method that you wrote earlier to display the music store for a single item:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (indexPath.row < _products.count) {
        [self performSegueWithIdentifier:@"PushDetail"
            sender:indexPath];
    } else {
        //*[self showMusicStore];
        [self performSegueWithIdentifier:@"PushMusic"
            sender:indexPath];
    }
}
```

Build and run, and tap on your Game Music store. A list of awesome Castlevania music will display! Simon would be proud.



Validating receipts on your server

At this point, you have a pretty sweet In-App Purchase setup. Users can buy consumables and non-consumables, you have support for built-in and downloadable purchases, you dynamically load your products from your own server, and you even have your own iTunes Store within the app!

However, there's still one less than ideal piece to this puzzle – you're validating receipts by contacting the App Store directly from your own app, rather than doing it on your own server, the way Apple recommends.

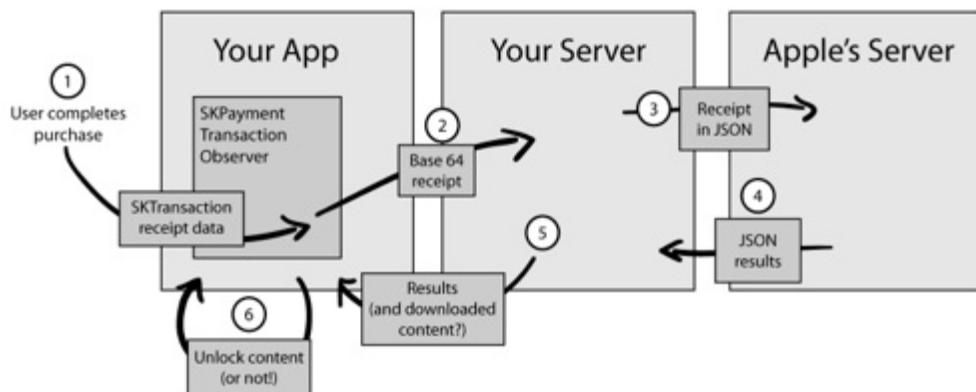
Since you have your own server set up already to host your list of products and icons, you might as well add receipt validation. It's actually not that hard!

Note: This section requires you to have MySQL and PHP on your server and to have some basic familiarity with them. If you are completely new to these topics, you should check out this tutorial first:

<http://www.raywenderlich.com/2941/how-to-write-a-simple-phpmysql-web-service-for-an-ios-app>

How receipt validation works

Apple recommends taking the following approach to performing receipt validations:



1. When a user completes (or restores) a purchase, a unique `SKTransaction` is made with a unique transaction ID. In your `SKPaymentTransactionObserver` callback, you will receive this `SKTransaction` object. It contains an `NSData` property called `transactionReceipt`. This is cryptographically-signed data about the transaction. You should treat it as private opaque data and shouldn't mess around with it – that's what Apple's receipt validation service is for.
2. You should send this receipt to your own server (ideally in Base64 encoding, because that's what Apple's receipt validation service will expect later).

3. Your own server will then send the encoded receipt data to Apple's validation service.
4. It will return success or failure based on whether the receipt is valid or not.
5. Your server forwards the results back to your app. If you are not using Apple's Hosted Content and are instead providing the downloadable content yourself, you would also return the downloadable content at this stage if validation succeeded.
6. You then unlock the content (or not) based on whether the receipt is valid (or not).

In this section, you'll try this out by implementing every part of the process from scratch! You'll also learn a few security improvements you can make to the process.

Bare bones

When working on a server-side project like this, I find the easiest way is to get the bare bones skeleton up and working first, and then iterate on it from there.

So to start, you'll make a PHP script that can accept two parameters from your app: the encoded receipt to validate, and whether to validate against the production servers or the real servers. The PHP script will then simply log out what it got, but always return success. Once you have that working, you'll expand on it from there.

Go to your web server and create a new file called **index.php**. Replace the contents with the following:

```
<?php

// 1
define('SUCCESS', '0');

// 2
function result($status, $error = '') {
    return array(
        'status' => $status,
        'error' => $error
    );
}

// 3
function logToFile($msg) {
    $fd = fopen('log.txt', 'a');
    $str = '[' . date('Y/m/d h:i:s', mktimes()) . '] ' . $msg;
    fwrite($fd, $str . "\r\n");
    fclose($fd);
}
```

```
// 4
function validateReceipt($receipt, $sandbox)
{
    logToFile(print_r(array($receipt, $sandbox), true));
    return result(SUCCESS);
}

// 5
$receipt = $_POST['receipt'];
$ sandbox = $_POST['sandbox'];
$retval = validateReceipt($receipt, $sandbox);

// 6
header('content-type: application/json; charset=utf-8');
echo json_encode($retval);

?>
```

Let's go over this section-by-section.

1. This defines a constant you will use to indicate the receipt was successfully validated. Right now, for testing, you will always return this.
2. This is a helper function that you'll use to return a result to the caller. It will include the status code (again, right now always 0 for success), and if there's a failure, also a string describing the error.
3. This is a helper function to log a message out to a file. For this to work, your directory needs to be writable by the process that runs your web server. You should check that now and set permissions before you forget.
4. This is the main function to validate the receipt. It takes two parameters – a Base64-encoded receipt, and a Boolean that indicates whether to use the sandbox servers or production servers. Right now this function just logs these values out to a file, and returns success.
5. This is the code that initially gets called when the script runs. It looks for two POST values: receipt and sandbox. It then passes them to the validateReceipt function.
6. In the end, it converts the array returned by validateReceipt into JSON, and displays the result to the caller using the echo command.

Let's try this out! Open **IAPHelper.m** and add the following import to the top of the file:

```
#import "NSData+Base64.h"
```

This is a helper extension that converts an `NSData` buffer to a Base64-encoded string.

Next add a new constant for the path to where you saved index.php, relative to your web server's base URL:

```
static NSString *const IAPServerVerifyURL =  
@"/clients/hangman2/";
```

You should replace this with the server path to wherever you happened to save the index.php file.

Finally, comment out your old validateReceiptForTransaction: method and add this new implementation instead:

```
- (void)validateReceiptForTransaction:  
    (SKPaymentTransaction *)transaction {  
  
    IAPPProduct * product =  
        _products[transaction.payment.productIdentifier];  
  
    // 1  
    NSString * receiptString =  
        [transaction.transactionReceipt base64EncodedString];  
  
    // 2  
    NSURL * url = [NSURL URLWithString:IAPServerBaseURL];  
    AFHTTPClient * httpClient = [[AFHTTPClient alloc]  
        initWithBaseURL:url];  
    NSDictionary * params = @{@"receipt" : receiptString,  
        @"sandbox" : @YES };  
    [httpClient postPath:IAPServerVerifyURL parameters:params  
        success:^(AFHTTPRequestOperation *operation, id  
            responseObject) {  
  
        // 3  
        NSError * jsonError;  
        NSDictionary * json = [NSJSONSerialization  
            JSONObjectWithData:operation.responseData  
            options:kNilOptions error:&jsonError];  
        if (!json) {  
            // 4  
            NSString * responseString =  
                operation.responseText;  
            NSLog(@"Failure parsing response: %@. Server response:  
            %@", jsonError, responseString);  
            [self notifyStatusForProductIdentifier:  
                transaction.payment.productIdentifier  
                string:@"Validation failed."];  
        }  
    }];  
}
```

```
        product.purchaseInProgress = NO;
        [[SKPaymentQueue defaultQueue] finishTransaction:
            transaction];
    } else {

        // 5
        int status = [json[@"status"] integerValue];
        NSString * error = json[@"error"];
        if (status != 0) {
            NSLog(@"Failure verifying receipt: %@", error);
            [self notifyStatusForProductIdentifier:
                transaction.payment.productIdentifier
                string:@"Validation failed."];
            product.purchaseInProgress = NO;
            [[SKPaymentQueue defaultQueue]
                finishTransaction: transaction];
        }

        // 6
        else {
            NSLog(@"Successfully verified receipt!");
            [self provideContentForTransaction:transaction
                productIdentifier:
                    transaction.payment.productIdentifier];
        }
    }

} failure:^(AFHTTPRequestOperation *operation,
NSError *error) {
    // 7
    NSLog(@"Failure connecting to server: %@", error);
    [self
notifyStatusForProductIdentifier:transaction.payment.productIdentifier
string:@"Validation failed."];
    product.purchaseInProgress = NO;
    [[SKPaymentQueue defaultQueue] finishTransaction:
transaction];
}];

}
```

Once again, let's take this one section at a time:

1. Convert the receipt into a Base64-encoded string. This is necessary because Apple's receipt validation service expects it in this format.
2. Determine the full path to your index.php script, pass the receipt and whether to use the sandbox as parameters, and issue the request. Notice that it's hard-coded

to always send YES to use the sandbox here – you'd have to switch that to NO for production.

3. Upon successful return from the web service, attempt to convert the response into JSON.
4. If this fails, log out the response string – because this could mean you have an error in your PHP script, and this is a nice, easy way to see more info about the problem!
5. If the JSON parsed OK, pull out the status code and message from the response. If the status indicates that the validation failed, notify the user and finish the transaction.
6. If the validation succeeded, unlock the content as usual.
7. Handle the other error case (most likely a server connectivity problem).

Phew – finally done! Build and run, and attempt to make an In-App purchase as usual. It should succeed, and if you look at your server's log file (named log.txt and in the same location as the index.php file) you should see the Base64-encoded receipt printed out:

```
[2012/07/29 11:53:43] Array
(
    [0] => ewoJInNpZ25hdHVyZSIgPSAiQWd6MTlly3hPbUtuQTR3S0oydDI0cnpFYnljbWFm
OWNtUCt1RdkZlh2Tz22Mi9nU3QrSXdNT1RXckNVSERYMnhYUmFhR2QwVkzenpE
UlhZakI2dCsxN1dWtNBeHZVMUI2Rk5WUXJtaG5KU2JtYk5mb2FCNXM4YUxKQmdH
UmoxTFFDMVV4WkhDV2ZFbGNQQm16Ry9rTEFyWEdjUDFSWTR3cEllLzlTVGtsaEFB
QURWekNDQTFNd2dnSTdvQU1DQVFJQ0NHVVVrVTNaV0FTMU1BMEdDU3FHU0liM0RR
RUJCUVVBTUg4eEN6QUpCZ05WQkFZVEFsVlRNUk13RVFZRZRUUtEQXBCY0hCc1pT
QkpibU11TVNzd0pBWURUVFMREIxQmNIQnNaU0JEwlhKMGFXWnBZMkYwYVc5dULF
RjFkR2h2Y21sMGVURXpNREVHQTFVRUF3d3FRWEJ3YkdVZ2FWUjFibVZ6SUZOMGIz
SmxJRU5sY25ScFptbGpZWFJwYjI0Z1FYVjBhRzl5YVhSNU1CNFhEVEE1TURZeE5U
SXlnRFUxTmxvWERURTBNRF14TkRJeU1EVTFObG93WkRFak1DRUdBMVVFQxd3YVVI
)
```

Note: If you do not see log.txt appear on your web server's directory, make sure that the web server has write access to that directory.

Great! Now that you have end-to end communication working along with error handling, you can proceed to writing the code to perform the validation with confidence.

Note: You might have noticed that in the server I am using, I have it set up to use HTTPS rather than HTTP. It is highly recommended you use HTTPS instead of HTTP when developing your own receipt validation server. This helps prevent man-in-the-middle attacks, since your server will have a certificate that proves it is who it says it is.

The good news is that setting up your own HTTPS server does not have to be expensive or difficult. There are many guides out there about how to obtain

and install one, such as this one: <http://library.linode.com/web-servers/apache/ssl-guides/fedora-14>

FYI, I was able to buy an HTTPS certificate through K Software for just \$20: http://www.ksoftware.net/ssl_certs.html

Validating the receipt

Now you'll add the code to validate the receipt. Again, the basic idea is you send a POST to an iTunes URL with the Base64 receipt data. It will then return a status code telling you whether it is a valid receipt or not.

Open **index.php** and add a few status codes to the top of the file:

```
define('ERROR_VERIFICATION_NO_RESPONSE', '1');
define('ERROR_VERIFICATION_FAILED', '2');
```

Then replace the `validateReceipt` function with the following:

```
function validateReceipt($receipt, $sandbox)
{
    // 1
    if ($sandbox) {
        $store =
            'https://sandbox.itunes.apple.com/verifyReceipt';
    } else {
        $store = 'https://buy.itunes.apple.com/verifyReceipt';
    }

    // 2
    $postData = json_encode(array('receipt-data' => $receipt));

    // 3
    $ch = curl_init($store);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_POST, true);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $postData);
    $encodedResponse = curl_exec($ch);
    curl_close($ch);

    // 4
    if ($encodedResponse == false)
    {
        return result(ERROR_VERIFICATION_NO_RESPONSE,
                     'Payment could not be verified (no response data).');
    }
}
```

```
// 5
$response = json_decode($encodedResponse);
$status = $response->{'status'};
$decodedReceipt = $response->{'receipt'};
if ($status != 0)
{
    return result(ERROR_VERIFICATION_FAILED, "Payment could
not be verified (status = $status).");
}

// 6
logToFile(print_r($decodedReceipt, true));

// 7
return result(SUCCESS);

}
```

Here we go again with a section-by-section breakdown:

1. Determine the URL for the store based on whether or not the sandbox parameter is set.
2. The Apple receipt validator expects a JSON-encoded dictionary with a single key ("receipt-data") with the Base-64-encoded receipt as its value. That's exactly what this sets up.
3. This fancy bit of code is what actually performs the request to Apple's receipt validator. Curl is a handy library that allows you to easily perform web requests – think of it being sort of like AFNetworking. Here you set up options to a) return the results as a string rather than outputting directly; b) send the request as a POST; and c) send the JSON data as the POST fields.
4. If there was no response, error out.
5. Decode the response as JSON. Checks the status code – if it's anything other than 0, verification failed, so return an error.
6. The Apple receipt validator returns some extra useful info in addition to success/failure. You log that out here.
7. Finally, return success if you got that far.

Back on your device, purchase another product. Note that you don't even have to change any code on your app or redeploy – this is one of the benefits of server-side development.

It should work as usual, but you are now using proper receipt validation with your own server, congrats! That wasn't too bad, was it?

Security improvements

Since you're validating receipts on your own server, you are already no longer vulnerable to the "In-App Purchase Hack" described in the last chapter.

However, this implementation still has some vulnerabilities:

- If an attacker sends you a receipt from another app, you would pass it to the receipt validator and it would say, "Yep, it's valid." You would then return success, even though it's not even for your app!
- You'll also return success if an attacker sends you a receipt from your app that you've already processed. So as long as the attacker has one valid receipt for your app, they can get unlimited in-app purchases.

You can address this by doing two checks:

1. Make sure the receipt is for one of your products.
2. Make sure you haven't already processed this transaction.

Implementing the first check is pretty easy. Remember how you currently log out the info returned to you from the Apple receipt validator? Take a look at your log and you'll see something like the following:

```
[2012/07/30 12:16:04] stdClass Object
{
    [original_purchase_date_pst] => 2012-07-29 21:16:03 America/Los_Angeles
    [unique_identifier] => 6b41cfe949c8af52ab31d5d953c647637fda10b
    [original_transaction_id] => 1000000053624822
    [btrs] => 1.0
    [transaction_id] => 1000000053624822
    [quantity] => 1
    [product_id] => com.razeware.zombies.tenhints
    [item_id] => 548953153
    [purchase_date_ms] => 1343621763093
    [purchase_date] => 2012-07-30 04:16:03 Etc/GMT
    [original_purchase_date] => 2012-07-30 04:16:03 Etc/GMT
    [purchase_date_pst] => 2012-07-29 21:16:03 America/Los_Angeles
    [bid] => com.razeware.zombies
    [original_purchase_date_ms] => 1343621763093
}
```

Notice that it contains a lot of great information, including the product identifier that the user is attempting to purchase. So you can simply make sure that it is your own product ID.

As for making sure you haven't already processed a transaction, that requires you to create a database table to store the transactions you have processed. Let's get started with that now.

In the resources for this chapter, you will find a folder named **ServerSQL**, with a file called **hangman.sql** inside. Use this SQL to create a database you'll need for the PHP receipt validation script you're about to write.

```
CREATE DATABASE IF NOT EXISTS hangman;
```

```
USE hangman;

CREATE TABLE `transactions` (
  `transaction_id` char(32) NOT NULL,
  `product_id` char(32) NOT NULL,
  `original_transaction_id` char(32) NOT NULL,
  `validation_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`transaction_id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Looking at the SQL, you'll see that it just creates a single table named transactions. This will keep track of every transaction that your script has validated – this way you can include a check when you verify a transaction to make sure you haven't processed it already, which prevents a replay attack.

Next, create a new helper class called **Database.php** on your server, and replace the contents with the following:

```
<?php

class Database extends PDO
{

    static private $host      = "localhost";
    static private $dbname   = "hangman";
    static private $user      = "user";
    static private $pass      = "pass";

    private static $instance = null;

    static function get() {
        if (self::$instance!=null) return self::$instance;

        try {
            self::$instance = new
                Database("mysql:host=".self::$host.
                    ";dbname=".self::$dbname,
                    self::$user, self::$pass);
            return self::$instance;
        }
        catch(PDOException $e) {
            print $e->getMessage();
            return null;
        }
    }
}
```

```
    }  
  
?>
```

Replace “user” and “pass” with a valid username and password for the new database you created. This helper class will make connecting to your database a bit easier.

Next, open **index.php** and modify the top of the file to the following:

```
require_once('Database.php');  
  
define('SUCCESS', '0');  
define('ERROR_VERIFICATION_NO_RESPONSE', '1');  
define('ERROR_VERIFICATION_FAILED', '2');  
define('ERROR_INVALID_PRODUCT_ID', '3');  
define('ERROR_TRANSACTION_ALREADY_PROCESSED', '4');  
  
define('PRODUCT_ID_PREFIX', 'com.razeware.hangman');  
  
function beginsWith($str, $sub) {  
    return (strcmp($str, $sub, strlen($sub)) == 0);  
}
```

Here you import your new database helper class, add two new status codes, add a product ID prefix that you will search for, and a helper function.

Be sure to replace the product ID prefix with your own.

Finally, add the following code to the end of `validateReceipt` (but before the return success):

```
// 1  
$product_id = $decodedReceipt->{'product_id'};  
$transaction_id = $decodedReceipt->{'transaction_id'};  
$original_transaction_id =  
    $decodedReceipt->{'original_transaction_id'};  
  
// 2  
if (!beginsWith($product_id, PRODUCT_ID_PREFIX)) {  
    return result(ERROR_INVALID_PRODUCT_ID,  
        'Invalid product id.');//  
}  
  
// 3  
$db = Database::get();
```

```
$statement = $db->prepare(  
    'SELECT * FROM transactions WHERE transaction_id=?');  
$statement->bindParam(1, $transaction_id, PDO::PARAM_STR, 32);  
$statement->execute();  
  
// 4  
if ($statement->rowCount() > 0) {  
    logToFile("Already processed $transaction_id.");  
    return result(ERROR_TRANSACTION_ALREADY_PROCESSED,  
        'Already processed this transaction.');//  
}  
  
// 5  
else {  
    logToFile("Adding $transaction_id.");  
    $statement = $db->prepare(  
        'INSERT INTO transactions(transaction_id, product_id,  
        original_transaction_id) VALUES (?, ?, ?)');  
    $statement->bindParam(1, $transaction_id,  
        PDO::PARAM_STR, 32);  
    $statement->bindParam(2, $product_id, PDO::PARAM_STR, 32);  
    $statement->bindParam(3, $original_transaction_id,  
        PDO::PARAM_STR, 32);  
    $statement->execute();  
}
```

And now for our last section-by-section tour of the code.

1. Pull out the product ID, transaction ID, and original transaction ID from the information returned to you by Apple's receipt validator. Again, this is the stuff you saw in the log message earlier.
2. Check to see if the product ID begins with the expected prefix, and bail if it doesn't.
3. Issue a database query to check for any transactions with the specified transaction ID.
4. If one exists, return a failure.
5. Otherwise, insert information about the transaction into the database.

Try purchasing a product on your device, and everything should work OK as usual. You can also try the following to make sure everything is working as expected:

- Temporarily set the `PRODUCT_ID_PREFIX` constant to something that is not your product ID prefix. It should reject any transaction.
- Look inside your database and find a transaction ID that is stored there. Then temporarily modify the line that sets `$transaction_id` to be hard-coded to that exact ID. It should reject any transaction.

At this point, you have a pretty robust server-side validation capability. There are still other avenues that a determined attacker could exploit to get through, but you've made things a lot harder and covered the low-hanging fruit. IMHO, anything above this is just icing on the cake!

Where to go from here?

If you've made it through both of these chapters, you are a true In-App Purchase boss!

You can say with confidence that you understand everything there is to know about consumable and non-consumable In-App Purchases, and have practical experience that will help you implement a very robust server-based store in your own app. Huzzah!

And in the process, you have guessed countless hidden iOS terms, callously destroyed many stickmen, and unleashed hordes of killer robots and zombies into the unsuspecting world!

If you still want to know more about In-App Purchases beyond this, you might want to play around with non-renewable and renewable subscriptions. But that is a topic beyond the scope of these chapters. You can read more about it in [Apple's In-App Purchase Programming Guide](#).

I hope you enjoyed these epic chapters, and hope to see you wield In-App Purchases to bring your users even more happiness – and of course, to bring yourself more riches! ☺

Chapter 11: Beginning Social Framework

By Felipe Laso Marsetti

In iOS 5, Apple added system-wide support for Twitter. This meant users could add their Twitter accounts in the Settings app, and from then on any application that needed to use Twitter could do so without requiring the user to log in. Also, iOS 5 provided some new APIs allowing developers to easily integrate Twitter into their apps.

In the end, this made things much nicer for users. It became easier for users to share content without the annoyance of having to constantly log in, and the new APIs increased Twitter support in apps and gave users a common and polished interface.

It also made things much nicer for developers. Integrating Twitter into applications became much easier than it was before. iOS 5 gave developers two new options to add Twitter functionality: using the standard Twitter composer (dubbed the “tweet sheet”) or getting full-blown access to the Twitter web API.

When using the tweet sheet, all you had to do was add a few lines of code (similar to displaying an alert view or email composer) and iOS took care of the rest – authentication, presenting the proper UI, sending the web API calls, and more!

But you’re weren’t just limited to the capabilities of the tweet sheet – you could also access the Twitter web API directly. This was much simpler to do than it was prior to iOS 5 because you no longer had to implement authentication yourself. Instead, you could access the user’s Twitter account information via the new Accounts Framework, and authentication was handled for you.

So yes, iOS 5 was great when it came to Twitter. But iOS 6 takes things even further – it introduces a new *Social Framework* that abstracts the way you work with any social network behind a set of common APIs. The Social Framework currently supports Twiter, Facebook, Sina Weibo (a popular Chinese social network), and more are sure to come in the future.

In this chapter, you’re going to learn how to use the new iOS 6 Social Framework to allow the user to easily share content. In the next chapter, you’ll learn how to

access the social network APIs directly for full access – for example, you'll learn how to programmatically allow a user to "like" your Facebook page!

This chapter does not assume any prior familiarity with the iOS 5 Twitter API, although if you're familiar with it learning the Social Framework will be even easier.

Note: If you have read the Twitter API chapters in *iOS 5 by Tutorials*, note the Twitter Framework has been deprecated in favor of the new Social Framework. Don't worry though – the new Social Framework is quite similar to the Twitter framework, so adapting will be easy – just keep reading to see how.

The other piece of good news is that Apple was wise enough to make the Accounts Framework provider agnostic, so most of that stays the same!

So are you excited to learn about the new Social Framework and how to integrate it into your own apps? It's time to get social!

Share sheets and the activity view

Just like the Twitter APIs provided a built in view controller to make Tweeting easy, the Social Framework provides two built in view controllers to make sharing easy with *any* social network:

- **Share sheets** to allow easy sharing of content (similar to the "tweet sheet").
- The **activity view** to allow the user to choose a method to share their data.

Let's take a look at each of these.

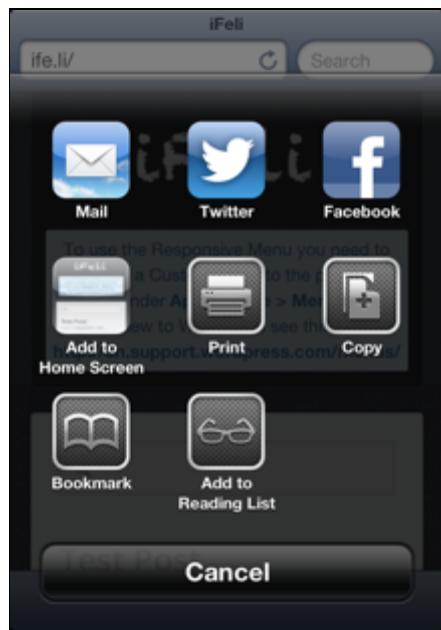
For **share sheets**, you programmatically decide which social network to display, configure what to initially show, and iOS takes care of the rest. Here's an example of a share sheet for Twitter:



The share sheets for Facebook and Sina Weibo are like tweet sheets but with some minor differences in visual appearance. With them you can share photos, links, text, and your current location.

To display a share sheet, you use a class called `SLSComposeViewController`. If you're familiar with the iOS 5 Twitter API, you can think of it as the granddaddy of `TWTweetComposeViewController`. The class is extremely simple and easy to use, and allows you to choose the social framework (Twitter, Facebook, or Sina Weibo) just by specifying the service type.

Next let's look at the **activity view**. This new view controller replaces a lot of the popovers and action sheets present in previous versions of stock iOS apps for sharing and performing actions on the content being consumed. If you have iOS 6 installed, then you've probably noticed this everywhere:



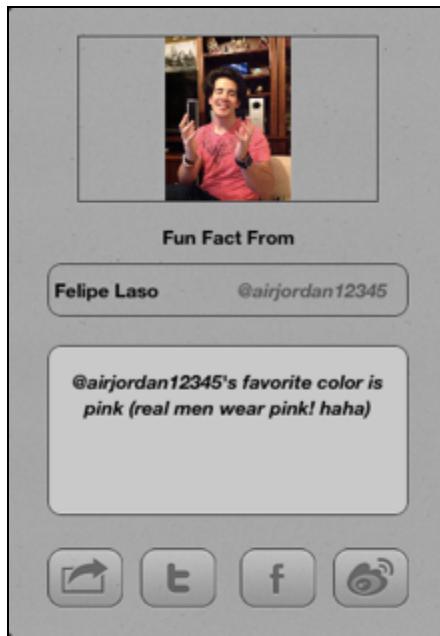
You can easily display this view controller in your own apps through the use of the new `UIActivityViewController` class. By default, an activity view comes with services like copy, email, sharing to social sites, etc. It's also possible to define your own custom actions (another thing you will learn to do in this chapter).

So there you have it: this chapter will focus on using the new share sheets and the activity view, and adding a custom activity to an activity view. Then in the next chapter you'll learn how to go even further and access the Facebook web API directly!

Excited about the possibilities yet? Awesome! Let's get started! 😊

Introducing Fun Facts

Before opening Xcode and getting your code on, let's take a look at the project you will build in this chapter, **Fun Facts**.



The idea behind Fun Facts is to provide you, our reader, with randomized silly and interesting facts about the authors and editors of *iOS 6 by Tutorials*. To get a new fact, all you have to do is shake the device and have a good laugh at the pictures and unexpected facts you will receive.

Once you've generated a fact, you have the option of sharing the image and fact using Twitter, Facebook or Sina Weibo (via their respective equivalents of the "tweet sheet"). Additionally, you will be able to perform certain actions on the fact and image using the activity view.

Finally, you will create your own activity for the activity view controller that will let users export a fact to the device's Photos app using a simple frame decorated with a custom drawing.

Now that you know what to expect with the Fun Facts app, this is a great place to stop with the theory and start with some code! ☺

Getting started

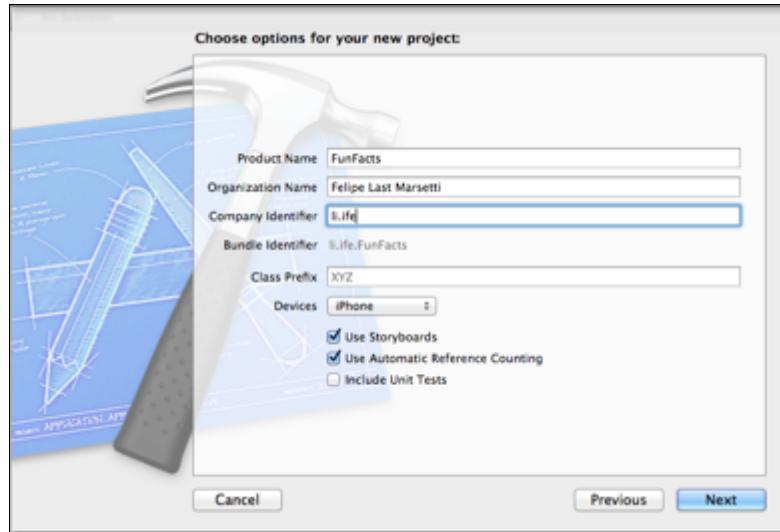
This chapter provides two options for how you can read the chapter:

- **If you want to do everything yourself:** Then keep reading through this section, and you will build the entire Fun Facts project yourself from scratch! This will be a good review of a lot of iOS development topics.
- **If you just want to focus on the Social Framework:** Then skip ahead to the "From action to activity" section. We'll give you a starter project there that has the UI pre-built so you can just focus on the Social Network integration.

Assuming you want to do everything yourself, let's get started!

Open Xcode and create a new Single View Application. For product name, use **FunFacts** and then fill out the Organization Name, Company Identifier and Class Prefix fields with your customary values.

Make sure only iPhone is selected in Devices, and that the Use Storyboards and Use Automatic Reference Counting checkboxes are selected (Unit Tests will not be necessary for this project).



Click Next and select where to create your project.

You should be looking at the summary for the FunFacts target now – don't move away from this window, because there's a small change needed for Supported Interface Orientations. The app will only support the portrait orientation, so be sure to deselect all other options.



Moving on to the Project Navigator (the left sidebar), you will notice the project template created some files already:

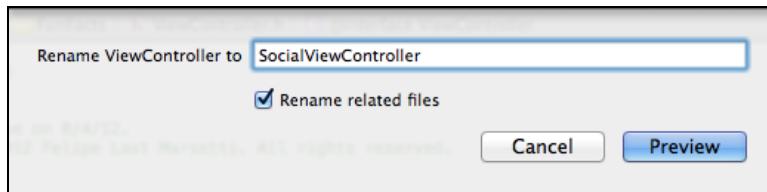
- AppDelegate
- ViewController
- MainStoryboard

MainStoryboard and AppDelegate are the standard names for these files, but ViewController is barely descriptive of what the controller will do, and it's not going

to be helpful if the project ends up having more than one view controller. Time to refactor!

Open **ViewController.h** and right-click the class name ViewController in the editor pane. From the options that show up, select **Refactor > Rename...**

In the dialog field, use **SocialViewController** for the name and make sure Rename Related Files is selected. Then click Preview.



A preview window of the operation will come up, but Xcode is usually very smart about this, so go ahead and click **Save** to proceed with the rename.

You might be prompted to create a snapshot at this point. You can decide to enable or disable – it won't matter either way – but a snapshot allows you to revert back to a previous state if something goes wrong with your project.

Just to double-check that the refactor went well, run the project by clicking the Run button at the top-left of the Xcode window, or by using the keyboard shortcut Command-R.

No errors? Awesome!

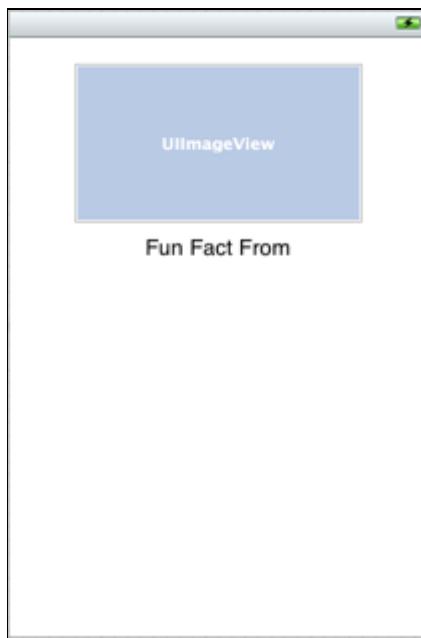
Building the user interface

Now select **MainStoryboard.storyboard** to begin laying out the UI for your app. Drag an image view from the Object Library onto the main view. Make it 220 points wide by 122 points high, and position it at 50 points in X and 20 points in Y. Then, in the Attributes inspector, select the Clip Subviews checkbox and set the View Mode to Aspect Fit.

After the image view comes the “Fun Fact From” label. Drag a label from the Object Library into the view; make it 112 points wide and 21 points high and position it at 104 points in X and 158 points in Y.

Finally, in the Attributes inspector, change the text to “Fun Fact From” and select the Hidden checkbox (it makes no sense to show this label before a random fact has been generated).

This is how your interface should look so far:



Below the label will be a view that contains two sub-view labels for the author name and Twitter handle. From the Object Library, drag a View object into the Social View Controller's view. Make the view 264 points wide and 39 points high and position it at 28 points in X and 187 points in Y.

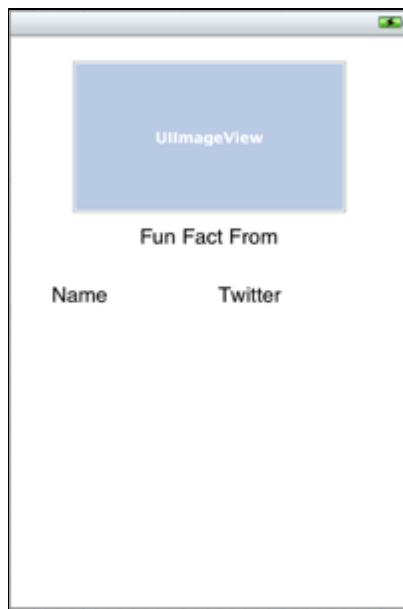
Inside this view will go two labels, so begin by dragging one onto the view. Make sure the label is inside the view – if necessary, use the Document Outline to check whether the label is a sub-view of this small view you just created.



Select your newly-placed label; change its size to 120 points in width and 26 points in height and position it at 5 points in X and 7 points in Y. In the Attributes inspector, change the text to **Name**.

Place another label next to the one you just customized (making sure it's inside the small view); make it 120 points wide and 26 points high, position it at 139 points in X and 7 points in Y and change its text to **Twitter**.

Here's what the view should look like now:



You've made it halfway – hang in there and you'll be done with the storyboard soon.

le click le storyboard...



Next up is the text view that will contain the author's fun fact. Drag a text view from the Object Library into the view, give it a width of 264 points and a height of 125 points and position it at 28 in X and 247 in Y.

So that you can see the text view clearly, give it a background color of your preference (this will change later, so don't be too choosy). Remove any text from the text view, set the alignment to center (the middle button), make it non-editable by deselecting the Editable checkbox in the Attributes inspector, and add the following text: **Shake to get a Fun Fact from a random iOS 6 by Tutorials author or editor!**

Now, for the final portion of the UI, drag four buttons into place and give them the following sizes, positions, titles and tags (the tag can be found in the Attributes inspector, along with the button title):

Action Button:

- 55 points wide, 44 points high
- 28 points in the X position, 396 points in the Y position
- **A** for the title
- 0 (zero) for the tag (this tag is not going to be used anyway)

Tweet Button:

- 55 points wide, 44 points high
- 97 points in the X position, 396 points in the Y position
- **T** for title
- 2 for the tag

Facebook Button:

- 55 points wide, 44 points high
- 168 points in the X position, 396 points in the Y position
- **F** for title
- 0 (zero) for the tag

Sina Weibo Button:

- 55 points wide, 44 points high
- 237 points in the X position, 396 points in the Y position
- **S** for title
- 1 for the tag

It should be pretty obvious what the Tweet, Facebook and Sina Weibo buttons are going to do – they will bring up share sheets for their respective social media platforms.

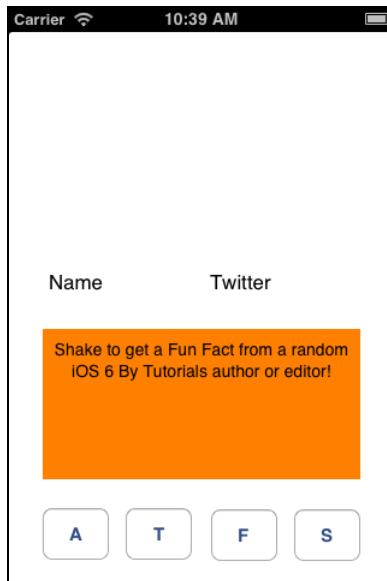
The Action button will bring up the activity view. At first, the options on the activity view will be limited and will depend upon how you have your device set up.

You may, for example, see activity view options for sharing via Facebook and Twitter, and this might seem redundant – after all, FunFacts already has buttons for sharing through those services!

But later on in the chapter, you will expand the activity view to include a cool custom option. Also, in your app you might not want to have to make individual buttons for each sharing option like this – you might find it more handy just to add the single activity view button. So bear with me as we get things working!

Yay, the UI layout is done for now. You'll come back to Interface Builder in a bit to connect the actions and outlets and to give it a coat of paint (some awesome design assets provided to you by Vicki Wenderlich).

Compile and run to check out your view controller, and you'll see the following:



Creating and connecting outlets

Back in **SocialViewController.m**, take a look at the class extension at the top of the file, right after the `#import` – it should look like:

```
@interface SocialViewController ()  
  
@end
```

With the newer versions of Xcode and the improvements to the Objective-C language and LLVM compiler, you no longer need to declare your `IBOutlet`s and `IBAction`s in the header file of your classes. You can now do so inside the class extensions in the implementation file and avoid unnecessary public variables.

Note: For an in-depth look at the improvements to Objective-C and the LLVM compiler, check out Chapter 2, “Programming in Modern Objective-C.”

Add the following code inside the class extension:

```
@property (weak, nonatomic) IBOutlet UIButton *actionButton;  
@property (weak, nonatomic)  
    IBOutlet UIView *authorBackgroundView;  
@property (weak, nonatomic)  
    IBOutlet UIImageView *authorImageView;  
@property (weak, nonatomic) IBOutlet UIButton *facebookButton;  
@property (weak, nonatomic) IBOutlet UITextView *factTextView;  
@property (weak, nonatomic) IBOutlet UILabel *factTitleLabel;
```

```
@property (weak, nonatomic) IBOutlet UILabel *nameLabel;
@property (weak, nonatomic) IBOutlet UIButton *twitterButton;
@property (weak, nonatomic) IBOutlet UILabel *twitterLabel;
@property (weak, nonatomic) IBOutlet UIButton *weiboButton;

- (IBAction)actionTapped;
- (IBAction)socialTapped:(id)sender;
```

There is an outlet for pretty much every element in the UI, including the view containing the author's name and Twitter username labels, as well as each of the four buttons.

An outlet for the container view and each of the buttons is declared so the UI can be customized later on in this chapter.

After the property declarations, there are two `IBActions`, one for when the Action button is tapped and one for when any of the three social buttons are tapped (the Facebook, Twitter, or Sina Weibo buttons).

You'll notice that `socialTapped:` receives an `id` parameter. This will be used to retrieve the tag of the button that sent the action and then bring up the appropriate share sheet.

So that Xcode doesn't complain about an incomplete implementation, go ahead and add stubs for the methods:

```
- (IBAction)actionTapped
{
}

- (IBAction)socialTapped:(id)sender
{
}
```

Note: Another awesome new feature of Objective-C is automatic synthesis for your properties. For more information on this, please see Chapter 2, "Programming in Modern Objective-C."

Time to make the connections inside Interface Builder. Switch back to **MainStoryboard.storyboard** and connect all of the outlets to their corresponding objects. Don't forget about the view containing the author name and Twitter username.

After connecting all the outlets, you need to hook up the actions to each of the four buttons. Connect the Action button's Touch Up Inside event to `actionTapped`, and the Touch Up Inside event of each of the three social buttons (Facebook, Twitter and Sina Weibo) to `socialTapped`.

Make sure everything is working properly by running your project. Try tapping all four buttons to ensure that you don't receive any errors or exceptions due to faulty connections.

Adding the Facts

Now it's time to add the fun facts into the app!

When a user shakes the device, the app will randomly acquire the information corresponding to an author from a property list, and any of the author images from inside the application bundle.

So let's add the property list with the facts and the images for each of the authors.

There should be a **Resources.zip** file among the resources for this chapter. If you extract it, you should see a file named `FactsList.plist` and a folder containing author images (plus another folder with images for the app UI, but we'll get to that later).

Drag all of these files into your Xcode project. Make sure that **Copy items into destination group's folder (if needed) is checked**, the **FunFacts** target is checked, and click **Finish**.

Then open **FactsList.plist**. This is a property list I created to store the fun facts for each author or editor. Take a look through and get a feel for how the structure works, because you're about to write some code to pull it into the app!

| Key | Type | Value |
|---------|------------|--|
| Root | Array | (9 items) |
| Item 0 | Dictionary | (4 items) |
| Item 1 | Dictionary | (4 items) |
| Item 2 | Dictionary | (4 items) |
| Item 3 | Dictionary | (4 items) |
| Item 4 | Dictionary | (4 items) |
| Item 5 | Dictionary | (4 items) |
| Item 6 | Dictionary | (4 items) |
| name | String | Felipe Laso |
| twitter | String | @airjordan12345 |
| image | String | FelipeLaso.jpg |
| facts | Array | (3 items) |
| Item 0 | String | @airjordan12345 loves cooking, reading programming books, basketball and video games |
| Item 1 | String | @airjordan12345's favorite color is pink (real men wear pink! haha) |
| Item 2 | String | @airjordan12345 lived in Miami for 2 years. Here he learned english and got his first programming book |
| Item 7 | Dictionary | (4 items) |
| Item 8 | Dictionary | (4 items) |

The root object is an array (so it will have to be loaded into an `NSArray`). Each of its items is a dictionary with **name**, **twitter**, **image** and **facts** as the keys, and **facts** is an array containing three facts for each author.

Right now, there is no variable or property to store the contents of the property list. To fix this, add the following property declaration inside the class extension of **SocialViewController.m**:

```
@property (strong, nonatomic) NSArray *authorsArray;
```

A good practice is to “lazy load” whatever you need into memory. This just means not creating instances of objects or loading items into memory until they are required (it’s not an ironclad law and it can, and has to be, broken in certain situations). A custom getter for `authorsArray` will help achieve this, so add the following method to **SocialViewController.m**:

```
- (NSArray *)authorsArray
{
    if (!_authorsArray)
    {
        NSString *authorsArrayPath = [[NSBundle mainBundle]
            pathForResource:@"FactsList" ofType:@"plist"];
        self.authorsArray = [NSArray
            arrayWithContentsOfFile:authorsArrayPath];
    }
    return _authorsArray;
}
```

All this code does is check if `_authorsArray` (the instance variable, not the property, since this code is inside the property getter) is not `nil`, otherwise the path to the `FactsList.plist` file is acquired and its contents are loaded into `_authorsArray` using iOS’s built-in property list deserialization.

To verify that the custom getter for the property is working, add the following code to the end of `viewDidLoad`:

```
NSLog(@"Authors Array = %@", self.authorsArray);
```

Run the app and check the console output. You should see the array of authors and the content of each dictionary corresponding to an author. If you get no errors, go ahead and delete the above line of code.

Another good practice you’re going to use is writing `#define` statements for strings when accessing the contents of a dictionary by key. The advantage of using a defined key is that you can easily change it once, and immediately have it updated everywhere it’s used.

I’m sure you already do these good practices religiously, being a reader of raywenderlich.com. ☺



Either way, this is what you're going to do for Fun Facts!

At the top of **SocialViewController.m**, below the `#import`, add the following lines:

```
#define AuthorFactsKey      @"facts"
#define AuthorImageKey        @"image"
#define AuthorNameKey         @"name"
#define AuthorTwitterKey      @"twitter"
```

Now, accessing the contents of an author's dictionary will be a breeze, and your code will be protected if the structure of the property list were to change.

So far, so good, right? Let's recap what you've done so far:

You created the Fun Facts UI in Interface Builder, declared properties and actions for those UI elements, made a custom getter for reading the array of authors from the property list, and defined the keys you'll use to look up items in dictionaries from the property list.

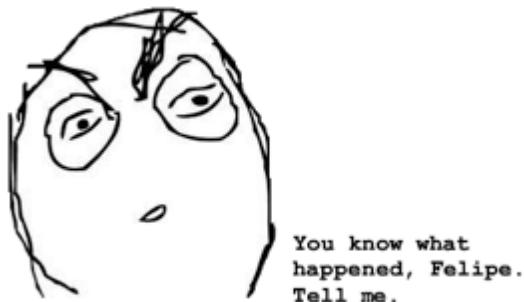
It sounds like it's time to get to the fun stuff!

Shake it like a Polaroid picture

First on the list is detecting a shake so that the displayed fact can be switched. To do this, a view controller must override `motionEnded:withEvent:` and check for the `UIEventSubtypeMotionShake` event. Add the following code to override the method and check for the required event:

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:
(UIEvent *)event
{
    if (motion == UIEventSubtypeMotionShake)
    {
        NSLog(@"SHAKE SHAKE SHAKE!");
    }
}
```

Build and run the app and shake the device (or if using the Simulator, go to Hardware > Shake Gesture in the menu bar). Did you see the expected output in the console? What happened?



Nothing happened, d'oh!

This is because in order for a view controller to detect motion events and gestures, it needs to tell the app that it can become the first responder, and then actually become the first responder when it enters the foreground.

Add the following method definitions so the view controller becomes the first responder and properly detects shake events:

```
- (BOOL)canBecomeFirstResponder
{
    return YES;
}

- (void)viewDidAppear:(BOOL)animated
{
    [self becomeFirstResponder];
}
```

`canBecomeFirstResponder` returns `YES` so the app knows that Social View Controller instances can become the first responder. Then the view controller becomes the first responder whenever it comes into view.

Run the app again and try the shake gesture one more time. Ahh, that's more like it! You should see the following output in the console:

```
SHAKE SHAKE SHAKE!
```

Perfect, you're making good progress! Remove the code to log to the console when a shake occurs, and replace it with the following:

```
// 1
```

```
NSUInteger authorRandSize = self.authorsArray.count;
NSUInteger authorRandomIndex =
    (arc4random() % ((unsigned)authorRandSize));

// 2
NSDictionary *authorDictionary =
    self.authorsArray[authorRandomIndex];

// 3
NSArray *facts = authorDictionary[AuthorFactsKey];
NSString *image = authorDictionary[AuthorImageKey];
NSString *name = authorDictionary[AuthorNameKey];
NSString *twitter = authorDictionary[AuthorTwitterKey];

// 4
NSUInteger factsRandSize = facts.count;
NSUInteger factsRandomIndex =
    (arc4random() % ((unsigned)factsRandSize));

// 5
self.factTitleLabel.hidden = NO;
self.factTextView.text = facts[factsRandomIndex];
self.nameLabel.text = name;
self.twitterLabel.text = twitter;
self.authorImageView.image = [UIImage imageNamed:image];
```

There's a fair bit of code here, but it's quite simple, so let's step through it.

1. First an `NSUInteger` called `authorRandSize` is declared and initialized with the count of the author's array. Then, a random index is generated between 0 and `authorRandSize` which you'll use to choose a random author.

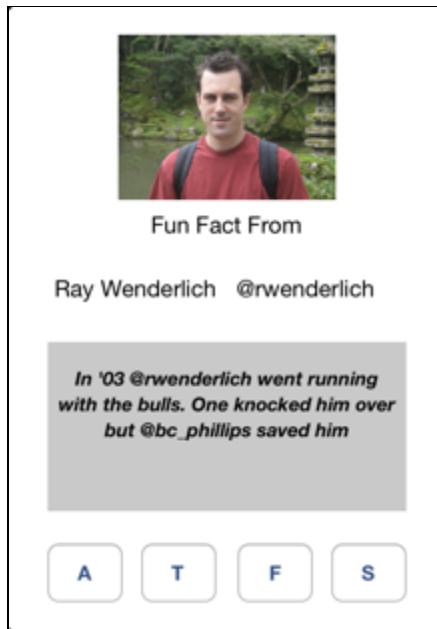
Note: The random number is generated using `arc4random`, a random number-generating function that is an improvement on the traditional C `rand()`.

2. Using the random index for an author in the array, the dictionary corresponding to the author is retrieved from the `self.authorsArray` property and stored in an `NSDictionary` called `authorDictionary`.
3. Using the #defines you set up earlier, pulls out the information of about the author from the dictionary. These variables are now ready for you to use inside your interface.
4. But before doing so, another random index between 0-3 has to be generated to retrieve a fact about the current author.

Note: The code to access an array's specific index and the author's dictionary values was written using the new literal syntax in Objective-C. For more information, see Chapter 2, "Programming in Modern Objective C."

- Finally, the fact title label is unhidden and the appropriate UI elements are populated with the randomly-acquired author data.

That's it! Build and run the app one more time, shake the device and watch as your UI comes alive with a cool, funny and random fact from one of our authors or editors.



The app is coming along nicely. All that remains is making the buttons work, creating a custom activity for the activity view, and giving it a great-looking UI.

From action to activity

Note: If you jumped ahead to this section, extract **FunFactsStarter.zip** from this chapter's resources and open the project in Xcode. At this point, the project just has a basic user interface – take a look through and make sure you understand everything. From this point on, you'll integrate the Social Framework into the app!

The next step is to display the new `UIActivityViewController` when the user taps the action button. To do this, add these lines of code inside `actionTapped`:

```
NSString *initialTextString = [NSString stringWithFormat:@"Fun  
Fact: %@", self.factTextView.text];  
UIActivityViewController *activityViewController =  
[[UIActivityViewController alloc]  
initWithActivityItems:@[initialTextString]  
applicationActivities:nil];  
[self presentViewController:activityViewController animated:YES  
completion:nil];
```

As you can see, displaying a `UIActivityViewController` is incredibly simple! You just allocate it and call `initWithActivityItems`, and then present it modally. Activity items can be either strings or images – here you just pass a single string.

Build and run the project, tap the Action button, and check out the results:



Pretty cool, eh? With a single line of code you can allow users to share something of interest in a variety of manners.

Note: Your options may be different than those shown here. The activity view is context sensitive, displaying icons based on the services available and the accounts set up on the device. If you want to see more options, go to Settings and configure Twitter, Facebook, etc.

Great, the Action button is working, but you don't want to share the default caption inside the text view, do you? Adding a simple Boolean flag to check whether or not the device has been shaken can help address this use case.

Back in the class extension for **SocialViewController.m**, add this property declaration:

```
@property (assign, nonatomic) BOOL deviceWasShaken;
```

Then, inside `motionEnded:withEvent:` add this line inside the `if` statement to indicate that the user has performed a shake gesture:

```
if (motion == UIEventSubtypeMotionShake)
{
    self.deviceWasShaken = YES;
    //...
}
```

Finally, replace `actionTapped` with the following:

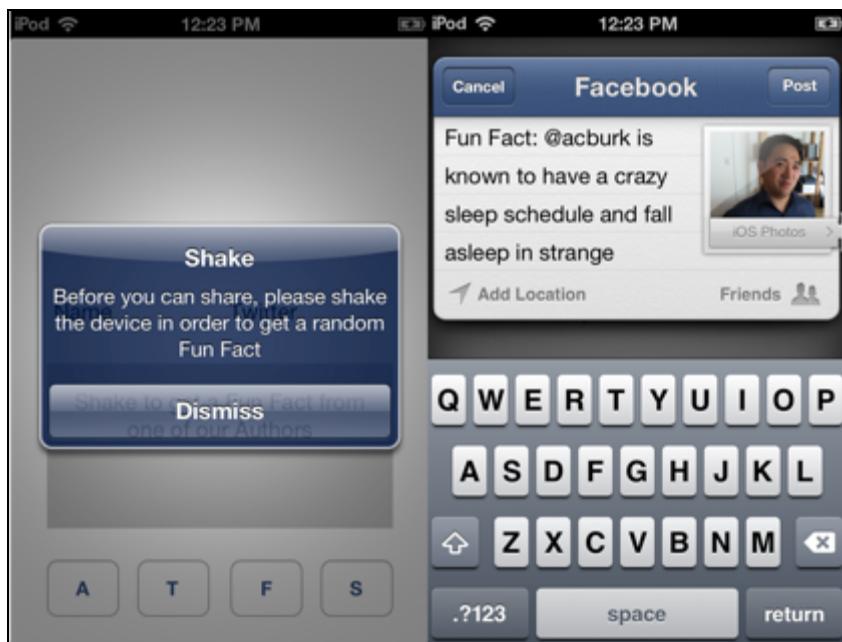
```
- (IBAction)actionTapped
{
    if (self.deviceWasShaken)
    {
        NSString *initialTextString = [NSString
            stringWithFormat:@"Fun Fact: %@", self.factTextView.text];
        UIActivityViewController *activityViewController =
            [[UIActivityViewController alloc]
                initWithActivityItems:@[self.authorImageView.image,
                    initialTextString] applicationActivities:nil];
        [self presentViewController:activityViewController
            animated:YES completion:nil];
    }
    else
    {
        UIAlertView *alertView = [[UIAlertView alloc]
            initWithTitle:@"Shake"
            message:@"Before you can share, please shake the
                device in order to get a random Fun Fact"
            delegate:nil
            cancelButtonTitle:@"Dismiss"
            otherButtonTitles:nil];
        [alertView show];
    }
}
```

This checks to see if the user has shook the device, and if not presents an error.

Also note that the code to present the activity view has slightly changed. Now in addition to sending some text to the `UIActivityViewController`, it sends the image

of the author as well! This is as simple as adding the image to the array of activity items.

Build and run to try it out:



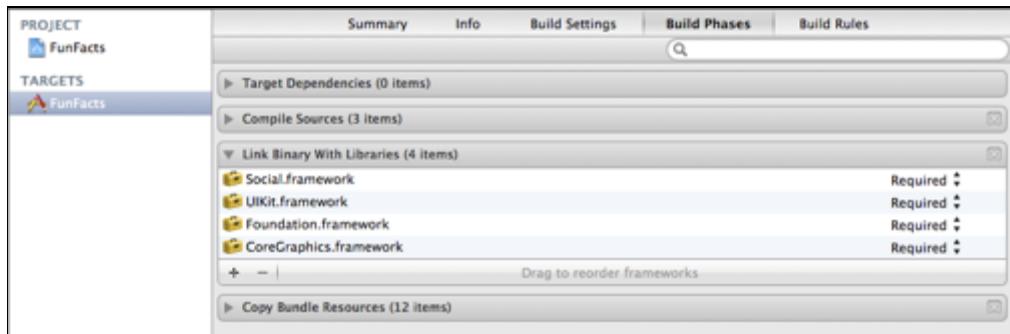
Another high-five for making such great progress so far!

This is great, but what if you want to avoid this intermediary dialog and share to a *specific* social framework in your app? We'll cover that next by showing you how to use the new `SILComposeViewController`!

Button up and get social

Before adding the code for sharing, you need to add the Social Framework into your project (believe it or not the `UIActivityViewController` does not require the Social Framework).

In the Project Navigator, select the **FunFacts** project and switch over to the Build Phases tab. Expand the Link Binary With Libraries section, click the (+) button and select **Social.Framework**. This is what your result should look like:



Now the Social Framework's headers need to be imported and, since this is a library that you will not be making changes to, it's a good idea to import them inside the precompiled headers file.

In the Project Navigator, find **FunFacts-Prefix.pch** (it's inside the Supporting Files group) and add the imports to the Social Framework, as shown below:

```
#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
    #import <Social/Social.h>
#endif
```

Great, now switch to **SocialViewController.m** and add the enumerator declaration as follows (you can add this below the `#define` lines):

```
typedef enum SocialButtonTags
{
    SocialButtonTagFacebook,
    SocialButtonTagSinaWeibo,
    SocialButtonTagTwitter
} SocialButtonTags;
```

If you remember earlier in the chapter, when customizing the UI elements inside MainStoryboard, you added tags to the three social buttons.

This enumerator is used so you can easily check for the tags of the buttons without having to use numbers that may not make much sense or provide clarity as to which button you're working with.

This project is now ready to get social!

Let's add the code for `socialTapped:`. Here is the code you need to put inside the method, along with comments that will be used to explain everything that goes on:

```
- (IBAction)socialTapped:(id)sender
{
    if (self.deviceWasShaken)
```

```
{  
    // 1  
    NSString *serviceType = @"";  
  
    // 2  
    switch (((UIButton *)sender).tag)  
    {  
        case SocialButtonTagFacebook:  
            serviceType = SLServiceTypeFacebook;  
            break;  
  
        case SocialButtonTagSinaWeibo:  
            serviceType = SLServiceTypeSinaWeibo;  
            break;  
  
        case SocialButtonTagTwitter:  
            serviceType = SLServiceTypeTwitter;  
            break;  
  
        default:  
            break;  
    }  
  
    // 3  
    if (![SLComposeViewController  
        isAvailableForServiceType:serviceType])  
    {  
        // 4  
        [self  
            showUnavailableAlertForServiceType:serviceType];  
    }  
    else  
    {  
        // 5  
        SLComposeViewController *composeViewController =  
            [SLComposeViewController  
                composeViewControllerForServiceType:serviceType];  
        [composeViewController  
            addImage:self.authorImageView.image];  
        NSString *initialTextString = [NSString  
            stringWithFormat:@"Fun Fact: %@",  
            self.factTextView.text];  
        [composeViewController  
            setInitialText:initialTextString];  
        [self presentViewController:composeViewController
```

```
        animated:YES completion:nil];
    }
}
else
{
    // 6
    UIAlertView *alertView = [[UIAlertView alloc]
        initWithTitle:@"Shake"
        message:@"Before you can share, please shake the device
in order to get a random Fun Fact"
        delegate:nil
        cancelButtonTitle:@"Dismiss"
        otherButtonTitles:nil];
    [alertView show];
}
}
```

It's a good amount of code, but most of it is pretty simple.

First there's a big if-else statement that checks whether or not the device has been shaken. If it has, then the sharing code begins, otherwise the method goes to the else portion of the statement and displays an alert view similar to the one in `actionTapped`.

Here's a section-by-section breakdown of the method:

1. Creates an empty string to store the service type to be used for sharing. This string will be filled out shortly, when a check is made on the button's tag to see which social network the user has opted to use.
2. Creates a switch statement based on the button's tag. Notice that the sender object is cast to a `UIButton` in order to retrieve its tag. This is perfectly legal, since you know that the sender is definitely a button. Depending on the button's tag, the appropriate service type is assigned to the string declared earlier. The social framework provides the service type constants to you, just so you know where they came from. Notice also how using an enumerator is much more verbose than simply using 0, 1 and 2 for the tags.
3. With the service type string acquired, the code checks to see if the service is available to the user. The main reason it may not be available is if an account for the service hasn't been set up via the Settings app.
4. If the selected service type is unavailable, the method makes a call to `showUnavailableAlertForServiceType:`. You will add this method soon – for now, just know that all it does is display an alert view with the properly-formatted name for the social service requested.
5. If the selected service is available, then the method creates an instance of `SLComposeViewController` and initializes it with the service type requested. The

author's image and fact are added to the social composer, and it's presented to the user via the standard call to `presentViewController:animated:completion:`.

- Finally, as previously mentioned, if the user has not shaken the device yet, then the code presents an alert to indicate this.

Before you run the project, add `showUnavailableAlertForServiceType:` as follows:

```
- (void)showUnavailableAlertForServiceType:  
    (NSString *)serviceType  
{  
    NSString *serviceName = @"";  
  
    if (serviceType == SLServiceTypeFacebook)  
    {  
        serviceName = @"Facebook";  
    }  
    else if (serviceType == SLServiceTypeSinaWeibo)  
    {  
        serviceName = @"Sina Weibo";  
    }  
    else if (serviceType == SLServiceTypeTwitter)  
    {  
        serviceName = @"Twitter";  
    }  
  
    UIAlertView *alertView = [[UIAlertView alloc]  
        initWithTitle:@"Account"  
        message:[NSString stringWithFormat:@"Please go to the device  
settings and add a %@ account in order to share through that  
service", serviceName]  
        delegate:nil  
        cancelButtonTitle:@"Dismiss"  
        otherButtonTitles:nil];  
    [alertView show];  
}
```

The method checks what service type was received as a parameter and creates a user-friendly string for the service's name. It then presents an alert view telling the user to add an account in Settings before attempting to share using the selected service.

Run the project again, and test uses different scenarios: without shaking the device, after shaking the device, and with and without accounts to social services inside Settings.



Woohoo, social sharing is fully implemented and working!

But you're not done, ohh no! There are still a few more tricks in store for you, my friend. ☺

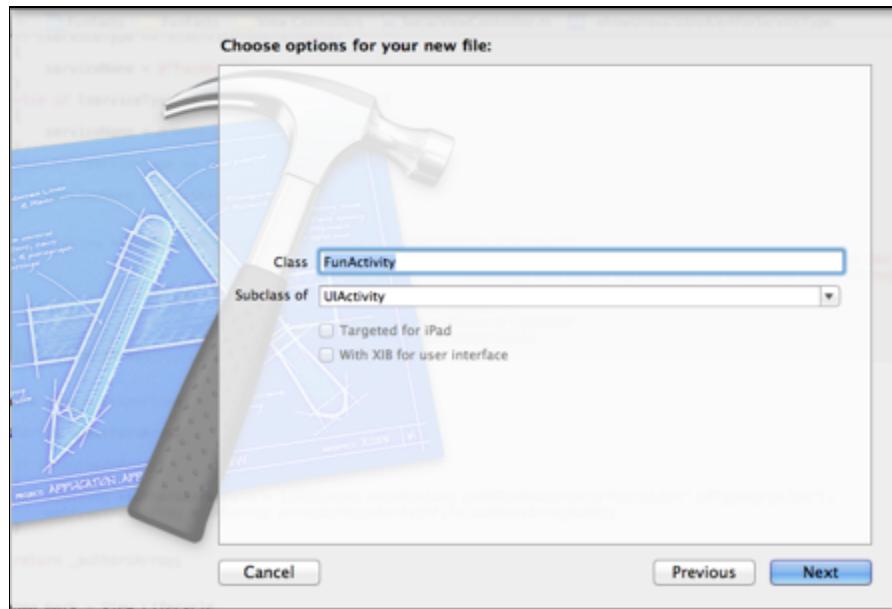
Creating a custom activity

As mentioned before, an activity view has support for custom activity items. What better way to test out custom activities than with some fun facts!

Note: Currently you can only add a custom activity to activity views inside your own app. I.e. there is currently no way to register your an activity that can be called from other apps.

Maybe this will change one day in the future, but it is still useful to be able to register activities within your own app. For example, in this app you will add a custom activity that allows the user to save a fun fact to the Photos library!

To create a custom activity, you need to subclass `UIActivity`. So, right-click the **FunFacts** folder in the Project Navigator and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **FunActivity**, make it a subclass of **UIActivity**, and click **Next** and finally **Create**.



A `UIActivity` subclass needs to override the following methods in order to indicate what items it supports, provide some info when being shown in an activity view, and perform its requested action:

```
// 1
- (UIImage *)activityImage;
// 2
- (NSString *)activityTitle;
// 3
- (NSString *)activityType;
// 4
- (BOOL)canPerformWithActivityItems:(NSArray *)activityItems;
// 5
- (void)prepareWithActivityItems:(NSArray *)activityItems;
// 6
- (void)performActivity;
```

Here's a brief explanation of each method:

1. Indicates the image to be shown for your custom activity inside the activity view.
2. The title to be shown inside the activity view.
3. An identifier (Apple recommends using something similar to your Bundle ID) that identifies your unique activity so you can easily reuse it in other projects or have several custom activities without them conflicting with each other.
4. An array of items that you need to check to see if you support them and if you can perform actions on them.

5. If you do support the items received in `canPerformActivityWithItems:`, then here is where you can prepare those items and store them before performing the action.
6. The user has tapped the activity button corresponding to your custom activity; time to execute the custom activity on the items previously received.

FunActivity will have support for text and images, which means that it will need properties to store a string and image when it's preparing to perform its action. Switch to **FunActivity.m** and add a class extension with the following properties:

```
@interface FunActivity ()  
  
@property (strong, nonatomic) UIImage *authorImage;  
@property (strong, nonatomic) NSString *funFactText;  
  
@end
```

There are just two properties, one for an image corresponding to an author and one for the fact string.

Now on to overriding the necessary methods and giving your custom activity class something to do. Add the following method overrides to **FunActivity.m**:

```
- (UIImage *)activityImage  
{  
    return [UIImage imageNamed:@"activity.png"];  
}  
  
- (NSString *)activityTitle  
{  
    return @"Save Quote To Photos";  
}  
  
- (NSString *)activityType  
{  
    return @"li.iFe.FunFacts.quoteView";  
}
```

The activity image you're using is `activity.png`. If you check out the file in the Project Navigator, you'll notice that it's a black and white image with alpha.

This style is similar to the images used by bar button and tab bar items: it needs to be black and white with transparency or it will not work. Also, the size of the image is the same as a regular iPhone app icon: 57x57 pixels for non-retina displays, and 114x114 pixels for retina displays.

The activity title is **Save Quote To Photos**, and the activity type is my standard Bundle Identifier along with "quoteView". Feel free to use your own Bundle ID for the activity type string.

Now your activity needs to check whether it can work with the items it receives from the activity view. To do so, add the following code to **FunActivity.m**:

```
- (BOOL)canPerformWithActivityItems:(NSArray *)activityItems
{
    for (int i = 0; i < activityItems.count; i++)
    {
        id item = activityItems[i];

        if ([item class] == [UIImage class] || [item
isKindOfClass:[NSString class]])
        {

        }
        else
        {
            return NO;
        }
    }

    return YES;
}
```

This code loops through each of the items in the activity items array and checks whether it's a `UIImage` or `NSString`. In case one of the items is not a string or an image, the method returns `NO`; otherwise if everything is OK at the end of the method it returns `YES` to indicate that it can perform an action with the received items.

Next up is preparing for the activity. Here's the code for that:

```
- (void)prepareWithActivityItems:(NSArray *)activityItems
{
    for (int i = 0; i < activityItems.count; i++)
    {
        id item = activityItems[i];

        if ([item class] == [UIImage class])
        {
            self.authorImage = item;
        }
        else if ([item isKindOfClass:[NSString class]])
        {
```

```
        self.funFactText = item;
    }
}
}
```

This code is very similar to `canPerformWithActivityItems:`, except that this time the method stores the image and text inside the properties previously declared.

Last but not least is `performActivity`, called when the user has requested to perform the activity you are creating. Add this method next:

```
- (void)performActivity
{
    CGSize quoteSize = CGSizeMake(640, 960);
    UIGraphicsBeginImageContext(quoteSize);

    UIView *quoteView = [[UIView alloc]
        initWithFrame:CGRectMake(0, 0, quoteSize.width,
        quoteSize.height)];
    quoteView.backgroundColor = [UIColor blackColor];

    UIImageView *imageView = [[UIImageView alloc]
        initWithImage:self.authorImage];
    imageView.frame = CGRectMake(20, 20, 600, 320);
    imageView.backgroundColor = [UIColor clearColor];
    imageView.contentMode = UIViewContentModeScaleAspectFit;

    [quoteView addSubview:imageView];

    UILabel *factLabel = [[UILabel alloc]
        initWithFrame:CGRectMake(20, 360, 600, 600)];
    factLabel.backgroundColor = [UIColor clearColor];
    factLabel.numberOfLines = 10;
    factLabel.font = [UIFont fontWithName:@"HelveticaNeue-Bold"
        size:44];
    factLabel.textColor = [UIColor whiteColor];
    factLabel.text = self.funFactText;
    factLabel.textAlignment = NSTextAlignmentCenter;

    [quoteView addSubview:factLabel];

    [quoteView.layer
        renderInContext:UIGraphicsGetCurrentContext()];

    UIImage *imageToSave =
        UIGraphicsGetImageFromCurrentImageContext();
```

```
UIGraphicsEndImageContext();

UIImageWriteToSavedPhotosAlbum(imageToSave, nil, nil, nil);
[self activityDidFinish:YES];
}
```

All this code does is create a custom graphics context and then a view with the author image and fact placed nicely inside it. After the view is created, it's rendered onto the context, stored in an image, and saved to the user's photo album. Finally, the activity indicates that it's finished by calling `activityDidFinish:` with `YES` as a parameter.

Note: The code is not too complicated, but it's also beyond the scope of this chapter and book. For further reference, check out our Core Graphics tutorial series: <http://www.raywenderlich.com/2033/core-graphics-101-lines-rectangles-and-gradients>

Before you move on, you need to add an import to the top of the file to satisfy a warning:

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

Awesome! You're almost ready to use your custom activity. But one final change is required in **SocialViewController.m**, so open that file now.

At the top of the file add an import for `FunActivity`:

```
#import "FunActivity.h"
```

Then, change the code in `actionTapped` as shown below:

```
- (IBAction)actionTapped
{
    if (self.deviceWasShaken)
    {
        FunActivity *funActivity = [[FunActivity alloc] init];

        NSString *initialTextString = [NSString
            stringWithFormat:@"Fun Fact: %@", self.factTextView.text];
        UIActivityViewController *activityViewController =
            [[UIActivityViewController alloc]
                initWithActivityItems:@[self.authorImageView.image,
                    initialTextString]
                applicationActivities:@[funActivity]];
```

```
[self presentViewController:activityViewController  
animated:YES completion:nil];  
}  
else  
{  
    // ...  
}
```

Now `actionTapped` creates an instance of `FunActivity` and passes it to `initWithActivityItems:applicationActivities:` as a parameter.

Your custom activity is now in place! Run the project, tap the Action button and select the Fun Facts activity. Then go check the Photo album on your device and check out your custom-drawn fact image:



The iOS Tutorial Team: collect them like trading cards! Maybe there will be a quiz published later on the blog, with prizes. ☺

Styling the UI

You've accomplished the core goals of the chapter, so this is an optional section and just here for fun. If you just wanted to learn the basics of using the Social Framework, pat yourself on the back and jump to the end of this chapter!

But if you want to make the UI in Fun Facts nice and elegant, stick around and I'll show you how.

In Xcode, select the project summary and drag **Images\icon.png** and **Images\icon@2x.png** into the appropriate spots in the App Icons section:



Open **MainStoryboard.storyboard** and make the following changes:

- Change the "Fun Fact From" label's font to Helvetica Neue Bold, 14 points and centered alignment.
- Make the "Name" label's font Helvetica Neue Bold, 14 points.
- Make the "Twitter" label's font Helvetica Neue Bold Italic, 14 points and dark gray for the text color.
- Set the font for the text view to Helvetica Neue Bold Italic, 14 points and centered alignment.
- Delete the title from the Action button and set icon_action.png as the image.
- Delete the title from the Twitter button and set icon_twitter.png as the image.
- Delete the title from the Facebook button and set icon_facebook.png as the image.
- Delete the title from the Sina Weibo button and set icon_sw.png as the image.
- Change the background color of your text view to clear.

That's it for the storyboard. There's only one more bit of code to add inside **SocialViewController.m** and you'll be finished with the chapter. Replace `viewDidLoad` with the following:

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

    self.authorBackgroundView.layer.borderWidth = 1.0f;
    self.authorBackgroundView.layer.borderColor = [[UIColor
        colorWithWhite:0.2 alpha:1.0] CGColor];
    self.authorBackgroundView.layer.cornerRadius = 10.0f;
    self.authorBackgroundView.layer.masksToBounds = YES;

    self.authorImageView.contentMode =
        UIViewContentModeScaleAspectFit;
    self.authorImageView.image = [UIImage
        imageNamed:@"funfacts.png"];
    self.authorImageView.layer.borderWidth = 1.0f;
```

```
self.authorImageView.layer.borderColor = [[UIColor
    colorWithRed:0.2 alpha:1.0] CGColor];
self.authorImageView.layer.shadowColor = [[UIColor
    colorWithRed:0.75 alpha:1.0] CGColor];
self.authorImageView.layer.shadowOffset =
    CGSizeMake(-1.0f, -1.0f);
self.authorImageView.layer.shadowOpacity = 0.5f;

self.factTextView.text = @"Shake to get a Fun Fact from a
random iOS 6 By Tutorials author or editor! ";
self.factTextView.layer.borderWidth = 1.0f;
self.factTextView.layer.borderColor = [[UIColor
    colorWithRed:0.2 alpha:1.0] CGColor];
self.factTextView.layer.cornerRadius = 10.0f;
self.factTextView.layer.masksToBounds = YES;
self.factTextView.layer.shadowColor = [[UIColor
    colorWithRed:0.75 alpha:1.0] CGColor];
self.factTextView.layer.shadowOffset =
    CGSizeMake(-1.0f, -1.0f);
self.factTextView.layer.shadowOpacity = 0.5f;

self.factTitleLabel.hidden = YES;

self.nameLabel.text = self.twitterLabel.text = @"";

[self.actionButton setBackgroundImage:[UIImage
    imageNamed:@"button.png"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 12, 0,12)]
    forState:UIControlStateNormal];
[self.facebookButton setBackgroundImage:[UIImage
    imageNamed:@"button.png"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 12, 0,12)]
    forState:UIControlStateNormal];
[self.twitterButton setBackgroundImage:[UIImage
    imageNamed:@"button.png"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 12, 0,12)]
    forState:UIControlStateNormal];
[self.weiboButton setBackgroundImage:[UIImage
    imageNamed:@"button.png"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 12, 0,12)]
    forState:UIControlStateNormal];

self.view.backgroundColor = [UIColor
    colorWithPatternImage:[UIImage imageNamed:@"bg.png"]];
}
```

This is a lengthy block of code, but it's just some UIKit appearance customizations and Core Graphics magic to set some shadows, borders, corners, etc. If you understand UIKit and Core Graphics, it will make sense.

Also you need this import at the top of the file:

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

Once again, the code itself is beyond the scope of this chapter, but it will give you a cool-looking UI and is an opportunity for some self-directed study of the frameworks used.

In any case, here's the result:



Core Graphics was created by aliens! Yes, this is your author imitating the famous "Aliens" meme, heh heh.

Where to go from here?

Congratulations, you now know how to use the Social Framework in your apps!

You've done a lot in this chapter:

- You've created an app that includes an activity view controller with your own custom activity.
- You integrated the share sheets for all three social networks built into iOS.
- You used the new Objective-C syntax and other new iOS 6 improvements.

- You performed some UI customization with Core Graphics, custom-drew an image for an author's facts, and saved it to the Photos album!

So where do you go from here? How about taking what you've learned and putting it to good use inside your own applications?

You should be able to come up with many other custom activities even cooler than the one you implemented here, and I'm sure your customers and users will appreciate you giving them additional ways to share your app's cool content. In addition, integrating social frameworks might even help you spread word about your app – and earn more money. ☺

If you are interested in learning more about the Social Framework and harnessing its full power, then stick around, because the next chapter will teach you how to access a user's accounts and interact with the APIs for Facebook and Twitter!

Chapter 12: Intermediate Social Framework

By Felipe Laso Marsetti

In the previous chapter, you learned how to share photos and status updates via Twitter, Facebook, and Sina Weibo using the new “share sheet” introduced in iOS 6.

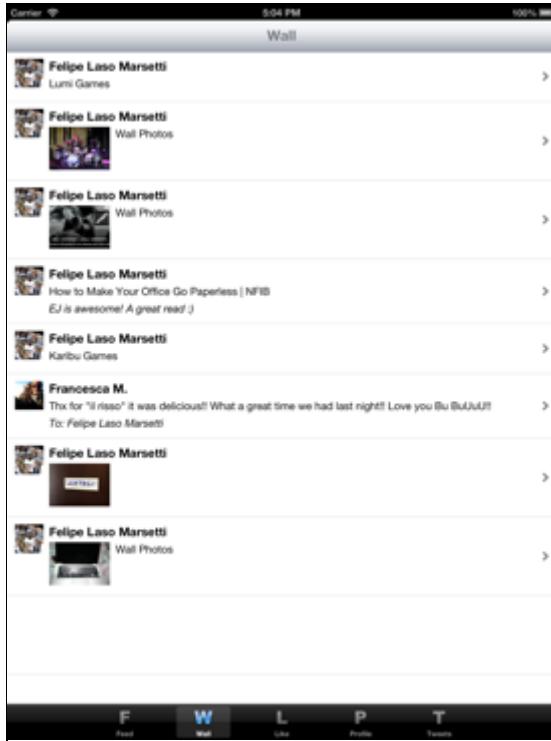
The “share sheet” is extremely handy simple to use, but is also just the tip of the iceberg of what you can do with these social networks. With the new iOS 6 Social Framework, you have full access to each of these social network’s web APIs!

The Social Framework allows you to write apps that do far more than sending tweets or posting to walls. For example, for Facebook you can get a user’s profile, see their friends, get their email address, “like” a post, see their pictures, and more. Anything the API allows (and the user gives permission for), your app can do!

And the best part is the iOS 6 Social Framework takes care of the most annoying details of working with these web APIs for you – things like authorization, dealing with session tokens, making requests, and receiving responses. You can keep your focus on the app-specific logic of what you’re trying to do.

In this chapter, you’ll get hands-on experience with working with the Facebook and Twitter web APIs programatically through the Social Framework. You will build a fun and practical iPad app named iSocial that retrieves your Facebook profile, feed, and wall. It will also let you “like” an item, retrieve your Twitter feed.

Here is a preview of the app you will create in this chapter:



As you can see, there are different tabs to exercise various capabilities of the Facebook and Twitter web APIs. In the process of developing this app, you'll learn how to:

- Use the Accounts framework
- Retreive a Twitter user's feed
- Set up a new Facebook app
- Use the Facebook Graph API documentation
- Access a user's Facebook profile, wall, and "like" an image
- And most importantly, you'll have the tools you need to do anything else these APIs allow!

Ready to get started? Grab a refreshing beverage of your choice, open up Xcode and get ready to learn how to Facebook into your apps without breaking a sweat!

Note: This chapter focuses primarily on Facebook API integration as that is the biggest new thing about iOS 6. It covers the basics of using the Accounts framework and the Twitter API as well, but if you want more in-depth coverage, check out Chapter 13 in *iOS 5 by Tutorials*, "Intermediate Twitter."

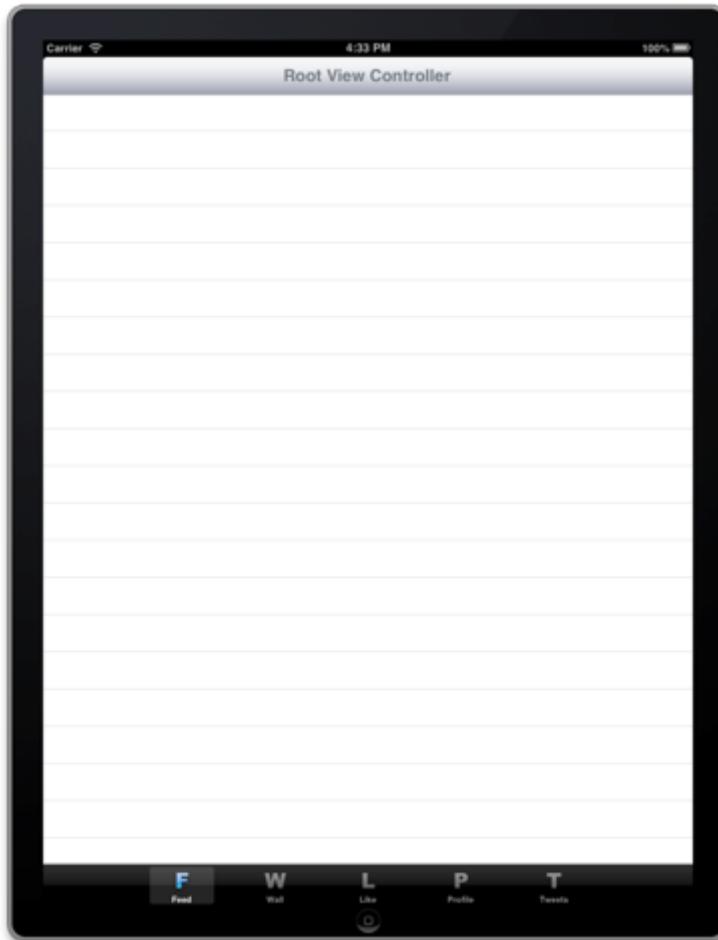
Getting started

The resources for this chapter include a starter project with the basic user interface for the app created for you. There's no social network integration in the project yet though – that's your job in this chapter! Extract the starter ZIP file to a convenient location and open the **iSocial** project in Xcode.

Open **MainStoryboard.storyboard**, and notice how the main UI for the app is a tab bar controller with five tabs – Feed, Wall, Like, Profile, and Tweets. The other view controllers simply implement the views and the navigation hierarchy for these tabs.



Build and run the app to make sure that everything compiles correctly and that there are no errors. If everything works, you should see something like the following:



So the app has a basic UI, but not much more. Now what?

Before you can do anything with the Twitter or Facebook APIs, you first need to get access to the user's account. In iOS 6, users enter their Twitter and Facebook usernames and password in Settings, and then your app can easily log in with these settings using the Accounts framework.

But how does the Accounts framework work? That's what we'll cover next!

The Accounts framework

Every time you use the Social framework to communicate with a social network, you first need to gain access to the user's account using the Accounts framework. Of course for this to work, the user has to have entered their username and password for the social network in Settings.

When you request access to a user's account, a dialog is presented to the user so they have a chance to accept or reject permission. If the user gives your app

permission, you can use the account to access the social network's web API (to view a wall, search Twitter, etc).

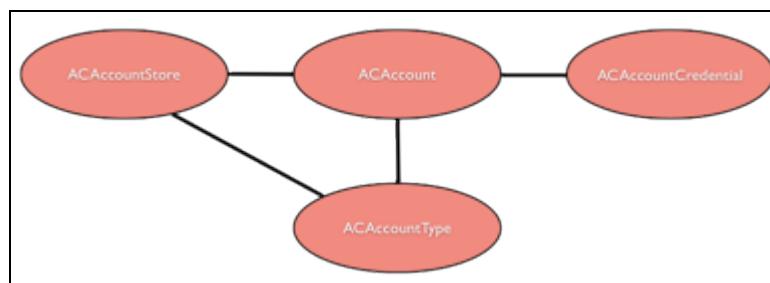
In addition, the Accounts framework will notify you of changes made to the account while the app is running and let you easily retrieve information about the account (like the provider type, identifier, username, and more).

A good understanding of the Accounts framework will prepare you for the upcoming work on iSocial, as well as on your own future projects. You want to understand how those magical lines of code work, right? ☺

The Accounts framework includes the following four classes:

- 1. ACAccountStore:** Primary object to access accounts.
- 2. ACAccount:** Object representing a single account for a single social network.
- 3. ACAccountType:** Object representing the type of social network (Twitter, Facebook, etc.).
- 4. ACAccountCredential:** Authentication credential that can be used for manual authentication.

Here's a graphical representation of their relationship:



As you can see, an `ACAccountStore` has access to a list of `ACAccounts`. Each `ACAccount` has an associated `ACAccountType` (Twitter, Facebook, etc.). Finally, each `ACAccount` has an `ACAccountCredential`.

Next, let's go over each of these objects in more detail.

ACAccountStore

This is the primary object your app will interact with in order to retrieve, add, or delete user's system accounts. As of iOS 6, the Accounts framework supports Twitter, Facebook, and Sina Weibo accounts.

There are two ways to retrieve a user's accounts: by their type, or by a unique identifier. Here's the typical workflow for interacting with the account store and retrieving accounts:

1. Request access for your app to use accounts of a particular type (Facebook, Twitter, or Sina Weibo).

2. If access was granted, then either retrieve all of the accounts by account type, or retrieve a single account by its unique identifier.
3. Use the retrieved account(s) in your app to make authenticated requests to the social network APIs.

You'll add the code for these three steps in iSocial soon, to retrieve a user's Facebook and Twitter accounts.

Note: `ACAccountStore` is not a singleton – you create an instance of an `ACAccountStore` when you need to get access to some accounts.

One important thing to keep in mind is accounts retrieved from a particular instance of an account store are linked to that store. Be careful not to use or modify accounts from one store with a different store, or it will likely result in a crash!

Apple recommends you keep a single instance of `ACAccountStore` throughout your app's lifecycle. You'll be happy to know that you will learn how to do this in iSocial.



It's important to handle the case where a user leaves your app and then updates, deletes, or adds an account to system settings. These changes can affect your app, so it's important to know about them.

Fortunately, the Accounts framework sends a notification when these types of changes occur – the `ACAccountStoreDidChangeNotification`. You can register for this and re-fetch or update accounts as necessary.

ACAccountType

`ACAccountType` is a class that stores information about accounts of a particular type. You don't create an instance of `ACAccountType` yourself, but instead retrieve one via an account store.

An account type has the `accessGranted` Boolean, so you can easily check whether you have access to a particular type of account. This class also has two other

properties: an `accountTypeDescription` string, which is a human-friendly description of the account type; and a unique identifier.

ACAccountCredential

If you want to authenticate users yourself by providing OAuth or OAuth2 session info, then you should use `ACAccountCredential`. You pass in the authenticated user's key and secret, and your app receives a token for you to use.

This class is not going to be covered in iSocial, but it's very handy if your app already uses OAuth or OAuth2 for login and authentication. This way, you can harness the power of the native Social framework without breaking your existing authentication process.

ACAccount

`ACAccount` is the core class of the Accounts framework. An instance of `ACAccount` holds all of the information about a single user's account stored in the system's Account database.

Here's a rundown of the properties and data available to you for an account:

- `accountType`: an `ACAccountType` object corresponding to the account's type.
- `accountDescription`: human-readable description of the account.
- `credential`: an `ACAccountCredential` object corresponding to the account's credentials.
- `identifier`: a unique ID for the account. This can be used to retrieve a specific account from an `ACAccountStore` instance.
- `username`: the account's username.

As you can see, there's nothing overly complex or mysterious about the Accounts framework – it's pretty straightforward and simple to use. If you still have any questions about how it all works, fear not! Once you write a few lines of code to retrieve accounts and interact with an account store, then any doubts you have will surely disappear. ☺

Retrieving accounts for iSocial

Now that you have a good understanding of how the Accounts framework is organized, let's put this knowledge to use in iSocial to retrieve a user's Twitter and Facebook accounts.

To retrieve a user's Twitter account, you don't need to do any setup on the Twitter developer site, so you're good to go.

However for Facebook you have to do a few extra steps. You need to create a Facebook app on Facebook's web site, input a few parameters, and then copy the

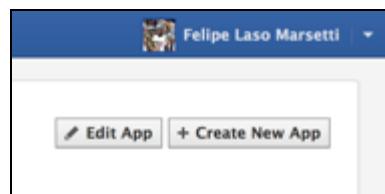
app's ID. You'll need this later to request and retrieve a user's Facebook account details.

Creating a Facebook app

Creating an app on Facebook is fairly simple and straightforward. Go to <http://developers.facebook.com> and log in with your Facebook account (or create one if you don't have one already).



Once logged in, click on the **Apps** button on the right side of the blue bar on top. On the new page, click the button at the top-right corner that reads **Create New App**.



In the dialog that appears, type in your app's name and make sure that App Namespace is empty and the Web Hosting checkbox is unselected. Click **Continue** when you're done, and submit the security check that's required.

 A screenshot of the "Create New App" dialog box. It has a blue header bar with the text "Create New App". Below the header, there are three input fields: "App Name" (containing "Social App" with a "Valid" status indicator), "App Namespace" (containing "Optional"), and "Web Hosting" (with an unchecked checkbox and a link to "Yes, I would like free web hosting provided by Heroku"). At the bottom, there's a link to "By proceeding, you agree to the Facebook Platform Policies" and two buttons: "Continue" and "Cancel".

Now you should have your new Facebook app and be looking at its Basic Settings screen.

The screenshot shows the Facebook Developers App Settings page for 'iSocial App - Basic'. The left sidebar includes links for Settings, Basic, Permissions, Payments, Advanced, App Details, Localize, Open Graph, Roles, Insights, and Related links. The main content area displays the app's basic info, including its App ID, App Secret, and Contact Email. Below this, there's a section titled 'Select how your app integrates with Facebook' with several options listed:

- Website with Facebook Login Log in to my website using Facebook.
- App on Facebook Use my app inside Facebook.com.
- Mobile Web Bookmark my web app on Facebook mobile.
- Native iOS App Publish from my iOS app to Facebook.
- Native Android App Publish from my Android app to Facebook.
- Page Tab Build a custom tab for Facebook Pages.

Things may look pretty daunting at first, but don't worry – you don't need to modify most of these settings at all.

Just scroll down to the Native iOS App section, click to expand it, and copy your app's **Bundle Identifier** exactly as it is set up in Xcode (you can find this information on the Summary tab for your project settings). Click **Save Changes** when you're done.

The screenshot shows the Xcode Project Settings - iOS Application Target panel. It includes the following fields:

- Bundle Identifier: li.Fe.Social
- Version: 1.0
- Build: 1.0
- Devices: iPad
- Deployment Target: 6.0

Below this, the 'Native iOS App' section is expanded, showing:

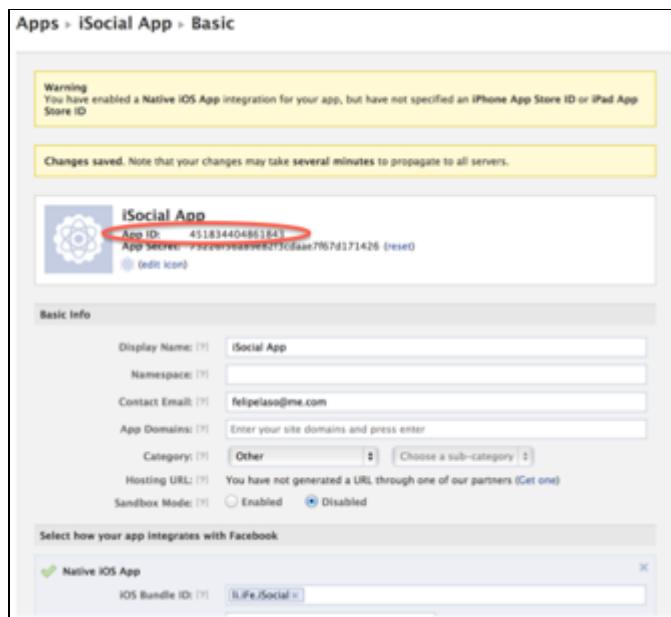
- iOS Bundle ID: li.Fe.Social
- iPhone App Store ID: (empty)
- iPad App Store ID: (empty)
- Configured for iOS SSQ: Enabled Disabled
- iOS Native Deep Linking: Enabled Disabled
- URI Scheme Suffix: (empty) Comma-separated list of URL scheme suffixes
- Native Android App: Publish from my Android app to Facebook.
- Page Tab: Build a custom tab for Facebook Pages.

A 'Save Changes' button is located at the bottom right of the expanded section.

Note: If you don't use the exact same Bundle Identifier as in your app, your iOS app will not be granted permission to access the user's Facebook account.

After you've saved your changes, you may see a few warnings about not having set up an iPhone or iPad App Store ID. Don't worry about these; they are not required for your app to function properly.

Right below the warnings, you will find your Facebook app's info, including the **App ID**. Copy the App ID and save it somewhere handy, because you'll need it in a moment.



And that's it – you're done with the setup on the Facebook developer site!

Accessing a user's Facebook account

Now that you have created a Facebook app and have a Facebook App ID, you're ready to write the code to acquire a user's Facebook account details.

Since you will access the Accounts and Social frameworks from multiple places, it's a good idea to add their header imports to the pre-compiled header file. Open **Supporting Files\iSocial-Prefix.pch** and add the following imports inside the `#ifdef __OBJC__` block:

```
#import <Accounts/Accounts.h>
#import <Social/Social.h>
```

Now go to **AppDelegate.h** and add these property and method declarations:

```
@property (strong, nonatomic) ACAccountStore *accountStore;
@property (strong, nonatomic) ACAccount *facebookAccount;
@property (strong, nonatomic) ACAccount *twitterAccount;

- (void)getFacebookAccount;
```

```
- (void)getTwitterAccount;
```

Here you declare a property for an `ACAccountStore` and two `ACAccount` objects. The user's Twitter and Facebook accounts are separate, hence the need for an `ACAccount` instance for each. You also declare two methods that will initially retrieve access to the user's Facebook and Twitter accounts.

Note: All of this is added inside the app delegate, because several view controllers will need access to either the account store or the accounts. It would be pointless to write the same code over and over again in each view controller, wouldn't it? You're keeping things nice and DRY (**Don't Repeat Yourself**). ☺

Also, remember that in the previous section you learned that it's recommended to always work with a single instance of `ACAccountStore`. By having the reference to the account store in the app delegate, you make sure that your account store instance remains active and alive throughout the app's lifecycle.

Switch to **AppDelegate.m** and add this line to the end of `application:didFinishLaunchingWithOptions:` before the `return` statement:

```
self.accountStore = [[ACAccountStore alloc] init];
```

This creates an instance of `ACAccountStore` and stores it in your app delegate's property. This will come in handy when fetching accounts from the account store.

Next add the code for `getFacebookAccount`:

```
- (void)getFacebookAccount
{
    // 1
    ACAccountType *facebookAccountType = [self.accountStore
        accountTypeWithAccountTypeIdentifier:
        ACAccountTypeIdentifierFacebook];

    // 2
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 3
        NSDictionary *facebookOptions = @{
            ACFacebookAppIdKey : @"451834404861843",
            ACFacebookPermissionsKey : @{@"email", @"read_stream",
                @"user_relationships", @"user_website"},
            ACFacebookAudienceKey : ACFacebookAudienceEveryone };
    });
}
```

```
// 4
[self.accountStore
    requestAccessToAccountsWithType:facebookAccountType
    options:facebookOptions completion:^(BOOL granted,
    NSError *error) {
    // 5
    if (granted)
    {
        [self getPublishStream];
    }
    // 6
    else
    {
        // 7
        if (error)
        {
            dispatch_async(dispatch_get_main_queue(), ^{
UIalertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Facebook"
message:@"There was an error retrieving your Facebook account,
make sure you have an account setup in Settings and that access is
granted for iSocial"
delegate:nil
cancelButtonTitle:@"Dismiss"
otherButtonTitles:nil];
                [alertView show];
            });
        }
        // 8
        else
        {
            dispatch_async(dispatch_get_main_queue(), ^{
UIalertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Facebook"
message:@"Access to Facebook was not granted. Please go to the
device settings and allow access for iSocial"
delegate:nil
cancelButtonTitle:@"Dismiss"
otherButtonTitles:nil];
                [alertView show];
            });
        }
    }
}
}];
```

```
}
```

This may seem a bit lengthy, but don't worry, we'll run through the code a section at a time:

1. Get the `ACAccountType` object for the Facebook account type. As mentioned in the theory section, you don't directly instantiate an `ACAccountType` object, but instead retrieve one from an account store instance using `accountTypeWithAccountTypeIdentifier:` and pass in the correct account type identifier constant, which in this case is `ACAccountTypeIdentifierFacebook`.
2. Declare an asynchronous Grand Central Dispatch (GCD) operation on a global concurrent queue. This operation will encapsulate the code to retrieve a user's Facebook account.

Depending on how much work is going on in the app's main thread, the operation to retrieve an account could take several seconds to complete and return the user's account.

To speed things up (and to prevent the UI from locking up), you can take advantage of GCD and perform this task asynchronously.

3. When retrieving Facebook accounts (and Facebook accounts only!), it's necessary to pass in an options dictionary. Among the options are the Facebook app ID (`ACFacebookAppIdKey`), permissions (`ACFacebookPermissionsKey`), and the audience (`ACFacebookAudienceKey`). Let's go over what these mean one by one.

ACFacebookAppIdKey: This is the Facebook App ID that you got by creating your Facebook app on Facebook's web page. You simply pass what you got there as a string. Remember to replace this with the app ID for the app you created on Facebook!

ACFacebookPermissionsKey: When asking for access to a Facebook account, you have to ask for specific permissions for what you want your app to be able to access. For example, there are different permissions to posting to a user's wall or seeing a user's email address. The user will see what your app wants to do in a dialog and can choose whether to accept or deny permission based on what you're asking for.

To see the full list of permissions your app can ask for, check out the Facebook's Graph API documentation located at:

<http://developers.facebook.com/docs/authentication/permissions>

| User and Friends Permissions | | |
|--|--|--|
| You can ask for the following permissions for users and friends in the <code>scope</code> parameter as part of the authentication process. | | |
| User permission | Friends permission | Description |
| <code>user_about_me</code> | <code>friends_about_me</code> | Provides access to the "About Me" section of the profile in the <code>about</code> property |
| <code>user_activities</code> | <code>friends_activities</code> | Provides access to the user's list of activities as the <code>activities</code> connection |
| <code>user_birthday</code> | <code>friends_birthday</code> | Provides access to the birthday with year as the <code>birthday</code> property |
| <code>user_checkins</code> | <code>friends_checkins</code> | Provides read access to the authorized user's check-ins or a friend's check-ins that the user can see. This permission is superseded by <code>user_status</code> for new applications as of March, 2012. |
| <code>user_education_history</code> | <code>friends_education_history</code> | Provides access to education history as the <code>education</code> property |
| <code>user_events</code> | <code>friends_events</code> | Provides access to the list of events the user is attending as the <code>events</code> connection |
| <code>user_groups</code> | <code>friends_groups</code> | Provides access to the list of groups the user is a member of as the <code>groups</code> connection |
| <code>user_hometown</code> | <code>friends_hometown</code> | Provides access to the user's hometown in the <code>hometown</code> property |
| <code>user_interests</code> | <code>friends_interests</code> | Provides access to the user's list of interests as the <code>interests</code> connection |

Generally, your app needs to request at least read permission (like "email") in order to retrieve an account successfully. For iSocial, you'll need access to the following permissions:

- a. email
- b. read_stream
- c. user_relationships
- d. user_website
- e. publish_stream

Permissions are sent as an array of strings, as shown in the code above.

ACFacebookAudienceKey: Finally, for the audience, you specify the group of people that will have access to your posts and updates sent from within your app. The possible values are:

- f. ACFacebookAudienceEveryone
- g. ACFacebookAudienceFriends
- h. ACFacebookAudienceOnlyMe

For this app, you want everyone to be able to see the posts/updates, so you choose ACFacebookAudienceEveryone.

4. With your Facebook options set up, it's time to perform the actual request to access and retrieve the user's system account. To do so, you use `requestAccessToAccountsWithType:options:completion:`, using the `ACAccountType` object created in step #1 and the options dictionary from step #3.

The completion block is the code that will be called when the system has finished requesting and retrieving an account. Note that the code called inside this block

doesn't guarantee access to the account; it simply means that the process is done and you can now respond according to the results.

- Once inside the completion block, you check the granted Boolean to determine whether or not your app was given access to the user's Facebook account. It is possible that, when prompted, the user denied your app access.

If access was granted, then you call a second method to get access to the publish_stream permission. The reason you have to do this in a separate call rather than just listing the publish_stream permission in the list of permissions is due to what appears to be a bug in the SDK at the time of writing this chapter. If you try to list them all in a single call at the time of writing this chapter, you will get the following error: "The Facebook server could not fulfill this access request: The proxied app cannot request write permissions without having been installed previously." However, asking for the permission separately works, so this is a (hopefully temporary) workaround until this issue is fixed.

If the granted Boolean is false, this means you don't have access to the user's Facebook account and so need to handle the situation accordingly.

- One of the reasons for not getting access to the Facebook account is because an error occurred in the process. If that's the case, then an alert view is displayed with an error message.

Note that the alert view code is embedded in a dispatch_async block to the main thread. This is because access to a user's account is performed in a background thread. All UI events and calls need to be done inside the main thread though, so you need to add this to make sure this code runs on the main thread.

- The other reason for not getting access to a user's Facebook account is because the user simply denied you access. Once again, an alert view call is embedded inside a GCD block.

Next, add the getPublishStream method, which is very similar:

```
- (void)getPublishStream {

    // 1
    ACAccountType *facebookAccountType = [self.accountStore
accountTypeWithAccountTypeIdentifier:
ACAccountTypeIdentifierFacebook];

    // 2
    dispatch_async(dispatch_get_global_queue(
DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        // 3
        NSDictionary *facebookOptions = @{
            ACFacebookAppIdKey : @"451834404861843",
            ACFacebookPermissionsKey : @[@"publish_stream"],
```

```
ACFacebookAudienceKey : ACFacebookAudienceEveryone };  
// 4  
[self.accountStore  
requestAccessToAccountsWithType:facebookAccountType  
options:facebookOptions completion:^(BOOL granted,  
NSError *error) {  
// 5  
if (granted)  
{  
    self.facebookAccount = [[self.accountStore  
accountsWithAccountType:facebookAccountType]  
lastObject];  
  
    dispatch_async(dispatch_get_main_queue(), ^{  
        [[NSNotificationCenter defaultCenter]  
postNotificationName:  
AccountFacebookAccountAccessGranted  
object:nil];  
    });  
}  
// 6  
else  
{  
    // 7  
    if (error)  
{  
        dispatch_async(dispatch_get_main_queue(), ^{  
            UIAlertView *alertView = [[UIAlertView  
alloc] initWithTitle:@"Facebook"  
message:@"There was an error retrieving your Facebook account,  
make sure you have an account setup in Settings and that access is  
granted for iSocial"  
delegate:nil  
cancelButtonTitle:@"Dismiss"  
otherButtonTitles:nil];  
            [alertView show];  
        });  
    }  
    // 8  
    else  
{  
        dispatch_async(dispatch_get_main_queue(), ^{  
            UIAlertView *alertView = [[UIAlertView  
alloc] initWithTitle:@"Facebook"  
message:@"Success! Facebook access granted."  
delegate:nil  
cancelButtonTitle:@"OK"  
otherButtonTitles:nil];  
            [alertView show];  
        });  
    }  
}
```

```
message:@"Access to Facebook was not granted. Please go to the
device settings and allow access for iSocial"
delegate:nil
cancelButtonTitle:@"Dismiss"
otherButtonTitles:nil];
[alertView show];
});
}
}
];
});
}

}
```

Again this is only necessary because a bug in the SDK at the time of writing this chapter won't let you request the publish_stream permission at the same time as the other permissions.

Again, don't forget to update your Facebook App ID key appropriately!

The only difference here is upon success, you call accountsWithAccountType: on your ACAccountStore and retrieve the last object of the returned array. Currently only one Facebook account per iOS device can be set up, so acquiring the last object will give you the correct account.

You also post a notification called AccountFacebookAccountAccessGranted to let other classes and controllers know that you've acquired a Facebook account. You will define this constant next!

Go to AppDelegate.h and add these two lines at the top of the file:

```
#define AccountFacebookAccountAccessGranted
@"FacebookAccountAccessGranted"
#define AccountTwitterAccountAccessGranted
@"TwitterAccountAccessGranted"
```

The second notification name will be used in getTwitterAccount, to notify observers when an account is acquired.

Accessing a user's Twitter account

Accessing a user's Twitter account is pretty similar, except for one thing: a user is allowed to have more than one Twitter account configured in Settings. So, when you get the accounts array you could have more than one account available to use.

getTwitterAccount will take care of this, and will ask the user to select the account they want to use for the app. You will store the selected account's identifier in the user defaults and retrieve it whenever the app launches.

Add this code at the top of **AppDelegate.m**, below the `#import` line:

```
#define AccountTwitterSelectedAccountIdentifier  
@"TwitterAccountSelectedAccountIdentifier"  
  
@interface AppDelegate () <UIAlertViewDelegate>  
  
@end
```

This declares that the app delegate adheres to the `UIAlertViewDelegate` protocol, and defines a constant you'll use to store the Twitter account's identifier in the user defaults.

Now add the following code to implement `getTwitterAccount`:

```
- (void)getTwitterAccount  
{  
    ACAccountType *twitterAccountType = [self.accountStore  
    accountTypeWithAccountTypeIdentifier:  
    ACAccountTypeIdentifierTwitter];  
  
    dispatch_async(dispatch_get_global_queue(  
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{  
        [self.accountStore  
        requestAccessToAccountsWithType:twitterAccountType  
        options:nil completion:^(BOOL granted, NSError *error)  
        {  
            if (granted)  
            {  
                // 1  
                NSArray *twitterAccounts = [self.accountStore  
                accountsWithAccountType:twitterAccountType];  
  
                // 2  
                NSString *twitterAccountIdentifier =  
                [[NSUserDefaults standardUserDefaults]  
                objectForKey:  
                AccountTwitterSelectedAccountIdentifier];  
                self.twitterAccount = [self.accountStore  
                accountWithIdentifier:  
                twitterAccountIdentifier];  
  
                // 3  
                if (self.twitterAccount)  
                {  
                    dispatch_async(dispatch_get_main_queue(), ^{  
                }  
            }  
        }  
    }  
}
```

```
[ [NSNotificationCenter defaultCenter]
postNotificationName:
    AccountTwitterAccountAccessGranted
object:nil];
} );
}
else
{
// 4
[ [NSUserDefaults standardUserDefaults]
removeObjectForKey:
    AccountTwitterSelectedAccountIdentifier];
[ [NSUserDefaults standardUserDefaults]
synchronize];

// 5
if (twitterAccounts.count > 1)
{
UIAlertViewController *alertView = [[UIAlertViewController alloc]
initWithTitle:@"Twitter"
message:@"Select one of your Twitter Accounts"
delegate:self
cancelButtonTitle:@"Cancel"
otherButtonTitles:nil];

for (ACAccount *account in
    twitterAccounts)
{
    [alertView addButtonWithTitle:
        account.accountDescription];
}

dispatch_async(
    dispatch_get_main_queue(), ^{
        [alertView show];
    });
}
// 6
else
{
    self.twitterAccount =
[twitterAccounts lastObject];

dispatch_async(
    dispatch_get_main_queue(), ^{

```

```
[ [NSNotificationCenter
    defaultCenter]
postNotificationName:
    AccountTwitterAccountAccessGranted
object:nil];
} );
}
}
else
{
    if (error)
    {
        dispatch_async(dispatch_get_main_queue(), ^{
UIalertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Twitter"
message:@"There was an error retrieving your Twitter account, make
sure you have an account setup in Settings and that access is
granted for iSocial"
delegate:nil
cancelButtonTitle:@"Dismiss"
otherButtonTitles:nil];
[alertView show];
});
}
else
{
    dispatch_async(dispatch_get_main_queue(), ^{
UIalertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Twitter"
message:@"Access to Twitter was not granted. Please go to the
device settings and allow access for iSocial"
delegate:nil
cancelButtonTitle:@"Dimiss"
otherButtonTitles:nil];
[alertView show];
});
}
}
];
} );
}
```

As I mentioned, most of the code to retrieve a Twitter account is identical to what you did for Facebook, except there is no options dictionary passed in to `requestAccessToAccountsWithType:options:completion:`, and the code to handle the

retrieval of the account (should you be granted access) handles the multiple accounts case.

Let's take a look at the differences in gaining access to Twitter accounts:

1. You get the array of Twitter accounts by calling `accountsWithAccountType:` on the account store.
2. You get the string identifier of the user's preferred Twitter account that's stored in `NSUserDefaults`. Then you get the `ACAccount` object for the Twitter account corresponding to the retrieved identifier. This will obviously result in `nil` if it's the first run of the app, or the first time getting the Twitter accounts (since the identifier string will be `nil` as well).
3. If `twitterAccount` is not `nil`, then it means you successfully retrieved the user's preferred Twitter account. If that's the case, then you post a notification on the main thread using GCD.
4. If `twitterAccount` is `nil`, then you have to handle things differently, starting with clearing any possible values stored in user defaults for the user's Twitter account identifier.
5. Your user may have more than one Twitter account stored in the device's settings. If that's the case, then you will show an alert view with all of her accounts so that the user can choose which one to use.

To do this, you create an alert view, set the app delegate as the alert view delegate, and loop through each of the accounts in the array to add their description strings as buttons.

You then show this alert view (again, making sure to call it from inside the main thread using GCD) and your user will choose the account s/he prefers.

6. If your user has only one Twitter account, then you simply store it in the `twitterAccount` property and post a notification to let the other classes know you've acquired the Twitter account.

To complete the process of getting Twitter accounts, you need to add an implementation for the alert view delegate's `alertView:clickedButtonAtIndex:`. Add the method as shown below:

```
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex != alertView.cancelButtonIndex)
    {
        ACAccountType *twitterAccountType = [self.accountStore
            accountTypeWithAccountTypeIdentifier:
            ACAccountTypeIdentifierTwitter];
        NSArray *twitterAccounts = [self.accountStore
            accountsWithAccountType:twitterAccountType];
```

```
self.twitterAccount =  
    twitterAccounts[(buttonIndex - 1)];  
  
[ [NSUserDefaults standardUserDefaults]  
    setObject:self.twitterAccount.identifier  
    forKey:AccountTwitterSelectedAccountIdentifier];  
[ [NSUserDefaults standardUserDefaults] synchronize];  
  
[ [NSNotificationCenter defaultCenter]  
    postNotificationName:  
        AccountTwitterAccountAccessGranted object:nil];  
}  
}
```

The method checks whether or not the user clicked on the Cancel button. If not, then you request access to the Twitter accounts array and retrieve the account at the position corresponding to the alert view's button index.

Once you have an account, you store its identifier in the user defaults and post the `AccountTwitterAccountAccessGranted` notification.

Hooray! You have now completed all of the code necessary to get a user's Twitter and Facebook accounts. ☺ Time to begin using the Social framework to access the Facebook and Twitter REST APIs and display some information.

Retrieving the Twitter feed

Believe it or not, you're now done with the Accounts framework!

Next it's time to get started working with the new iOS 6 Social Framework and the `SLRequest` class. If you're wondering what the `SLRequest` class is, we'll get to that soon enough. ☺

iSocial will focus primarily on Facebook integration, since Twitter was covered extensively in *iOS 5 by Tutorials* already. Nevertheless, to give you a brief overview of how Twitter functionality works in the new Social Framework, you will retrieve a user's Twitter feed and present it in a table view.

Brief overview of the Twitter documentation

To retrieve a user's Twitter feed, you need to call a specific method of the Twitter REST API. I'll show you how to use this API in this chapter, but it's important to understand how I figured this out in the first place, so you can use the same techniques to exercise *any* Twitter API call you'd like to use in your own apps.

First of all, you can see a list of all of the possible Twitter API calls you can make here: <http://dev.twitter.com/docs/api/1.1>

The screenshot shows the Twitter Developers REST API v1.1 Resources page. It has two main sections: 'Timelines' and 'Tweets'. The 'Timelines' section includes resources like 'GET statuses/mentions_timeline', 'GET statuses/user_timeline', and 'GET statuses/home_timeline'. The 'Tweets' section includes resources like 'GET statuses/reweets/:id', 'GET statuses/show/:id', and 'POST statuses/destroy/:id'. Each resource entry provides a brief description.

Take a look through to get an idea for the kinds of things you can do with the Twitter API. There's quite a lot there!

For iSocial, we want to retrieve a user's Twitter feed. The API call to do this is called **GET statuses/home_timeline** and the documentation on this call can be found here: https://dev.twitter.com/docs/api/1.1/get/statuses/home_timeline

The screenshot shows the detailed documentation for the 'GET statuses/home_timeline' endpoint. It includes information such as the last update (Wed, 2012-09-05 11:06), the REST API version (1.1), resource information (Rate Limited: Yes, Requests per rate limit window: 15, Authentication: Requires user context, Response Formats: json, HTTP Methods: GET, Resource family: statuses, Response Object: Tweets, API Version: v1.1), OAuth tool (with a note to sign in), and related documentation (link to 'GET statuses/user_timeline'). Parameters listed include count (optional, default 20, example value 5), since_id (optional, example value 12345), and max_id (optional, example value 56789).

The documentation for a particular resource specifies the resource URL, HTTP method, whether or not it requires authentication, and the parameters (both mandatory and optional) to send with the request. In this case, you will only use the count parameter to limit the number of posts retrieved.

Scrolling down the documentation page, you will find a sample response. Here's what a single post looks like in the JSON response:

```
{
    "coordinates": null,
```

```
"truncated": false,
"created_at": "Tue Aug 28 21:16:23 +0000 2012",
"favorited": false,
"id_str": "240558470661799936",
"in_reply_to_user_id_str": null,
"entities": {
    "urls": [
        ],
    "hashtags": [
        ],
    "user_mentions": [
        ]
},
"text": "just another test",
"contributors": null,
"id": 240558470661799936,
"retweet_count": 0,
"in_reply_to_status_id_str": null,
"geo": null,
"retweeted": false,
"in_reply_to_user_id": null,
"place": null,
"source": "<a href=\"http://realitytechnicians.com\" rel=\"nofollow\">OAuth Dancer Reborn</a>",
"user": {
    "name": "OAuth Dancer",
    "profile_sidebar_fill_color": "DDEEF6",
    "profile_background_tile": true,
    "profile_sidebar_border_color": "C0DEED",
    "profile_image_url":
"http://a0.twimg.com/profile_images/730275945/oauth-dancer_normal.jpg",
    "created_at": "Wed Mar 03 19:37:35 +0000 2010",
    "location": "San Francisco, CA",
    "follow_request_sent": false,
    "id_str": "119476949",
    "is_translator": false,
    "profile_link_color": "0084B4",
    "entities": {
        "url": {
            "urls": [
                {

```

```
        "expanded_url": null,
        "url": "http://bit.ly/oauth-dancer",
        "indices": [
            0,
            26
        ],
        "display_url": null
    }
]
},
"description": null
},
"default_profile": false,
"url": "http://bit.ly/oauth-dancer",
"contributors_enabled": false,
"favourites_count": 7,
"utc_offset": null,
"profile_image_url_https":
"https://si0.twimg.com/profile_images/730275945/oauth-
dancer_normal.jpg",
"id": 119476949,
"listed_count": 1,
"profile_use_background_image": true,
"profile_text_color": "333333",
"followers_count": 28,
"lang": "en",
"protected": false,
"geo_enabled": true,
"notifications": false,
"description": "",
"profile_background_color": "C0DEED",
"verified": false,
"time_zone": null,
"profile_background_image_url_https":
"https://si0.twimg.com/profile_background_images/80151733/oauth-
dance.png",
"statuses_count": 166,
"profile_background_image_url":
"http://a0.twimg.com/profile_background_images/80151733/oauth-
dance.png",
"default_profile_image": false,
"friends_count": 14,
"following": false,
"show_all_inline_media": false,
"screen_name": "oauth_dancer"
```

```
},
"in_reply_to_screen_name": null,
"in_reply_to_status_id": null
},
```

As you can see there's a ton of info there, but I've bolded some of the interesting bits you're likely to care about, such as the tweet text, created date, Twitter handle of the user who sent the tweet, and their profile image. Detailed descriptions of each of these fields can be found in the Twitter API docs.

Armed with this information, it's time to retrieve the feed!

Retrieving the Twitter feed

Open **TwitterFeedViewController.m** and add the following imports at the top of the file:

```
#import "AppDelegate.h"
#import "TwitterCell.h"
```

Now add a class extension with two properties below the `#import` lines:

```
@interface TwitterFeedViewController()
@property (strong, atomic) NSArray *tweetsArray;
@property (strong, atomic) NSMutableDictionary
*imagesDictionary;
@end
```

The `tweetsArray` property will store the Twitter feed retrieved from the Twitter API, and the `imagesDictionary` will be used to cache the user profile images once they've been downloaded.

Let's start adding the methods necessary to retrieve a user's Twitter feed. Add the following to `TwitterFeedViewController`'s implementation section:

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

    UIRefreshControl *refreshControl =
        [[UIRefreshControl alloc] init];
    [refreshControl addTarget:self
        action:@selector(refreshTwitterFeed)
        forControlEvents:UIControlEventValueChanged];
    self.refreshControl = refreshControl;
}
```

A `UIRefreshControl` is added to the table view controller. When the user pulls to refresh, `refreshTwitterFeed` (a method you will soon write) will be called to re-fetch the Twitter Feed.

Now add the following code:

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

    if ([self.view window] == nil)
    {
        self.view = nil;

        _imagesDictionary = nil;
        _tweetsArray = nil;
    }
}

- (void)viewWillDisappear:(BOOL)animated
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];

    [super viewWillDisappear:animated];
}
```

`didReceiveMemoryWarning` and `viewWillDisappear` clean up any memory and unregister any notifications observers previously registered.

Of course, you might notice that you haven't registered for any notifications in the code so far! That happens in the following code:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(refreshTwitterFeed)
        name:AccountTwitterAccountAccessGranted object:nil];

    AppDelegate *appDelegate = [[UIApplication
        sharedApplication] delegate];

    if (appDelegate.twitterAccount)
    {
        [self refreshTwitterFeed];
    }
}
```

```
    else
    {
        AppDelegate *appDelegate = [[UIApplication
            sharedApplication] delegate];
        [appDelegate getTwitterAccount];
    }
}
```

`viewWillAppear:` registers the for the `AccountTwitterAccountAccessGranted` notification so when the notification occurs, the `refreshTwitterFeed` method will be called (which you'll write in a moment). Remember, this notification is sent from the `getTwitterAccount` method you wrote earlier, when it successfully retrieves the user's Twitter account.

Also, `viewWillAppear` checks to see if the app delegate's `twitterAccount` property is `nil` or not. If it is, then `getTwitterAccount` is called; otherwise, `refreshTwitterFeed` is called to fetch the user's Twitter feed.

`refreshTwitterFeed` is the core method that handles the connection to the Twitter REST API and gets the user's feed. Here's the implementation – add it to **TwitterFeedViewController.m**:

```
- (void)refreshTwitterFeed
{
    // 1
    SLRequest *request = [SLRequest
        requestForServiceType:SLServiceTypeTwitter
        requestMethod:SLRequestMethodGET
        URL:[NSURL URLWithString:
            @"http://api.twitter.com/1/statuses/home_timeline.json"]
        parameters:@{@"count" : @"50"}];

    // 2
    AppDelegate *appDelegate =
        [[UIApplication sharedApplication] delegate];
    request.account = appDelegate.twitterAccount;

    // 3
    [request performRequestWithHandler:^(NSData *responseData,
        NSHTTPURLResponse *urlResponse, NSError *error) {
        [self.refreshControl endRefreshing];

    // 4
    if (error)
    {
        AppDelegate *appDelegate = (AppDelegate *)
        }
```

This is what the above code does:

1. `SLRequest` is the Social framework class you use to connect to Twitter and Facebook APIs. You use the factory method to get a request to Twitter for the home timeline endpoint using the GET HTTP method.

The HTTP method is determined in the API's documentation. In this case, the home timeline endpoint is called via GET. A dictionary of parameters is passed in so that the number of tweets received is limited to 50 items or less.

2. Retrieving a user's feed is an authenticated operation. To set this in the request, you first get the Twitter account from the app delegate, and then assign it to the request's `account` property. This ensures an authenticated request.
3. Now that all of the request's info and parameters have been set, you can perform the request by calling `performRequestWithHandler:`.
4. The request could fail for many reasons. If it does, then the block's `error` variable will not be `nil` and an error message is presented by calling `presentErrorWithMessage:` on the app delegate (you will add this method implementation soon).
5. If no errors occurred during the request, then you parse the JSON response and store it in an `NSArray`. In this case, you use an `NSArray` because the Twitter documentation for this request indicates that the response is an array of tweet dictionary objects.
6. Parsing a JSON response could also fail, so a conditional statement is used to check for any errors. If there are any errors, then you use the same utility method from the app delegate to present an alert view with the error.
7. If no error occurred when parsing the JSON response, then you create the `images` mutable dictionary and assign the parsed JSON object to `tweetsArray`. Finally, you use GCD to reload the table's data on the main thread.

`SLRequest` performs requests in a background thread, so the completion handler is not guaranteed to be executing in the main thread. This is the reason for reloading the table's data using GCD.

Time to add `presentErrorWithMessage:` to the app delegate! Open **AppDelegate.h** and add this method declaration inside the interface:

```
- (void)presentErrorWithMessage:(NSString *)errorMessage;
```

Then, in **AppDelegate.m**, add the method implementation:

```
- (void)presentErrorWithMessage:(NSString *)errorMessage
{
    UIAlertView *alertView = [[UIAlertView alloc]
        initWithTitle:@"Error"
        message:errorMessage
        delegate:nil
        cancelButtonTitle:@"Dismiss"
        otherButtonTitles:nil];
```

```
    dispatch_sync(dispatch_get_main_queue(), ^{
        [alertView show];
    });
}
```

That's pretty straightforward! This will save you some time, versus creating alert views all across your view controllers in iSocial.

The `show` method is called inside a GCD block because, for the most part, this method will be called inside `SLRequest` completion handlers, and those handlers are not guaranteed to execute in the main thread. Thus, adding a simple GCD call will help avoid any crashes.

Back in **TwitterFeedViewController.m**, you need to implement the appropriate table view data source methods. Add the following methods to your file:

```
- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return [self heightForCellAtIndex:indexPath.row];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return self.tweetsArray.count;
}
```

`tableView:heightForRowAtIndexPath:` calls a private method to dynamically set the row's height depending on the amount of text inside a post.

`tableView:numberOfRowsInSection:` returns the count of the `tweetsArray` property.

Implement `heightForCellAtIndex:` as shown below:

```
- (CGFloat)heightForCellAtIndex:(NSUInteger)index
{
    NSDictionary *tweet = self.tweetsArray[index];

    CGFloat cellHeight = 50;

    NSString *tweetText = tweet[@"text"];

    CGSize labelHeight = [tweetText sizeWithFont:[UIFont
        systemFontOfSize:15.0f]
        constrainedToSize:CGSizeMake(700, 250)];
```

```
    cellHeight += labelHeight.height;

    return cellHeight;
}
```

First you get the dictionary for the current row's tweet. Then you create a `CGFloat` variable with an initial height of 50 points. This height will vary, depending on what you used for the post label's height, so make sure you adjust it to what your storyboard reflects. Basically, the initial height should be the default cell height (110) minus the height of the post label.

The tweet's text is stored in `tweetText`, and then you call `sizeWithFont:constrainedToSize:` to determine the height of the label for the given tweet's text. The constrained size width is, once again, the width of your post label, so be sure to use your label's value. Also make sure that the font size matches the font size you have set, since by default the font size would be 17 instead of 15, as in the code above. If any of these values are wrong, you'll get an incorrect value for the calculated cell height.

Finally, you add the label's height to the default cell height and return it.

Next, implement `tableView:cellForRowAtIndexPath:` as shown below:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // 1
    TwitterCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"TwitterCell"];

    // 2
    NSDictionary *currentTweet = [self.tweetsArray
        objectAtIndex:indexPath.row];
    NSDictionary *currentUser = currentTweet[@"user"];

    // 3
    cell.usernameLabel.text = currentUser[@"name"];
    cell.tweetLabel.text = currentTweet[@"text"];
    cell.tweetLabel.frame =
        CGRectMake(cell.tweetLabel.frame.origin.x,
                  cell.tweetLabel.frame.origin.y,
                  700, ([self heightForCellAtIndex:indexPath.row] - 50));

    // 4
    NSString *userName = cell.usernameLabel.text;
```

```
if (self.imagesDictionary[userNamed])
{
    // 5
    cell.userImageView.image =
        self.imagesDictionary[userNamed];
}
else
{
    // 6
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        NSURL *imageURL = [NSURL URLWithString:
            [currentUser objectForKey:@"profile_image_url"]];

        __block NSData *imageData;

        // 7
        dispatch_sync(dispatch_get_global_queue(
            DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
            imageData =
                [NSData dataWithContentsOfURL:imageURL];

            [self.imagesDictionary setObject:
                [UIImage imageWithData:imageData]
                forKey:userNamed];
            // 8
            dispatch_sync(dispatch_get_main_queue(), ^{
                cell.userImageView.image =
                    self.imagesDictionary[cell.usernameLabel.text];
            });
        });
    });

    return cell;
}
```

Let's break this down step-by-step:

1. A TwitterCell instance is either de-queued from the table view or created from scratch. You are using the same identifier you set up inside Interface Builder for the custom Twitter cell.
2. You retrieve a dictionary corresponding to the current tweet from the tweets array, in addition to the current user's dictionary object.

3. You fill the cell's labels with the appropriate text and the `tweetLabel`'s frame is resized to the height returned by `heightForCellAtIndex:`. (Note that you'll again need to adjust the 700 for the width and the 50 deducted to match your layout.)
4. You store the user's username in an `NSString` – it will be used in a few places in this method, so it's easier to have a variable for it.
5. To improve performance and reduce the amount of network usage, you store the user profile images inside the mutable dictionary you declared earlier. Because the current cell's user image could already have been retrieved, you perform a check. If the image was found in the dictionary, then you set the cell's image view.
6. If no image was found in the dictionary, then you download it. Because downloading many images could take some time and lock up your interface, you will once again use GCD for this.

An asynchronous block is created in a background thread, the user's profile image URL is retrieved, and an empty `NSData` object is created (to store the downloaded image).

7. A `dispatch_async()` call will return immediately, whereas a `dispatch_sync()` call will wait until the block finishes executing. You want the former to avoid locking up the UI.

For downloading the image, however, you use a `dispatch_sync()` call, so that you download the image (still asynchronously) before returning from this block. This will optimize performance by fully downloading an image before the queue is available for more work.

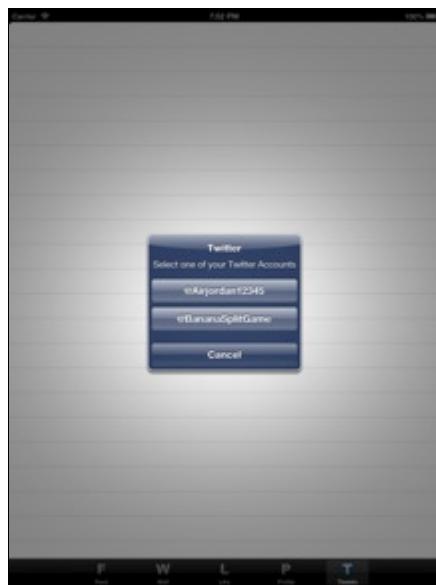
You'd rather have the user see images appear one by one than no images for several seconds. You also store the image in the dictionary. From now on, the image will be retrieved from the cache and performance should be picture-perfect.

8. After the image has been downloaded, you make a call on the main thread to update the cell's image view with the newly-acquired image.

The first moment of truth

This is an epic moment in the chapter and in your Social/Accounts framework career – the first build and run of iSocial! ☺

Run your app and tap the Tweets tab. If you have any Twitter accounts set up in the system settings, then the app will retrieve your Twitter account (after asking you to grant it access). If you have more than one account, you are shown an alert view to select the account you prefer.



After you grant the app access to Twitter and (if necessary) select your preferred account, you should see your table view populate with beautiful, dynamically-sized cells full of tweets!



Notice how the images download very quickly and without locking up or freezing the UI. Fantastic work, my friend! You've now completed the first view controller of the app.

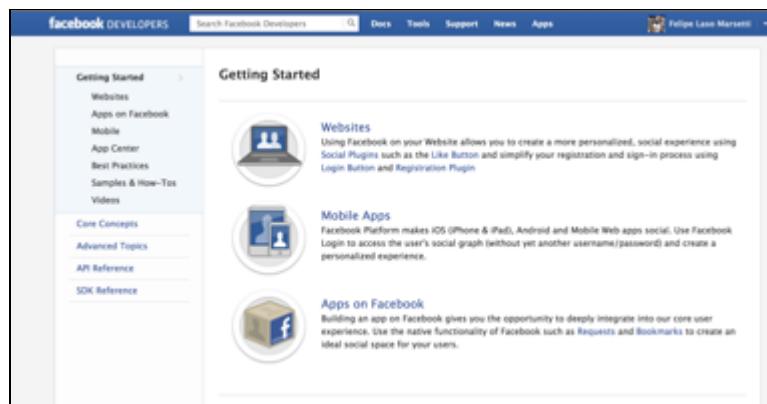
That's all the Twitter interaction for iSocial. For more information on things you can do with the Twitter API, please refer to Chapter 13 in *iOS 5 By Tutorials*.

Facebook documentation overview

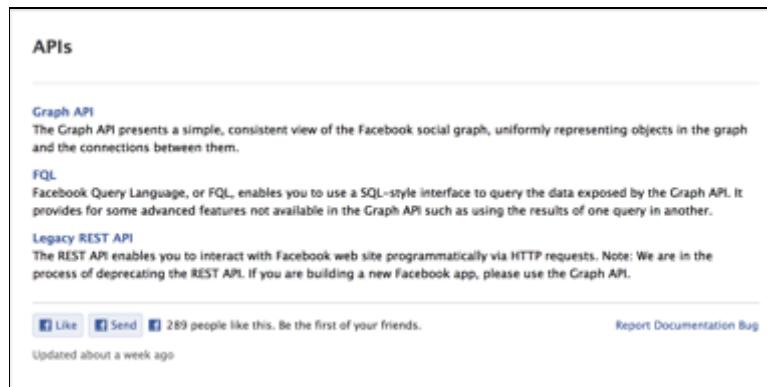
Compared to the Twitter documentation, the Facebook documentation is a serious mess. I often find it frustrating to locate what I need and I feel as if everything is just all over the place.

It's for this reason that I think a quick overview of how to read, use and find the proper documentation will particularly help you before working with Facebook!

Go to <http://developers.facebook.com> and click on Docs at the top of the window. On the left side of the screen you should see some links, and amongs them should be **API Reference**. Click on it, and you will see the following:

A screenshot of the Facebook Developers website. The URL is http://developers.facebook.com. At the top, there is a navigation bar with links for Docs, Tools, Support, News, and Apps. A user profile for 'Felipe Leme Marcelli' is visible on the right. The main content area has a sidebar on the left with links to Getting Started (Websites, Apps on Facebook, Mobile, App Center, Best Practices, Samples & How-Tos, Videos), Core Concepts, Advanced Topics, API Reference (which is currently selected and highlighted in blue), and SDK Reference. The main content area is titled 'Getting Started' and contains three sections: 'Websites', 'Mobile Apps', and 'Apps on Facebook'. Each section has a small icon and a brief description.

On this new page you will find a few options for APIs. You are interested in the **Graph API**, so click on that link.

A screenshot of the 'Graph API' documentation page. The title 'APIs' is at the top. Below it, there are three sections: 'Graph API' (described as a simple, consistent view of the Facebook social graph), 'FQL' (Facebook Query Language), and 'Legacy REST API' (described as enabling interaction via HTTP requests). At the bottom of the page, there are social sharing buttons ('Like', 'Send') and a note that 289 people like this. There is also a link to 'Report Documentation Bug' and a timestamp 'Updated about a week ago'.

Welcome to the Graph API documentation:

On the left side you will see the main sections: the Core Concepts section for the Graph API and the sections covering the objects that you will interact with as you use the API.

The first thing you will interact with is the user's profile, so locate that object on the left side (it's near the bottom) and click on it (hint: it's the **User** link ☺).

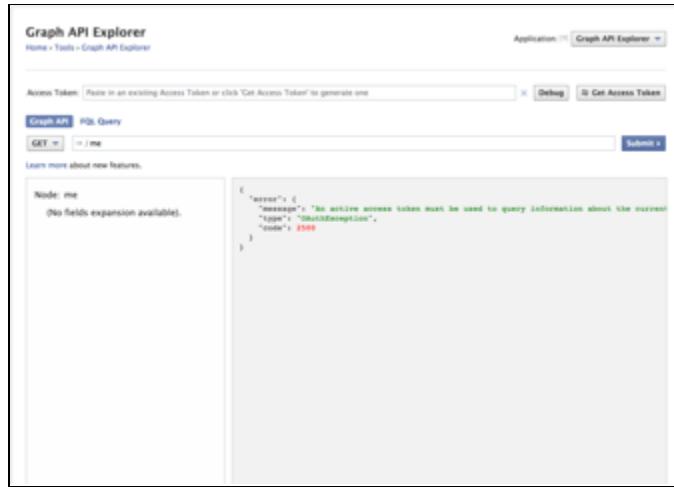
The documentation for an object contains the resource's URL and a Fields table with some interesting information, as well as a bunch of information specific to that object. In the case of the user, we see how to send posts, check likes, add friends, etc. Take a look:

| Name | Description | Permissions | Returns |
|-----------------------------|---|--------------------------|--|
| <code>id</code> | The user's Facebook ID | No access_token required | string |
| <code>name</code> | The user's full name | No access_token required | string |
| <code>first_name</code> | The user's first name | No access_token required | string |
| <code>middle_name</code> | The user's middle name | No access_token required | string |
| <code>last_name</code> | The user's last name | No access_token required | string |
| <code>gender</code> | The user's gender: <code>female</code> or <code>male</code> | No access_token required | string |
| <code>locale</code> | The user's locale | No access_token required | string containing the ISO Language Code and ISO Country Code |
| <code>languages</code> | The user's languages | <code>user_likes</code> | array of objects containing language <code>id</code> and <code>name</code> |
| <code>link</code> | The URL of the profile for the user on Facebook | No access_token required | string containing a valid URL |
| <code>username</code> | The user's Facebook username | No access_token required | string |
| <code>third_party_id</code> | An anonymous, but unique identifier for the user; only returned if specifically requested | Requires access_token | string |

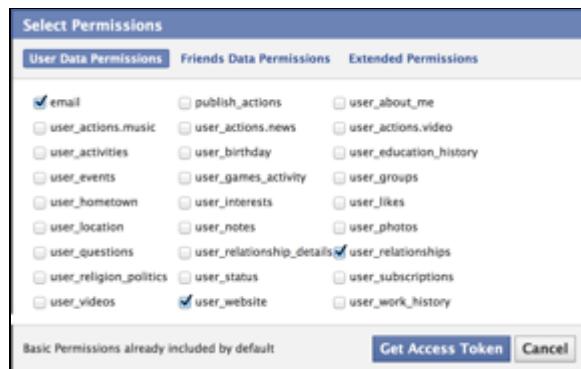
The Fields table has a few columns:

- **Name:** What you will pass in with your request's parameter.
- **Description:** An informative description of the field.
- **Permissions:** Lets you know what permissions you need to specify when accessing the user's system Facebook account in order access this field.
- **Returns:** is the type of object you will get back for this particular field.

Click on the example link inside the User documentation page and take a look. Sweet, a fancy web console for you to play and experiment with!



If you don't have an access token already, click the **Get Access Token** button to see the permissions overlay.



From the User Data Permissions, select **email**, **user_website** and **user_relationships**, then click **Next**. You may be asked to grant access to the Facebook Graph API explorer app – go ahead and do so.

Back in the web console, click on Submit and take a look at the results.

The screenshot shows the Facebook Graph API web console. In the top left, there are buttons for 'Graph API' and 'PQL Query'. Below that, a dropdown menu says 'GET' and has a 'me' option selected. To the right of the dropdown is a search bar and a 'Submit' button. Underneath the dropdown, there's a link 'Learn more about new features.' On the left, a sidebar labeled 'Node: me' contains some user information. On the right, the main area displays a large JSON object representing a user profile. The JSON includes fields like 'id', 'name', 'first_name', 'last_name', 'username', 'bio', 'gender', 'relationship_status', 'location', 'website', 'languages', and several 'languages' arrays. One array contains 'id' and 'name' for 'Arabic' and 'English'. Another array contains 'id' and 'name' for 'Spanish' and 'English'. At the bottom of the JSON object, there are fields for 'created_time' and 'updated_time'.

```

{
  "id": "100000000000000",
  "name": "Felipe Laxe Marsetti",
  "first_name": "Felipe",
  "last_name": "Laxe Marsetti",
  "username": "felipe_laxe",
  "bio": "I'm a code monkey living in Middle Earth. I wanna be the next John Carmack :). Yip, I'm a code monkey living in Middle Earth.",
  "gender": "male",
  "relationship_status": "Single",
  "location": "Middle Earth, Middle-earth",
  "website": "http://ide.tt",
  "languages": [
    {
      "id": "100000000000000",
      "name": "Spanish"
    },
    {
      "id": "100000000000001",
      "name": "English"
    }
  ],
  "created_time": "2012-08-30T00:00:00+0000",
  "updated_time": "2013-08-30T00:00:01+0000"
}

```

These are the default results without specifying any fields.

The web console is a nice way for you to test out calls and see what the JSON response will look like before coding your app. It's also of great help when you're debugging and you don't want to write lines and lines of code just for testing.

Feel free to do some tests with the console and, when ready, come back to the next section to retrieve and show the user's profile inside iSocial.

Showing the Facebook profile

The UI for the user's Facebook profile should be setup for you in the starter project. Let's start adding the code necessary for things to work.

Open **ProfileViewController.m** and add the following to the top of the file:

```
#import "AppDelegate.h"
#import "WebViewController.h"
```

And add a new property to keep track of the profile information into the class extension:

```
@property (strong) NSDictionary * profileDictionary;
```

Then add the following inside the implementation section:

```
- (void)didReceiveMemoryWarning
{
    if ([self.view window] == nil)
    {
        _bioTextView = nil;
        _birthdayLabel = nil;
        _coverImageView = nil;
        _emailLabel = nil;
```

```
_genderLabel = nil;
_hometownLabel = nil;
_languagesLabel = nil;
_locationLabel = nil;
_nameLabel = nil;
_pictureImageView = nil;
_relationshipStatusLabel = nil;
_usernameLabel = nil;
_websiteButton = nil;
}

[super didReceiveMemoryWarning];
}

- (void)viewWillDisappear:(BOOL)animated
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];

    [super viewWillDisappear:animated];
}
```

`didReceiveMemoryWarning` is the standard implementation you've seen so far. You remove the controller as an observer of any notifications, and clean up the properties that may reside in memory. `viewWillDisappear:` just removes the controller as a notification observer.

Add the following next:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(reloadProfile)
        name:AccountFacebookAccountAccessGranted object:nil];

    AppDelegate *appDelegate =
        [[UIApplication sharedApplication] delegate];

    if (appDelegate.facebookAccount)
    {
        [self reloadProfile];
    }
    else
    {
        AppDelegate *appDelegate =
```

```
        [[UIApplication sharedApplication] delegate];
        [appDelegate getFacebookAccount];
    }
}
```

This code is similar to what was written in TwitterFeedViewController.m. You first register for the `AccountFacebookAccountAccessGranted` notification, and then you check for an existing Facebook account in the app delegate's property.

If an account exists, you call `reloadProfile` (which you will implement next) to get the user's profile data; otherwise, you call the app delegate's `getFacebookAccount` method. Simple stuff, right?

Time to write the biggest method in this controller, `reloadProfile`:

```
- (void)reloadProfile
{
    // 1
    SLRequest *request = [SLRequest
        requestForServiceType:SLServiceTypeFacebook
        requestMethod:SLRequestMethodGET
        URL:[NSURL URLWithString:@"https://graph.facebook.com/me"]
        parameters:@{ @"fields" :
    @"bio,birthday,cover,email,first_name,gender,hometown,languages,
    last_name,link,location,picture,relationship_status,security_setting
    s,username,website"}];

    // 2
    AppDelegate *appDelegate =
        [[UIApplication sharedApplication] delegate];
    request.account = appDelegate.facebookAccount;

    // 3
    [request performRequestWithHandler:^(NSData *responseData,
        NSHTTPURLResponse *urlResponse, NSError *error) {
        if (error)
        {
            // 4
            AppDelegate *appDelegate = (AppDelegate *)
                [[UIApplication sharedApplication] delegate];

            [appDelegate presentErrorWithMessage:
                [NSString stringWithFormat:
                    @"There was an error reading your Facebook feed. %@",

                    [error localizedDescription]]];
        }
    }];
}
```

```
else
{
    // 5
    NSError *jsonError;
    NSDictionary *responseJSON = [NSJSONSerialization
        JSONObjectWithData:responseData
        options:NSJSONReadingAllowFragments
        error:&jsonError];

    if (jsonError)
    {
        // 6
        AppDelegate *appDelegate = (AppDelegate *)
            [[UIApplication sharedApplication] delegate];

        [appDelegate presentErrorWithMessage:
            [NSString stringWithFormat:
                @"There was an error reading your Facebook feed. %@",

                [error localizedDescription]]];
    }
    else
    {
        // 7
        self.profileDictionary = responseJSON;
        [self getPictures];

        // 8
        dispatch_async(dispatch_get_main_queue(), ^{
            self.bioTextView.text =
                self.profileDictionary[@"bio"];
            self.birthdayLabel.text =
                self.profileDictionary[@"birthday"];
            self.emailLabel.text =
                self.profileDictionary[@"email"];
            self.genderLabel.text =
                self.profileDictionary[@"gender"];
            self.hometownLabel.text =
                self.profileDictionary[@"hometown"][@name"];
    }

    // 9
    NSArray *languages =
        self.profileDictionary[@languages];
    NSMutableString *languagesString =
        [NSMutableString stringWithString:@""];
}
```

```
        for (int i = 0; i < languages.count; i++)
        {
            NSDictionary *language = languages[i];

            [languagesString
                appendString:language[@"name"]];

            if (i < (languages.count - 1))
            {
                [languagesString appendString:@", "];
            }
        }

        self.languagesLabel.text = [NSString
            stringWithFormat:@"%@", languagesString];

        // 10
        self.locationLabel.text =
        self.profileDictionary[@"location"][@"name"];
        self.nameLabel.text = [NSString
            stringWithFormat:@"%@ %@", self.profileDictionary[@"first_name"],
            self.profileDictionary[@"last_name"]];
        self.usernameLabel.text =
            self.profileDictionary[@"username"];
        self.relationshipStatusLabel.text =
        self.profileDictionary[@"relationship_status"];

        // 11
        if (!self.profileDictionary[@"website"])
        {
            self.websiteButton.hidden = YES;
        }
        else
        {
            self.websiteButton.hidden = NO;
        }

        self.facebookButton.hidden = NO;
    });
}
}];
}
```

Let's walk through things together, one step at a time.

1. As usual, the first thing you need to do is create an `SLRequest` with the resource endpoint, the HTTP method, and request parameters. To get a user's profile, you specify all of the fields you want to get. It's a long options dictionary, but it's not complicated. ☺
2. With the request ready, you then retrieve the user's Facebook account from the app delegate and assign it to the requests account property. This is a required step, since this resource expects authentication.
3. You now perform the actual request by calling `performRequestWithHandler:`.
4. If an error occurred during the request, you use the app delegate's utility method to show an alert view with an error message.
5. If no request errors occurred, then you proceed to parse the JSON response into a native `NSDictionary` object.
6. It's possible that an error occurred during the JSON parsing and if it did, you use the app delegate's utility method to present an alert view with the error.
7. Inside this block of code is where you process the request when everything is successful. The first thing you do is assign the parsed JSON object to `profileDictionary`, and then you call `getPictures` to retrieve the user's profile and cover images.
8. Because all operations on the UI must be done in the main thread, you call a GCD block on the main thread and update some of the labels with the values retrieved from Facebook.

Note: If you need to remember what the response structure looks like and what keys to use to read the info from the dictionary, please refer back to the Graph API documentation.

9. The user's languages are returned inside an array. This portion of code puts all of the languages together into a single, comma-separated string to be shown inside a single label.
10. The remaining labels are given their proper values from the info inside the profile dictionary.
11. Finally, you check if the user has set the website field in their profile. If there is a website, then you unhide the website button; otherwise you hide it.

The Facebook button is unhidden because every user profile can be seen on Facebook. Now that the user's profile info has been acquired, it's safe to let users tap that button by unhiding it.

Phew! That was a lot of code, but you made it to the end! ☺

Add the implementation for `getPictures` as shown below:

```
- (void)getPictures
```

```
{  
    dispatch_async(dispatch_get_global_queue(  
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{  
  
        NSString *pictureURLString =  
            self.profileDictionary[@"picture"][@"data"][@"url"];  
  
        NSURL *pictureURL = [NSURL  
            URLWithString:pictureURLString];  
  
        __block NSData *pictureData;  
  
        dispatch_sync(dispatch_get_global_queue(  
            DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{  
            pictureData = [NSData  
                dataWithContentsOfURL:pictureURL];  
  
            UIImage *pictureImage =  
                [UIImage imageWithData:pictureData];  
  
            dispatch_sync(dispatch_get_main_queue(), ^{  
                self.pictureImageView.image = pictureImage;  
            });  
        });  
    });  
  
    dispatch_async(dispatch_get_global_queue(  
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{  
  
        NSString *coverURLString =  
            self.profileDictionary[@"cover"][@"source"];  
  
        NSURL *coverURL = [NSURL URLWithString:coverURLString];  
  
        __block NSData *coverData;  
  
        dispatch_sync(dispatch_get_global_queue(  
            DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{  
            coverData = [NSData dataWithContentsOfURL:coverURL];  
            UIImage *coverImage =  
                [UIImage imageWithData:coverData];  
  
            dispatch_sync(dispatch_get_main_queue(), ^{  
                self.coverImageView.image = coverImage;  
            });  
        });  
    });  
}
```

```
    });
}
}
```

This code has two asynchronous operations on GCD to retrieve the profile and cover images, respectively.

Each block gets the image URL from the profile dictionary and loads it into an instance of `NSData` by calling `dataWithContentsOfURL:`.

When the image's data has been downloaded, it's converted to a `UIImage` object and, inside a block called on the main thread, the appropriate image view outlet is updated with the image.

Finally, add the action methods for each button:

```
- (IBAction)viewOnFacebookTapped
{
    NSString * urlString = self.profileDictionary[@"link"];

    WebViewController * webViewController = [self.storyboard
        instantiateViewControllerWithIdentifier:
        WebViewControllerIdentifier];
    webViewController.initialURLString = urlString;

    [self presentViewController:webViewController animated:YES
        completion:nil];
}

- (IBAction)viewWebsiteTapped
{
    NSString * urlString = self.profileDictionary[@"website"];

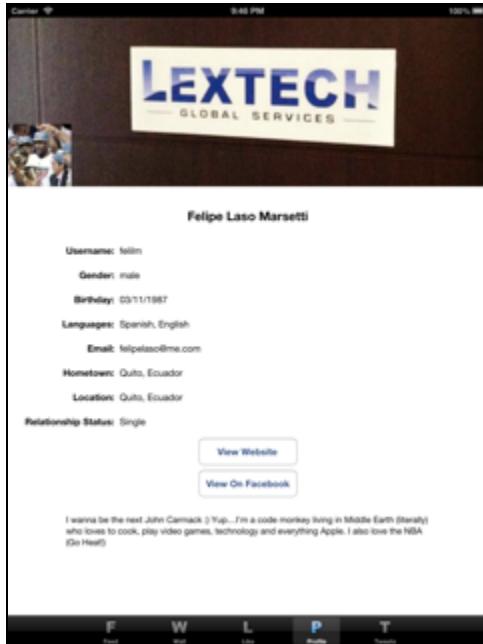
    WebViewController * webViewController = [self.storyboard
        instantiateViewControllerWithIdentifier:
        WebViewControllerIdentifier];
    webViewController.initialURLString = urlString;

    [self presentViewController:webViewController animated:YES
        completion:nil];
}
```

Both of these methods retrieve the URL for the requested resource, create an instance of `WebViewController`, set the `initialURLString` property and present the view to the user.

Running and viewing the profile

Time again to build and run the project and check out the results. Since this is the first time you access the user's Facebook account, you may be asked to grant permissions. Once you allow that, check out the results:



Ohh noes, I'm single! ☺ And my website still needs some work, as you will see in the next image. It's a hard time to be Feli! ☺



The Facebook feed and wall

Ready to begin writing the code to retrieve a user's Facebook feed and wall?

This section is a tad bit long, as it contains the core portions of the code in the app. If necessary, take a small break before beginning, stretch your legs, grab a snack or a drink, and come back. I'll be here waiting for when you are ready. ☺

Back? Fantastic, let's get to work.

A user's feed or wall can contain objects of different types. Because there are so many object types and a limited number of pages in this chapter (we don't want to cover the entire Facebook API ☺), iSocial will only handle items that are status updates, links, photos, and videos.

A user's feed is identical to the wall, except for the Graph API endpoint used to retrieve the information. What this means for you is that `FacebookWallViewController` will use all of the code written in `FacebookFeedViewController`, sans two methods that need to be customized for each one.

Add these method declarations inside **FacebookFeedViewController.h**:

```
- (NSString *)feedString;
- (NSString *)titleString;
```

Now go to **FacebookFeedViewController.m** and implement them as follows:

```
- (NSString *)feedString
{
    return @"https://graph.facebook.com/me/home";
}

- (NSString *)titleString
{
    return @"Feed";
}
```

`feedString` returns the Graph API path for the resource you want, in this case the user's feed. `titleString` returns the title to be shown on the navigation bar.

Since `FacebookWallViewController` is a sub-class of `FacebookFeedViewController`, these two methods are all you need in **FacebookWallViewController.m**. So go ahead and add them as shown:

```
- (NSString *)feedString
{
    return @"https://graph.facebook.com/me/feed";
}
```

```
}

- (NSString *)titleString
{
    return @"Wall";
}
```

The URL for the user's wall is obviously different from the feed string. So is the navigation bar title.

Time to return to **FacebookFeedViewController.m** to add the necessary import statements and some properties.

Add all of these imports at the top of the file:

```
#import "AppDelegate.h"
#import "CommentViewController.h"
#import "FacebookCell.h"
#import "MessageCell.h"
#import "PhotoCell.h"
#import "WebViewController.h"
```

Then add a new class extension, right below the import statements:

```
@interface FacebookFeedViewController ()
@property (strong, atomic) NSArray *feedArray;
@property (strong, atomic) NSMutableDictionary *imagesDictionary;
@end
```

If you remember, this code is nearly identical to that of `TwitterFeedViewController`. There's an array for the feed objects and a mutable dictionary to cache the profile images.

Replace `didReceiveMemoryWarning` and add `viewWillDisappear:` as follows:

```
- (void)didReceiveMemoryWarning
{
    if ([self.view window] == nil)
    {
        _feedArray = nil;
        _imagesDictionary = nil;
    }

    [super didReceiveMemoryWarning];
}

- (void)viewWillDisappear:(BOOL)animated
```

```
{  
    [[NSNotificationCenter defaultCenter] removeObserver:self];  
  
    [super viewWillDisappear:animated];  
}
```

This follows the same pattern as in previous controllers.

Next, replace `viewDidLoad` with the following:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    UIRefreshControl *refreshControl =  
        [[UIRefreshControl alloc] init];  
    [refreshControl addTarget:self  
        action:@selector(refreshFacebookFeed)  
        forControlEvents:UIControlEventValueChanged];  
    self.refreshControl = refreshControl;  
  
    self.title = [self titleString];  
}
```

`viewDidLoad` creates a refresh control so your users can manually refresh their feed. It also sets the controller's title (for the navigation bar) to the return value of `titleString`.

Add the following:

```
- (void)viewWillAppear:(BOOL)animated  
{  
    [super viewWillAppear:animated];  
  
    [[NSNotificationCenter defaultCenter] addObserver:self  
        selector:@selector(refreshFacebookFeed)  
        name:AccountFacebookAccountAccessGranted object:nil];  
  
    AppDelegate *appDelegate =  
        [[UIApplication sharedApplication] delegate];  
  
    if (appDelegate.facebookAccount)  
    {  
        [self refreshFacebookFeed];  
    }  
    else
```

```
{  
    AppDelegate *appDelegate =  
        [[UIApplication sharedApplication] delegate];  
    [appDelegate getFacebookAccount];  
}  
}
```

You register for the `AccountFacebookAccountAccessGranted` notification and check whether the app delegate's Facebook account is `nil` or not. If it's `nil` then you fetch the account from the system database, otherwise you download the Facebook feed.

`refreshFacebookFeed` will be in charge of downloading the actual Facebook feed. Add the implementation for that method as shown below:

```
- (void)refreshFacebookFeed  
{  
    // 1  
    SLRequest *request = [SLRequest  
        requestForServiceType:SLServiceTypeFacebook  
        requestMethod:SLRequestMethodGET  
        URL:[NSURL URLWithString:[self feedString]]  
        parameters:@{ @"limit" : @"30" }];  
    AppDelegate *appDelegate =  
        [[UIApplication sharedApplication] delegate];  
    request.account = appDelegate.facebookAccount;  
  
    // 2  
    [request performRequestWithHandler:^(NSData *responseData,  
                                         NSHTTPURLResponse *urlResponse, NSError *error) {  
        [self.refreshControl endRefreshing];  
  
        if (error)  
        {  
            // 3  
            AppDelegate *appDelegate = (AppDelegate *)  
                [[UIApplication sharedApplication] delegate];  
  
            [appDelegate presentErrorWithMessage:  
                [NSString stringWithFormat:  
                    @"There was an error reading your Facebook feed. %@",  
                    [error localizedDescription]]];  
        }  
        else  
        {  
            // 4  
            NSError *jsonError;
```

```
NSDictionary *responseJSON = [NSJSONSerialization
    JSONObjectWithData:responseData
    options:NSJSONReadingAllowFragments
    error:&jsonError];

if (jsonError)
{
    // 5
    AppDelegate *appDelegate = (AppDelegate *)
        [[UIApplication sharedApplication] delegate];

    [appDelegate presentErrorWithMessage:
        [NSString stringWithFormat:
            @"There was an error reading your Facebook feed. %@", 
            [error localizedDescription]]];
}

else
{
    // 6
    NSMutableArray *cleanFeedArray =
        [NSMutableArray array];

    for (NSDictionary *item in responseJSON[@"data"])
    {
        if ([item[@"type"] isEqualToString:@"status"] ||
            [item[@"type"] isEqualToString:@"link"] ||
            [item[@"type"] isEqualToString:@"photo"] ||
            [item[@"type"] isEqualToString:@"video"])
        {
            if ([item[@"type"] isEqualToString:@"status"])
            {
                if (!item[@"message"])
                {
                    continue;
                }
            }

            [cleanFeedArray addObject:item];
        }
    }

    // 7
    self.feedArray =
        [NSArray arrayWithArray:cleanFeedArray];
    self.imagesDictionary =

```

```
        [NSMutableDictionary dictionary];

    // 8
    dispatch_async(dispatch_get_main_queue(), ^{
        [self.tableView reloadData];
    });
}
}];
```

Let's take a look at each block of code. This should, however, look quite familiar, as you've used the `SLRequest` class in several places already.

1. You create an `SLRequest` instance, and set the HTTP method, resource URL, parameters, and the account to use for authentication. The URL for the user's feed is acquired via `feedString`. For parameters, you set `limit` to 30 so that only 30 items are returned via the response.
 2. You perform the request and pass in a completion block to parse the feed or check for errors when done.
 3. If an error occurred during the operation, then you present an alert view to the user using the app delegate's utility method.
 4. Otherwise, you proceed to parse the JSON response into an `NSDictionary`.
 5. If an error occurred when parsing the JSON data, then you show an alert view to the user.
 6. When no errors occur in the request or when parsing it, you get back a dictionary with the response. The `data` object of the response dictionary contains an array of feed items. For iSocial, you don't want to use all of the items, so you create a temporary array and store only status, link, video, or photo items in it.
 7. You store the filtered array in `feedArray` and create an empty mutable dictionary to cache the profile pictures.
 8. Finally, you call `reloadData` on the table view inside a GCD operation on the main thread. The completion handler of an `SLRequest` isn't guaranteed to execute on the main thread, so this step is required.

Table data source methods

Moving on to the table view data source, remove all the existing table view boilerplate code for the table view delegate and data source, and then add the following methods:

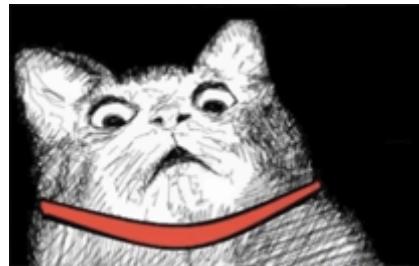
```
- (CGFloat)tableView:(UITableView *)tableView  
heightForRowAtIndexPath:(NSIndexPath *)indexPath  
{
```

```
    return [self heightForCellAtIndex:indexPath.row];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return self.feedArray.count;
}
```

The `heightForCellAtIndex:` method (which you'll add soon) dynamically calculates the height of a row, depending on the type of Facebook object and its contents. For the number of rows in the table view, you return the size of `feedArray`.

Brace yourself for `tableView:cellForRowIndexPath:`, the longest method in this controller!



You will add the code for it in portions. Start by defining the method as follows:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowIndexPath:(NSIndexPath *)indexPath
{
    // 1
    FacebookCell *cell;
    cell.userImageView.image = nil;

    // 2
    NSDictionary *currentItem = [self.feedArray
        objectAtIndex:indexPath.row];
    NSDictionary *currentUser = currentItem[@"from"];
    NSString *currentUserID = currentUser[@"id"];

    // 3
    if ([currentItem[@"type"] isEqualToString:@"status"])
    {
        // Code to format status cell
    }
    // 4
    else if ([currentItem[@"type"] isEqualToString:@"link"])
    {
```

```
// Code to format link cell
}

// 5
else if ([currentItem[@"type"] isEqualToString:@"photo"] ||
          [currentItem[@"type"] isEqualToString:@"video"])
{
    // Code to format photo or video cell
}

// 6
if (self.imagesDictionary[currentUserID])
{
    cell.userImageView.image =
        self.imagesDictionary[currentUserID];
}
else
{

// 7
dispatch_async(dispatch_get_global_queue(
    DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    NSString *pictureURL = [NSString
        stringWithFormat:@"%@/%@/picture?type=small",
        @"https://graph.facebook.com",
        currentUser[@"id"]];

    NSURL * imageURL = [NSURL URLWithString:pictureURL];

    __block NSData * imageData;

    // 8
    dispatch_sync(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        imageData =
            [NSData dataWithContentsOfURL:imageURL];

        [self.imagesDictionary setObject:
            [UIImage imageWithData:imageData]
            forKey:currentUserID];

        dispatch_sync(dispatch_get_main_queue(), ^{
            cell.userImageView.image =
                self.imagesDictionary[currentUserID];
        });
    });
}
```

```
        });
    });

    return cell;
}
```

Here's what's happening above:

1. You create a `FacebookCell` variable. Either a `MessageCell` or `PhotoCell` instance will be created (both are subclasses of `FacebookCell`), depending on the item type for the current cell.
By having a pointer to the super class, you can easily use it for either type of cell and determine what subclass to create at runtime, as you will see in a second.
2. You store the current item's dictionary, the user's dictionary, and the user's ID string in local variables.
3. There's a big conditional statement that starts here. Depending on the type of item, you will create and format the cell appropriately. In this case, the conditional statement will execute the code if the current item is a status item.
4. If the Facebook item for the current cell is a link item, then the cell is set up here.
5. If the item is either a video or photo, then the cell is properly sized and formatted in this section of code.
6. After the cell has been formatted, you check to see if the profile image of the posting user is already cached in your images dictionary. If so, then you use it to populate the cell's image view.
7. When a user image isn't found in the dictionary, you make an asynchronous call using GCD to handle the process of downloading the image.
8. This block (inside the asynchronous background call to GCD) is the one that handles the actual download of the image and the placing of it in the cell's image view (this is done in the main thread via a GCD call).

With the core method in place, it's time to put in the code to create and format the cell according to the Facebook item type, starting with a status item. Paste the following in section #3 of the above code:

```
// 1
MessageCell *messageCell = [tableView
    dequeueReusableCellWithIdentifier:@"MessageCell"];

messageCell.usernameLabel.text =
    currentItem[@"from"][@"name"];
messageCell.messageLabel.text = currentItem[@"message"];
```

```
// 2
CGFloat messageLabelHeight = [self
    heightForString:currentItem[@"message"]
    withFont:[UIFont systemFontOfSize:15.0f]
    constrainedtoSize:CGSizeMake(670, 500)];

messageCell.messageLabel.frame =
    CGRectMake(messageCell.messageLabel.frame.origin.x,
    messageCell.messageLabel.frame.origin.y,
    670, messageLabelHeight);

// 3
NSArray *toArray = currentItem[@"to"][@"data"];

if (toArray.count > 0)
{
    messageCell.toLabel.hidden = NO;
    messageCell.toLabel.text =
        [self toLabelStringFromArray:toArray];
    messageCell.toLabel.frame =
        CGRectMake(messageCell.toLabel.frame.origin.x,
        messageCell.messageLabel.frame.origin.y +
        messageCell.messageLabel.frame.size.height + 5,
        messageCell.toLabel.frame.size.width,
        [self heightForString:messageCell.toLabel.text
            withFont:[UIFont systemFontOfSize:15.0f]
            constrainedtoSize:CGSizeMake(670, 500)]);
}
else
{
    messageCell.toLabel.hidden = YES;
}

// 4
int commentCount =
    ((NSArray *)currentItem[@"comments"][@"data"]).count;

if (commentCount > 0)
{
    messageCell.accessoryType =
        UITableViewCellAccessoryDisclosureIndicator;
    messageCell.selectionStyle =
        UITableViewCellStyleBlueBackground;
    messageCell.userInteractionEnabled = YES;
}
```

```
else
{
    messageCell.accessoryType =
        UITableViewCellAccessoryNone;
    messageCell.selectionStyle =
        UITableViewCellSelectionStyleNone;
    messageCell.userInteractionEnabled = NO;
}

// 5
cell = messageCell;
```

Ready to step through it? Here's what's happening:

1. Status items are to be shown in a `MessageCell`. You acquire a cell from the table view and set the contents of the username and message labels.

2. A message in a Facebook status can be rather long. It is necessary to calculate the message label's height depending on the amount of text it contains. This section handles that. You can use whatever font you prefer, just make sure that the font you specify in code is the same as the one you've set in Interface Builder for your cell's label.

Similarly, for the constrain size of the label, be sure to use your message label's width as it's set in Interface Builder.

3. A status could be directed to specific people. To check this, you retrieve the array of people the message is directed to and loop through it as long as there is at least one item.

If the condition is met (meaning there is at least one person tagged in the status update) then you call `toLabelStringFromArray:` to return a single string with all of the recipients concatenated and separated by commas.

With the string of recipients ready, you calculate the height for `toLabel` and set its frame accordingly.

4. As you know, Facebook allows for users to comment on items you share. If the current item has comments, then you allow for selection on the cell and enable the accessory disclosure indicator.

This allows you to use the cell accessory to indicate that an item has comments. It also allows your users to select an item that has comments, in order to view them.

5. When setup for the cell is complete, you assign it to the `FacebookCell` pointer. This is so the cell can be properly returned at the end of the method, but with the correct subclass and formatting.

Paste the next block of code into the table view method, in the `if` condition for the link item (below the `// code to format link cell comment`):

```
MessageCell *messageCell = [tableView
    dequeueReusableCellWithIdentifier:@"MessageCell"];

    messageCell.usernameLabel.text =
        currentItem[@"from"][@"name"];
    messageCell.messageLabel.text = currentItem[@"name"];

    CGFloat descriptionLabelHeight = [self
        heightForString:currentItem[@"name"]
        withFont:[UIFont systemFontOfSize:15.0f]
        constrainedtoSize:CGSizeMake(670, 500)];

    messageCell.messageLabel.frame =
        CGRectMake(messageCell.messageLabel.frame.origin.x,
        messageCell.messageLabel.frame.origin.y,
        670, descriptionLabelHeight);

    if (currentItem[@"message"])
    {
        messageCell.toLabel.hidden = NO;
        messageCell.toLabel.text = currentItem[@"message"];
        messageCell.toLabel.frame =
            CGRectMake(messageCell.toLabel.frame.origin.x,
            messageCell.messageLabel.frame.origin.y +
            messageCell.messageLabel.frame.size.height + 5,
            messageCell.toLabel.frame.size.width,
            [self heightForString:messageCell.toLabel.text
            withFont:[UIFont italicSystemFontOfSize:15.0f]
            constrainedtoSize:CGSizeMake(670, 500)]);
    }
    else
    {
        messageCell.toLabel.hidden = YES;
    }

    messageCell.accessoryType =
        UITableViewCellAccessoryDisclosureIndicator;
    messageCell.selectionStyle =
        UITableViewCellStyleBlue;
    messageCell.userInteractionEnabled = YES;

    cell = messageCell;
```

This code does almost the same thing as the code for handling a status item, except for a few differences. `messageLabel` is populated with the item's title and `toLabel` is populated with the contents of the user description of the link.

If the user wrote no comment, then you hide `toLabel`. The link cell will always allow for selection and show a disclosure indicator, because the link can be opened inside the web view controller.

The logic is the same as for the message cell, except you take into consideration the differences between a status update and sharing a link.

Last but not least is the code to format a video or photo item cell. Paste the following below the // Code to format photo or video cell comment in the table view method:

```
PhotoCell *photoCell = [tableView
    dequeueReusableCellWithIdentifier:@"PhotoCell"];

photoCell.usernameLabel.text =
    currentItem[@"from"][@"name"];
photoCell.messageLabel.text = currentItem[@"name"];

CGFloat descriptionLabelHeight =
    [self heightForString:currentItem[@"name"]
        withFont:[UIFont systemFontOfSize:15.0f]
        constrainedtoSize:CGSizeMake(597, 500)];

photoCell.messageLabel.frame =
    CGRectMake(photoCell.messageLabel.frame.origin.x,
    photoCell.messageLabel.frame.origin.y,
    597, descriptionLabelHeight);

photoCell.accessoryType =
    UITableViewCellAccessoryDisclosureIndicator;
photoCell.selectionStyle =
    UITableViewCellStyleBlueBackground;
photoCell.userInteractionEnabled = YES;

dispatch_async(dispatch_get_global_queue(
    DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    NSString *pictureURL = currentItem[@"picture"];

    NSURL *imageURL = [NSURL URLWithString:pictureURL];

    __block NSData *imageData;

    dispatch_sync(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        imageData =
```

```
[NSData dataWithContentsOfURL:imageURL];

UIImage *pictureImage =
[UIImage imageWithData:imageData];

dispatch_sync(dispatch_get_main_queue(), ^{
    photoCell.pictureImageView.image =
        pictureImage;
});

cell = photoCell;
```

Again, it's the same logic as before, except that this time you download the thumbnail of the image to be shown in the cell as a small preview. All photo cell instances will allow for selection so that your users can view the video or photo directly on Facebook in the Web View Controller.

The table view data source methods are complete. ☺ Hooray!

Next, add the following method:

```
- (NSString *)toLabelStringFromArray:(NSArray *)toArray
{
    NSMutableString *toString =
    [NSMutableString stringWithString:@"To: "];

    for (int i = 0; i < toArray.count; i++)
    {
        [toString appendString:[toArray[i][@"name"]]];

        if (i < (toArray.count - 1))
        {
            [toString appendString:@", "];
        }
    }

    return [NSString stringWithFormat:@"%@", toString];
}
```

This is the method that takes care of concatenating an array of tagged users in a status item and returning a single string. It's very small and straightforward, but it helped reduce the size of `tableView:cellForRowAtIndexPath:` just a bit.

Next add a utility method to help calculate the height of a label for a given string, font and constrain size. It will be of great help inside `heightForCellAtIndex:` and will help you save many lines of code:

```
- (CGFloat)heightForString:(NSString *)theString withFont:(UIFont *)font constrainedtoSize:(CGSize)constrainSize
{
    CGSize stringHeight = [theString sizeWithFont:font
                                         constrainedToSize:constrainSize];

    return stringHeight.height;
}
```

The method uses the received parameters to call `sizeWithFont:constrainedToSize:` and returns the height of the `CGSize` object you receive.

Finally, add this method:

```
- (CGFloat)heightForCellAtIndex:(NSUInteger)index
{
    // 1
    NSDictionary *feedItem = self.feedArray[index];

    CGFloat cellHeight = 0.0;

    // 2
    if ([feedItem[@"type"] isEqualToString:@"status"])
    {
        cellHeight = 50;

        NSString *facebookText = feedItem[@"message"];

        CGSize messageLabelHeight = [facebookText
                                      sizeWithFont:[UIFont systemFontOfSize:15.0f]
                                      constrainedToSize:CGSizeMake(670, 500)];

        cellHeight += messageLabelHeight.height < 20 ? 20.0 :
            messageLabelHeight.height;

        NSArray *toArray = feedItem[@"to"][@"data"];

        if (toArray.count > 0)
        {
            NSString *toString =
                [self toLabelStringFromArray:toArray];

            CGSize toLabelHeight = [toString
```

```
        sizeWithFont:[UIFont italicSystemFontOfSize:15.0f]
        constrainedToSize:CGSizeMake(670, 500)];
    cellHeight += toLabelHeight.height < 20.0 ? 20.0 :
        toLabelHeight.height;
    }
}
// 3
else if ([feedItem[@"type"] isEqualToString:@"link"])
{
    cellHeight = 50;

    NSString *descriptionText = feedItem[@"name"];

    CGSize descriptionLabelHeight = [descriptionText
        sizeWithFont:[UIFont systemFontOfSize:15.0f]
        constrainedToSize:CGSizeMake(670, 500)];

    cellHeight += descriptionLabelHeight.height < 20 ? 20.0 :
        descriptionLabelHeight.height;

    if (feedItem[@"message"])
    {
        NSString *messageString = feedItem[@"message"];

        CGSize messageLabelHeight = [messageString
            sizeWithFont:[UIFont italicSystemFontOfSize:15.0f]
            constrainedToSize:CGSizeMake(670, 500)];
        cellHeight += messageLabelHeight.height < 20.0 ?
            20.0 : messageLabelHeight.height;
    }
}
// 4
else if ([feedItem[@"type"] isEqualToString:@"photo"] ||
    [feedItem[@"type"] isEqualToString:@"video"])
{
    cellHeight = 50;

    NSString *descriptionText = feedItem[@"name"];

    CGSize descriptionLabelHeight = [descriptionText
        sizeWithFont:[UIFont systemFontOfSize:15.0f]
        constrainedToSize:CGSizeMake(670, 500)];

    cellHeight += descriptionLabelHeight.height < 20 ? 20.0 :
        descriptionLabelHeight.height;
```

```
        cellHeight = cellHeight < 110 ? 110 : cellHeight;  
    }  
  
    return cellHeight;  
}
```

This is a utility method that's used heavily inside `tableView:cellForRowAtIndexPath:` to calculate the height of several labels, according to the current Facebook item type and contents. Here's what's happening:

1. You get the Facebook item for the current cell's index and create a variable to store and return the final height of the cell.
2. If the current item is a status update, then you calculate the cell's height based on the status message and the number of mentioned users. The fonts and constrain sizes are using the values set to match the label values from Interface Builder. (Change them here if your settings are different.)
3. When the Facebook item is a link, then you check for the user's message and the link title in order to calculate the cell's height.
4. Finally, if the item is a video or photo, you calculate the height of the description text and add it to the final cell size.

Running the app and checking your results

Phew! That was a lot of code you wrote back there, but you've done an amazing job. High five! As a reward, guess what time it is?

Yes, that's right, time to run the app and test your results!



I made stuff!

If you get any errors, then go back and check your code to verify that you copied and typed in everything correctly.

If you didn't get any errors, then for heaven's sake run the app and check out the results!



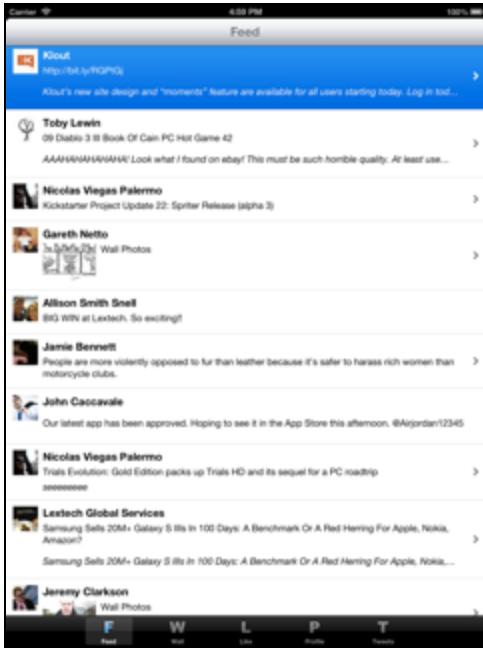
Also, tap the Wall tab bar icon and look at your personal wall items coming down. ☺



By sub-classing `FacebookFeedViewController` and overriding two methods, you are able to get all of the functionality you just created, but with different content. Neat, isn't it?

Any updates you make to the Feed View Controller will automatically be applied to the Wall View Controller, as well since the same code is executed for both.

There's one key element missing and you may have noticed it already:



If you tap on the cells with a disclosure indicator, you don't actually get a web browser window or see the comments. It's time to write that before calling the Feed and Wall View Controllers complete.

Showing comments and the web browser

All of the UI setup for the Comment View Controller has already been setup in the starter project. What remains is for you to write the controller logic and the `tableView:didSelectRowAtIndexPath:` method in `FacebookFeedViewController.m`.

Start with the former; go to **CommentViewController.h** and declare a property to store the array of comments:

```
@property (strong, nonatomic) NSArray *commentsArray;
```

Switch to **CommentViewController.m** and add the following import at the top:

```
#import "TwitterCell.h"
```

Add a class extension with a mutable dictionary property to cache the user profile images:

```
@interface CommentViewController()
@property (strong, atomic) NSMutableDictionary *imagesDictionary;
@end
```

Things are going well so far. It's time to write the controller's code, but before doing that, delete everything inside the implementation section and then add the following:

```

- (void)didReceiveMemoryWarning
{
    if ([self.view window] == nil)
    {
        _commentsArray = nil;
        _imagesDictionary = nil;
    }

    [super didReceiveMemoryWarning];
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.imagesDictionary = [NSMutableDictionary dictionary];
    self.title = @"Comments";
}

```

`didReceiveMemoryWarning` cleans up your variables. `viewDidLoad` sets the controller's title for the navigation bar item and creates and initializes `imagesDictionary` with an empty mutable dictionary.

For the table view data source, add these two methods:

```

- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return [self heightForCellAtIndex:indexPath.row];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return self.commentsArray.count;
}

```

The number of cells in the table is the number of comments inside `commentsArray`. For the height of each cell you will write another utility method called `heightForCellAtIndex:` to calculate the height of the row based on the amount of text.

Add that method now:

```

- (CGFloat)heightForCellAtIndex:(NSUInteger)index
{

```

```
NSDictionary *comment = self.commentsArray[index];

CGFloat cellHeight = 50;

NSString *message = comment[@"message"];

CGSize labelHeight = [message sizeWithFont:[UIFont
    systemFontOfSize:15.0f]
    constrainedToSize:CGSizeMake(700, 500)];

cellHeight += labelHeight.height;

return cellHeight;
}
```

This is exactly the same code you worked with back in `FacebookFeedViewController`. Heck, it's even simpler in here. The constraint font and sizes correspond to the values of the message label inside Interface Builder. Be sure to use your own values so things line up correctly.

Finally, add the following method:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    TwitterCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"TwitterCell"];

    NSDictionary *currentComment =
        self.commentsArray[indexPath.row];
    NSDictionary *currentUser = currentComment[@"from"];

    cell.usernameLabel.text = currentUser[@"name"];
    cell.tweetLabel.text = currentComment[@"message"];
    cell.tweetLabel.frame =
        CGRectMake(cell.tweetLabel.frame.origin.x,
        cell.tweetLabel.frame.origin.y,
        700, ([self heightForCellAtIndex:indexPath.row] - 50));

    NSString *userID = currentUser[@"id"];

    if (self.imagesDictionary[userID])
    {
        cell.userImageView.image =
            self.imagesDictionary[userID];
    }
}
```

```
else
{
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        NSString *pictureURL = [NSString stringWithFormat:
            @"%@/%@/picture?type=small",
            @"https://graph.facebook.com", userID];

        NSURL * imageURL = [NSURL URLWithString:pictureURL];

        __block NSData * imageData;

        dispatch_sync(dispatch_get_global_queue(
            DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
            imageData =
                [NSData dataWithContentsOfURL:imageURL];

            [self.imagesDictionary setObject:
                [UIImage imageWithData:imageData]
                forKey:userID];

            dispatch_sync(dispatch_get_main_queue(), ^{
                cell.userImageView.image =
                    self.imagesDictionary[userID];
            });
        });
    });
}

return cell;
}
```

This code should look familiar to you. Most of it is taken straight out of `FacebookFeedViewController`.

You set the label values, calculate their height, and either retrieve an image from the cache or download it asynchronously using Grand Central Dispatch.

The code for the Comment View Controller is done. There's just one more method to add to the Feed View Controller, and you'll be done with this monster of a section!

Open **FacebookFeedViewController.m** and add this method for the table view delegate:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // 1
    NSDictionary *selectedItem = self.feedArray[indexPath.row];

    // 2
    if ([selectedItem[@"type"] isEqualToString:@"status"])
    {
        NSDictionary *itemComments = selectedItem[@"comments"];

        int commentCount =
            ((NSArray *)itemComments[@"data"]).count;

        if (commentCount > 0)
        {
            CommentViewController *commentViewController =
                [self.storyboard
                    instantiateViewControllerWithIdentifier:
                    CommentViewControllerIdentifier];
            commentViewController.commentsArray =
                itemComments[@"data"];
            [self.navigationController
                pushViewController:commentViewController
                animated:YES];
        }
    }
    // 3
    else if ([selectedItem[@"type"] isEqualToString:@"link"] ||
              [selectedItem[@"type"] isEqualToString:@"photo"] ||
              [selectedItem[@"type"] isEqualToString:@"video"])
    {
        NSString * urlString = selectedItem[@"link"];

        WebViewController * webViewController =
            [self.storyboard
                instantiateViewControllerWithIdentifier:
                WebViewControllerIdentifier];
        webViewController.initialURLString = urlString;

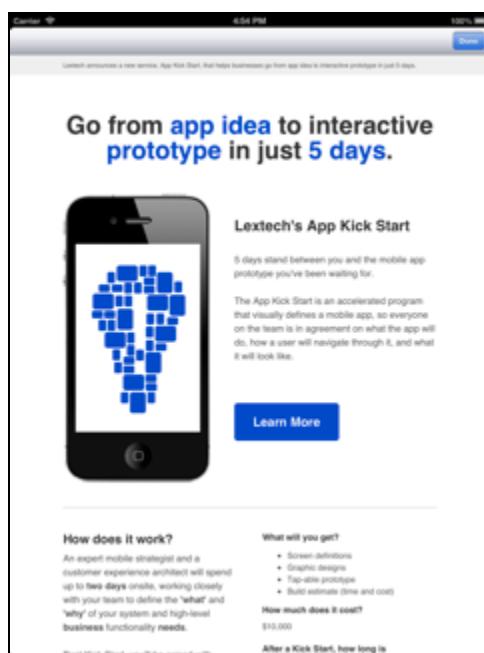
        [self presentViewController:webViewController
            animated:YES completion:nil];
    }
    // 4
    [self.tableView deselectRowAtIndexPath:indexPath]
```

```
    animated:YES];  
}
```

Here's what happening in the above method:

1. First you retrieve the feed object corresponding to the selected cell.
2. If the item is a status update, then you get the comments dictionary, make sure there's at least one comment and create an instance of `CommentViewController` to be pushed onto your navigation stack.
3. If the item is a link, photo or video, then you get the item's link and present it in a modal web view controller.
4. Finally, you deselect the selected row with an animation.

Run the app, select the Wall or Feed View Controller and tap on different items.





OK, there's one more view controller left to create! I hope you "like" it!

BA DUM TSS



Letting your users “like” an item

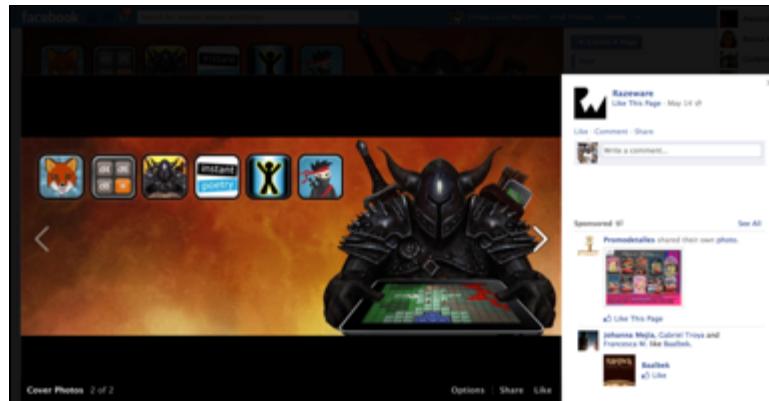
Let's get something out of the way: unfortunately, Facebook doesn't allow you to like a page using the Graph API. It's super silly I know and may change in the future, but it's a known limitation and something you are going to have to live with for the time being.

To know if you can like something, you need to look through the graph API documentation and check for the "likes" connection.

A photo object does allow for liking via the Graph API:

| Connections | | | |
|--|------------------------------------|--|--|
| The <code>Photo</code> object has the following connections. | | | |
| Name | Description | Permissions | Returns |
| <code>comments</code> | All of the comments on this photo. | any valid <code>access_token</code> or <code>user_photos</code> or <code>friends_photos</code> | array of objects containing <code>id</code> , <code>from</code> , <code>message</code> and <code>created_time</code> fields. |
| <code>likes</code> | Users who like this photo. | any valid <code>access_token</code> or <code>user_photos</code> or <code>friends_photos</code> | array of objects containing the <code>id</code> and <code>name</code> fields. |
| <code>picture</code> | The album-sized view of the photo. | any valid <code>access_token</code> or <code>user_photos</code> or <code>friends_photos</code> | HTTP 302 redirect to the URL of the picture |
| <code>tags</code> | The Users tagged in the photo. | any valid <code>access_token</code> or <code>user_photos</code> or <code>friends_photos</code> | Tags with names and IDs (if available). |

So how about you let users like the cover image of Razeware, the software company owned and operated by Ray?



In order to like an item, you need to know its ID. You can acquire the ID in one of two ways: either by going to the Graph API web console, or copying the ID from the URL of your browser.

Graph API | FQL Query

GET → /408881369146835

Submit ↗

Learn more about new features.

Node: 408881369146835

```
{
  "id": "408881369146835",
  "from": {
    "name": "Razeware",
    "category": "Games/Toys",
    "id": "132723039429312"
  },
  "picture": "https://fbcdn-photos-a.akamaihd.net/hphotos-ak-prn1/524119_408881369146835_122723039429312_n.jpg?width=250&height=250",
  "source": "https://fbcdn-photos-a.akamaihd.net/hphotos-ak-prn1/x720x720/526119_408881369146835_n.jpg?width=250&height=250",
  "height": 244,
  "width": 229,
  "images": [
    {
      "height": 794,
      "width": 2048,
      "source": "https://fbcdn-photos-a.akamaihd.net/hphotos-ak-prn1/x2048x2048/526119_n.jpg?width=2048&height=794"
    },
    {
      "height": 315,
      "width": 850,
      "source": "https://fbcdn-photos-a.akamaihd.net/hphotos-ak-prn1/x720x720/526119_n.jpg?width=850&height=315"
    },
    {
      "height": 244,
      "width": 729,
      "source": "https://fbcdn-photos-a.akamaihd.net/hphotos-ak-prn1/x720x720/526119_n.jpg?width=729&height=244"
    },
    {
      "height": 231,
      "width": 650,
      "source": "https://fbcdn-photos-a.akamaihd.net/hphotos-ak-prn1/x650x650/526119_n.jpg?width=650&height=231"
    },
    {
      "height": 177,
      "width": 500,
      "source": "https://fbcdn-photos-a.akamaihd.net/hphotos-ak-prn1/x500x500/526119_n.jpg?width=500&height=177"
    }
  ]
}
```

<https://www.facebook.com/photo.php?fbid=408881369146835&set=a.408881349146837.100803.122723294429312&type=1>

We'll go with the latter, but if you want to show the Razeware cover image by itself or acquire more info, then check out the console so you can get those parameters.

I've gone ahead and copied the URL for the image so you can load it in your view controller.

Go to **LikeViewController.m** and add the following import and define statements:

```
#import "AppDelegate.h"

#define PhotoGraphURL
@"https://graph.facebook.com/408881369146835/likes"
#define PhotoURL
@"https://fbcdn-sphotos-e-a.akamaihd.net/hphotos-ak-
prn1/s720x720/526119_408881369146835_2134350025_n.jpg"
```

You need to import the app delegate's header so that you can access the Facebook account for the current user. The defined strings are the links to where you can download the cover image for Razeware, and the graph path for liking and un-liking the photo.

Next add two new properties to the class extension:

```
@property (strong) NSString * userID;
@property (assign) BOOL userLikesPhoto;
```

Now add the functionality necessary for everything to work. Remove the existing placeholder for `didReceiveMemoryWarning` and add these two methods:

```
- (void) didReceiveMemoryWarning
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];

    if ([self.view window] == nil)
    {
        _coverImageView = nil;
        _infoLabel = nil;
        _likeButton = nil;
    }

    [super didReceiveMemoryWarning];
}

- (void)viewWillDisappear:(BOOL)animated
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];

    [super viewWillDisappear:animated];
}
```

didReceiveMemoryWarning cleans things up as usual and, along with viewWillDisappear:, removes the controller as a notification observer.

Now add the implementation for viewDidLoad:

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

    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        NSURL *imageURL = [NSURL URLWithString:PhotoURL];

        __block NSData *imageData;

        dispatch_sync(dispatch_get_global_queue(
            DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
            imageData = [NSData dataWithContentsOfURL:imageURL];

            dispatch_sync(dispatch_get_main_queue(), ^{
                self.coverImageView.image =
                    [UIImage imageWithData:imageData];
            });
        });
    });

    SLRequest *request = [SLRequest
        requestForServiceType:SLServiceTypeFacebook
        requestMethod:SLRequestMethodGET
        URL:[NSURL URLWithString:@"https://graph.facebook.com/me"]
        parameters:nil];
    AppDelegate *appDelegate =
        [[UIApplication sharedApplication] delegate];
    request.account = appDelegate.facebookAccount;

    [request performRequestWithHandler:^(NSData *responseData,
        NSHTTPURLResponse *urlResponse, NSError *error) {
        if (error)
        {
            AppDelegate *appDelegate = (AppDelegate *)
                [[UIApplication sharedApplication] delegate];

            [appDelegate presentErrorWithMessage:[NSString
                stringWithFormat:@"There was an error getting the user's ID. %@", [error
                localizedDescription]]];
        }
    }];
}
```

```
    }
    else
    {
        NSError *jsonError;
        NSDictionary *responseJSON = [NSJSONSerialization
            JSONObjectWithData:responseData
            options:NSJSONReadingAllowFragments
            error:&jsonError];

        if (jsonError)
        {
            AppDelegate *appDelegate = (AppDelegate *)
                [[UIApplication sharedApplication] delegate];

            [appDelegate presentErrorWithMessage:[NSString
                stringWithFormat:@"There wasn't an error reading the user's ID. %@", error
                localizedDescription]]];
        }
        else
        {
            self.userID = responseJSON[@"id"];
        }
    }
};

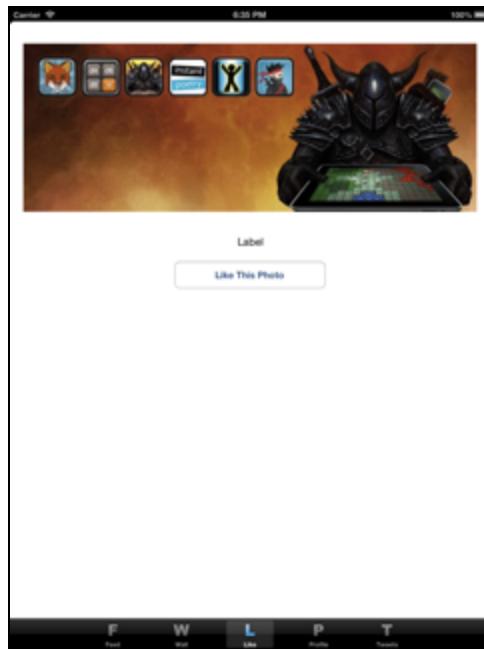
}
```

Inside the method you download the image in the URL constant defined earlier and place it in the controller's image view.

You also perform a request to the Graph API in order to get the user's info and retrieve the Facebook account ID. You need this ID to check if a user has liked the photo or not.

Most of this code is a straight copy-and-paste from previous examples – it just turns out to be quite long because of all the error checks.

Take a break to build and run the app, and then switch to the Like tab. You should see something similar to the following:



Next add the following:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(updatePhotoStatus)
        name:AccountFacebookAccountAccessGranted object:nil];

    AppDelegate *appDelegate =
        [[UIApplication sharedApplication] delegate];

    if (appDelegate.facebookAccount)
    {
        [self updatePhotoStatus];
    }
    else
    {
        self.likeButton.enabled = NO;

        AppDelegate *appDelegate =
            [[UIApplication sharedApplication] delegate];
        [appDelegate getFacebookAccount];
    }
}
```

When the view first appears, you register for the `AccountFacebookAccountAccessGranted` notification. Then you check to see if the user's Facebook account has already been retrieved.

If there's an account, then you update the photo status (which will update the label's content and the button's state) by calling `updatePhotoStatus`. Otherwise you ask the app delegate to get the user's Facebook account.

Add the code for `updatePhotoStatus` next:

```
- (void)updatePhotoStatus
{
    // 1
    SLRequest *request = [SLRequest
        requestForServiceType:SLServiceTypeFacebook
        requestMethod:SLRequestMethodGET
        URL:[NSURL URLWithString:PhotoGraphURL]
        parameters:nil];

    AppDelegate *appDelegate =
        [[UIApplication sharedApplication] delegate];
    request.account = appDelegate.facebookAccount;

    [request performRequestWithHandler:^(NSData *responseData,
                                         NSHTTPURLResponse *urlResponse, NSError *error) {
        if (error)
        {
            // 2
            AppDelegate *appDelegate = (AppDelegate *)
                [[UIApplication sharedApplication] delegate];

            [appDelegate presentErrorWithMessage:[NSString
                stringWithFormat:@"There was an error getting the status of the
                photo's likes. %@", [error localizedDescription]]];
        }
        else
        {
            // 3
            NSError *jsonError;
            NSDictionary *responseJSON = [NSJSONSerialization
                JSONObjectWithData:responseData
                options:NSJSONReadingAllowFragments
                error:&jsonError];

            if (jsonError)
            {

```

```
// 4
AppDelegate *appDelegate = (AppDelegate *)
    [[UIApplication sharedApplication] delegate];

    [appDelegate presentErrorWithMessage:[NSString
stringWithFormat:@"There was an error reading the photo's status.
%@", [error localizedDescription]]];
}

else
{
// 5
NSString *userID = self.userID;
NSArray *likes = responseJSON[@"data"];

for (NSDictionary *user in likes)
{
    if ([user[@"id"] isEqualToString:userID])
    {
        self.userLikesPhoto = YES;

        break;
    }
}

// 6
self.likeButton.enabled = YES;

if (self.userLikesPhoto)
{
    dispatch_sync(dispatch_get_main_queue(), ^{
        self.infoLabel.text =
@"You have already liked this picture";
        [self.likeButton setTitle:
 @"Unlike This Photo"
        forState:UIControlStateNormal];
    });
}
else
{
    dispatch_sync(dispatch_get_main_queue(), ^{
        self.infoLabel.text = @"You haven't liked
the picture. Tap the button to like it";
        [self.likeButton setTitle:@"Like This
Photo" forState:UIControlStateNormal];
    });
}
```

```
        }
    }
}];
```

`updatePhotoStatus` checks whether or not the logged-in user has already liked the photo. The request URL is the same to check, like, or unlike the photo – the difference lies in the HTTP method you use.

“Hey Felipe, break it down for me one last time!” OK, then:

1. You create the request using the photo’s graph URL, set no parameters, and use the GET HTTP method.
2. After the request completes, you check for errors and present an alert view if there were any.
3. If no request errors occurred, you parse the response data.
4. Should an error occur when parsing the JSON data, you present an alert view to the user.
5. Otherwise, you look through all of the users who liked the photo and compare their IDs to your user’s ID (the one you downloaded inside `viewDidLoad`). If there’s a match, you set the flag indicating that the user liked the photo and break out of the loop.
6. When you finish checking the status of the photo, you re-enable the Like button and update the label’s text and the button’s title, depending on whether the user already liked the image.

By now you probably know this code better than I do; you’re a master of `SLRequests` and the Social framework!

Just one more method and iSocial is complete. ☺ Add the implementation for `likeTapped` as shown:

```
- (IBAction)likeTapped
{
    self.likeButton.enabled = NO;

    if (self.userLikesPhoto)
    {
        SLRequest *request = [SLRequest
            requestForServiceType:SLServiceTypeFacebook
            requestMethod:SLRequestMethodDELETE
            URL:[NSURL URLWithString:PhotoGraphURL]
            parameters:nil];
    }

    AppDelegate *appDelegate =
```

```
[[UIApplication sharedApplication] delegate];
request.account = appDelegate.facebookAccount;

[request performRequestWithHandler:
^(NSData *responseData, NSHTTPURLResponse *urlResponse,
NSError *error) {
    dispatch_sync(dispatch_get_main_queue(), ^{
        self.likeButton.enabled = YES;
    });

    if (error)
    {
        AppDelegate *appDelegate = (AppDelegate *)
            [[UIApplication sharedApplication] delegate];

        [appDelegate presentErrorWithMessage:[NSString
stringWithFormat:@"There was an error unliking the photo. %@", [error
localizedDescription]]];
    }
    else
    {
        NSError *jsonError;
        id responseJSON = [NSJSONSerialization
JSONObjectWithData:responseData
options:NSUTF8StringEncoding
error:&jsonError];

        if (jsonError)
        {
            AppDelegate *appDelegate = (AppDelegate *)
                [[UIApplication sharedApplication] delegate];

            [appDelegate presentErrorWithMessage:[NSString
stringWithFormat:@"There was an error unliking the photo. %@", [error
localizedDescription]]];
        }
        else
        {
            if ([responseJSON intValue] == 1)
            {
                self.userLikesPhoto = NO;

                dispatch_sync(dispatch_get_main_queue(),

```

```
self.infoLabel.text = @"You haven't liked the picture. Tap the
button to like it";

[self.likeButton setTitle:@"Like This Photo"
forState:UIControlStateNormal];
    });
}
}
}
}];

}
else
{
    SLRequest *request = [SLRequest
        requestForServiceType:SLServiceTypeFacebook
        requestMethod:SLRequestMethodPOST
        URL:[NSURL URLWithString:PhotoGraphURL]
        parameters:nil];

    AppDelegate *appDelegate =
        [[UIApplication sharedApplication] delegate];
    request.account = appDelegate.facebookAccount;

    [request performRequestWithHandler:^(NSData *responseData,
    NSHTTPURLResponse *urlResponse, NSError *error) {
        dispatch_sync(dispatch_get_main_queue(), ^{
            self.likeButton.enabled = YES;
        });

        if (error)
        {
            AppDelegate *appDelegate = (AppDelegate *)
                [[UIApplication sharedApplication] delegate];

            [appDelegate presentErrorWithMessage:[NSString
                stringWithFormat:@"There was an error liking the photo. %@", [error
                localizedDescription]]];
        }
        else
        {
            NSError *jsonError;
            id responseJSON = [NSJSONSerialization
                JSONObjectWithData:responseData
                options:NSJSONReadingAllowFragments error:&jsonError];

            if (jsonError)
```

```
        {
            AppDelegate *appDelegate = (AppDelegate
*)[[UIApplication sharedApplication] delegate];

            [appDelegate presentErrorWithMessage:[NSString
stringWithFormat:@"There was an error liking the photo. %@",

[error localizedDescription]]];
        }
        else
    {
        if ([responseJSON intValue] == 1)
        {
            self.userLikesPhoto = YES;

            dispatch_sync(dispatch_get_main_queue(),
^{
                self.infoLabel.text = @"You have
already liked this picture";
                [self.likeButton setTitle:@"Unlike
This Photo" forState:UIControlStateNormal];
            });
        }
    }
};

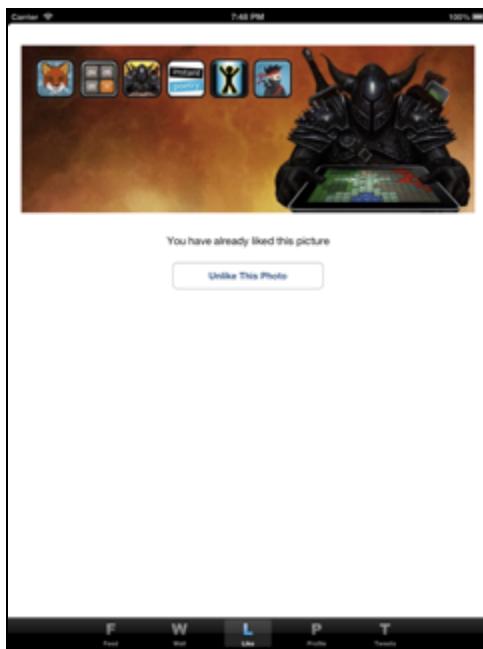
}
}
```

Though the code for the method is long, it's mostly error checking for the SLRequests.

To summarize things, what the method does is check the state of the userLikesPhoto flag and either like or unlike the photo. The request code is identical to what you've used before, except that to like the photo you use an HTTP POST request and to unlike you use an HTTP DELETE.

This is the beauty of REST APIs, and it makes interacting with resources quite fun. When the operation returns, the response will be a true or false value. You check this to see if the operation was successful and update the button's title and info label accordingly.

Build and run the project one last time. Select the Like tab and look at the results:



Hooray! Give yourself a huge high five for making it to the end of the chapter. You've come a long way toward mastering the Social framework, and the Accounts framework to boot! You're ready to create beautiful apps that interact with Twitter, Facebook, or Sina Weibo.

Where to go from here?

Facebook has a huge API and iSocial just scratches the surface of what you can do with it. If you want to learn more about the Facebook or Twitter APIs, iSocial provides you with a great starting point that you can build on to play around some more. Here are some ideas for what you could do next:

- Turn iSocial into a hybrid client for Twitter, Facebook, and Sina Weibo.
- Expand iSocial into a full Facebook client.
- Explore some of the other API calls you can make with the Twitter and Facebook APIs that seem interesting to you.
- Allow users to post to Twitter and Facebook.

Whatever you decide to do, you are the winner. The Social framework is very easy to master and the real challenge is in understanding and using the various social networks' APIs.

I'm very happy that Apple has added native support for Facebook and Sina Weibo this year (in addition to Twitter). I can't wait to see what you guys create!

Chapter 13: Beginning Challenges with GameKit

By Kauserali Hafizji

You've probably heard of Game Center, the online multiplayer social gaming network introduced in iOS 4.1. It allows users to invite friends to play a game, start a multiplayer gaming session, track game achievements, and a lot more.

In addition to making implementing these standard social features much easier for developers, it also helps with a fundamental problem for developers: app discovery. With over 1 million apps in the App Store today, the odds of a single user discovering your app can be frustratingly low. Game Center helps to solve this problem by allowing the user to see what games his/her friends are playing, thereby increasing the visibility of your game.

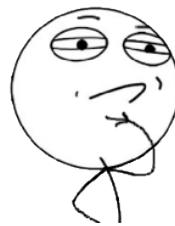
iOS 6.0 introduces new APIs in Game Center that not only continue to help your game gain visibility, but also increase user engagement. One such feature is the ability to send game challenges to friends, even those who don't have the game installed. For example, a user could get a high score in your game and send a challenge to their friend saying "Hah, beat this!"

When the friend receives the challenge, they will see the message along with a direct link to download your app. It's not hard to imagine how this could increase user retention manyfold! When you consider the large number of players using Game Center, this alone is a very good reason to add challenges to your games.

Before you can use challenges you need to use Game Center, so first this chapter will take you through setting up Game Center, highlighting all the iOS 6 updates along the way. These include a new view controller that lets players interact with Game Center from right within your app, and the ability to share scores and achievements through social networks.

Then you'll spend the last third of this chapter working with challenges, Game Center's major new feature. You will learn the process behind challenges and fully enable them in a game. By the end, you'll be ready for more advanced features of the challenges API, which is the subject of the next chapter. ☺

CHALLENGE CONSIDERED



Best of all, for “testing” purposes you get to play a fun scroller game we developed for this chapter! The game is called MonkeyJump, and may be familiar to readers of raywenderlich.com.

So if you’re ready to accept your first in a series of challenges, let’s meet the monkey!

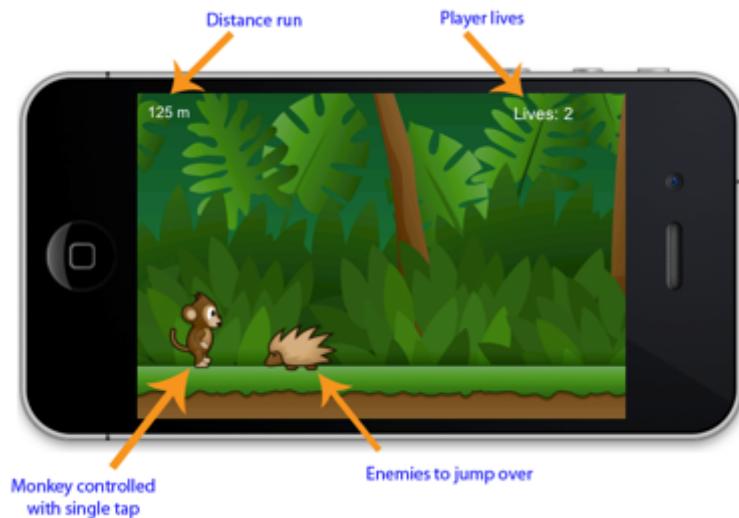
Note: This chapter assumes that you have prior knowledge of Cocos2D and have built games using it. If you are new to Cocos2D, you can read some of the many Cocos2D tutorials available on raywenderlich.com.

The MonkeyJump game

MonkeyJump is a simple side scroller written using my favorite game engine, Cocos2D. It was originally based on a simple game called CatJump developed for a Learning Cocos2D workshop, but I added some new features to make it more fun and added some cute new art from Vicki Wenderlich.

The main character of the game is, obviously, a monkey. ☺ The objective of the game is to make the monkey to cover the maximum possible distance while avoiding as many enemies as he can.

MonkeyJump is easy to play – even your mom could play! The player simply has to tap the screen to make the monkey jump over enemies. The game keeps track of the distance the monkey runs and uses it to determine the player’s score.



Try it out for yourself! In the resources for this chapter you'll find a "MonkeyJump Starter" project – open it in Xcode and build and run. Try to see how far you can get! ☺

After you're done playing around, take a quick tour through the code to get a basic understanding of the different scenes and layers used. The game consists of four scenes:

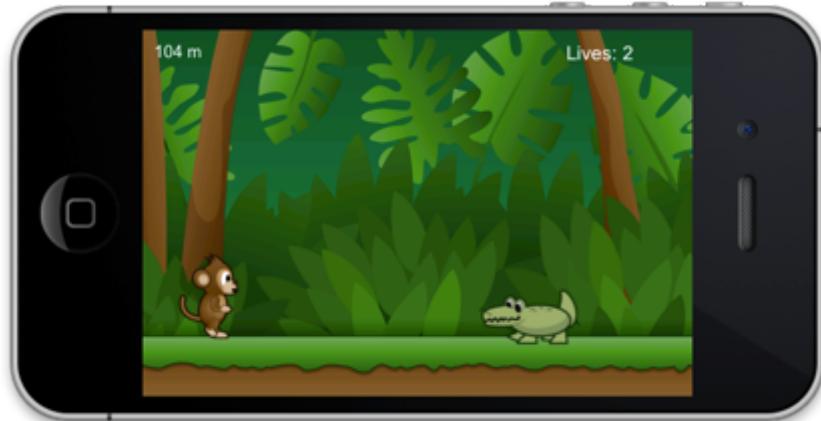
- **GameIntroScene/GameIntroLayer:** This is the first scene presented to the player, and is used to show an introductory image.



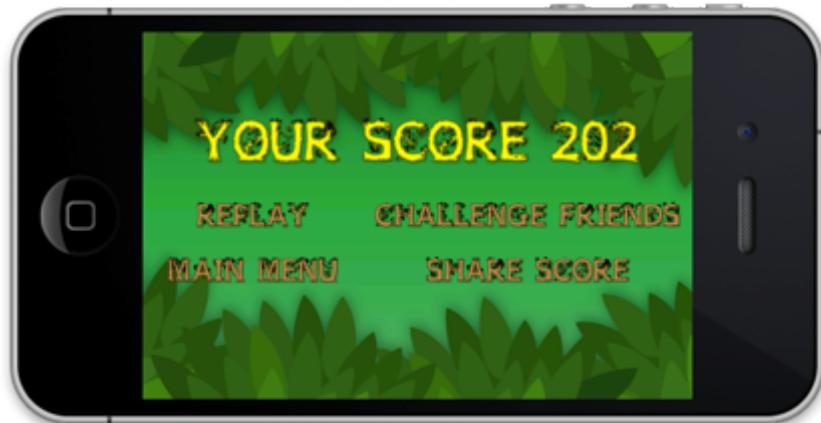
- **MenuScene/MenuLayer:** This scene displays a menu to the player a few seconds after the GameIntroScene. The player can either choose to play a new game or visit Game Center to view leaderboards/achievements. Tapping Game Center does nothing at the moment though – that's your job in this chapter!



- **GameScene/GameLayer:** This is the actual game and is presented when the player taps on the New Game button on the MenuScene. This is the most important class gameplay-wise – take a peek if you’re curious how the game works. It’s actually quite simple!



- **GameOverScene:** As the name suggests, this scene is presented when the player finishes playing the game. It allows the player to replay the game, share his/her score, go back to the main menu or challenge friends.



Setting up Game Center

As mentioned earlier in this chapter, before you can do anything with Game Center Challenges, you first have to set up your app to use Game Center! And to do that, you first need to do three things:

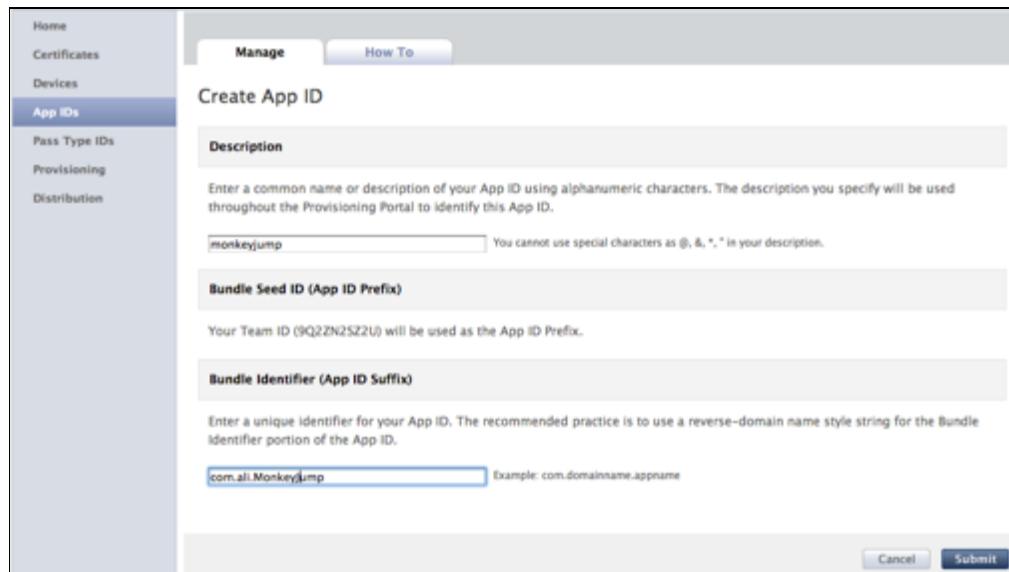
1. Create and set an App ID.
2. Register your app on iTunes Connect.
3. Enable Game Center features, leaderboards and achievements.

Let's go through each of these steps in turn. This will be old hat for many readers, but I promise we'll go through these necessary tasks quickly.

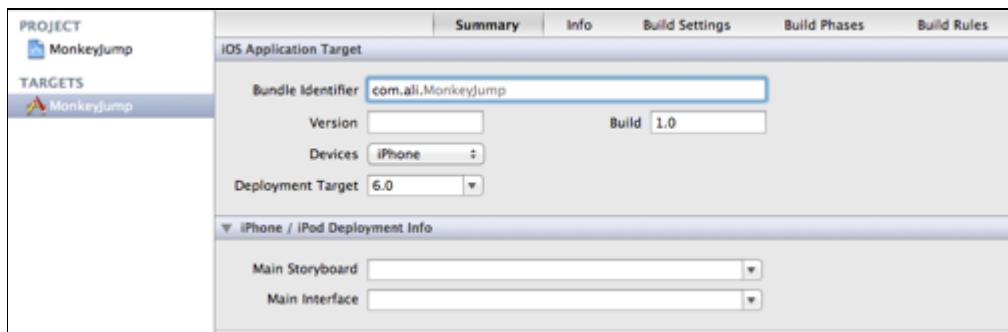
Creating and setting an App ID

The first thing you need to do is create an App ID. To do this, log on to the iOS Dev Center and from there, select the iOS Provisioning Portal.

From the Provisioning Portal, select App IDs and create a new App ID. Use **monkeyjump** as the name and enter a bundle identifier – usually it's good to use reverse DNS notation for a domain you control like com.ali.MonkeyJump (you can use your name if you don't have a domain name).



Once you are done, click the **Submit** button. Open the MonkeyJump Xcode project, select project root, then the MonkeyJump target (if it's not selected), and in the Summary tab change the Bundle Identifier to the identifier you created in the iOS Provisioning Portal.

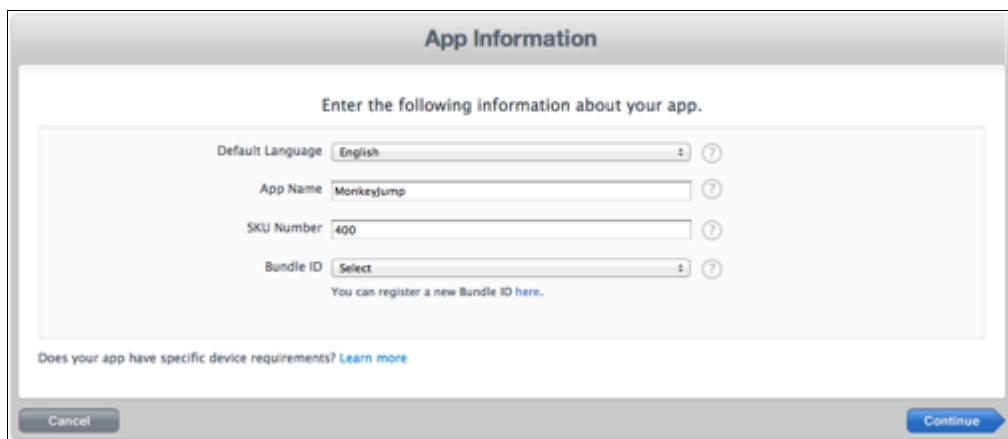


Give the app a quick compile and run it on a device. If everything is in place the game will start right up. If it doesn't, clean and build the project and try again.

Registering your app in iTunes Connect

The next step is to create a new app on iTunes Connect. Log in to iTunes Connect, switch to the application management screen, and click the **Add New App** button in the top left corner. (If you have both Mac and iOS developer accounts, you might have to select the type of app – which is iOS, of course.)

On the first screen, enter **MonkeyJump** as the game name, **400** as the SKU number (this can be any number/word, so if you want you can set it to something else) and select the Bundle Identifier you created in the previous step.



When you are done entering all the values, press **Continue**. Follow the prompts and enter all the required details. Since you just need to get through these steps for the purposes of this chapter, fill in only the necessary values and be as brief as you want to be. ☺

You will need to upload a large app icon and a screenshot. To make the process easier, I have added an iTunes.zip file to the chapter resources that includes the files you'll need. You can extract the ZIP file and upload these to quickly finish the registration process.

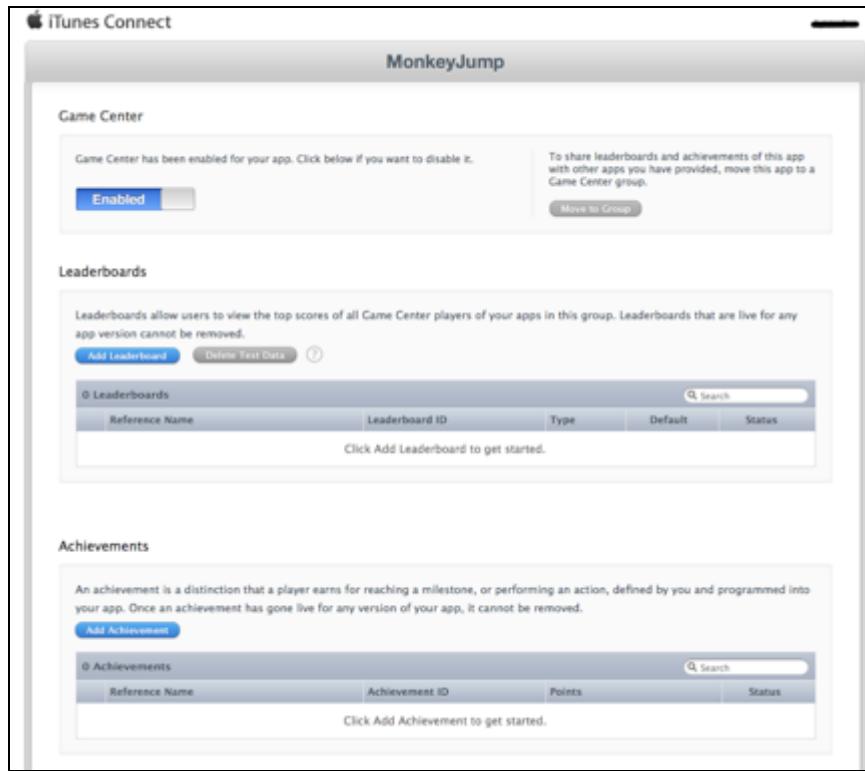
When you're done, click on the Save button and if everything went fine, you will be presented with the following screen:

Hurray! You have registered your app with iTunes Connect and completed the most perfunctory business. Now there are just a few more steps to activate Game Center. Don't worry, the tough part is over. ☺

Enabling Game Center features

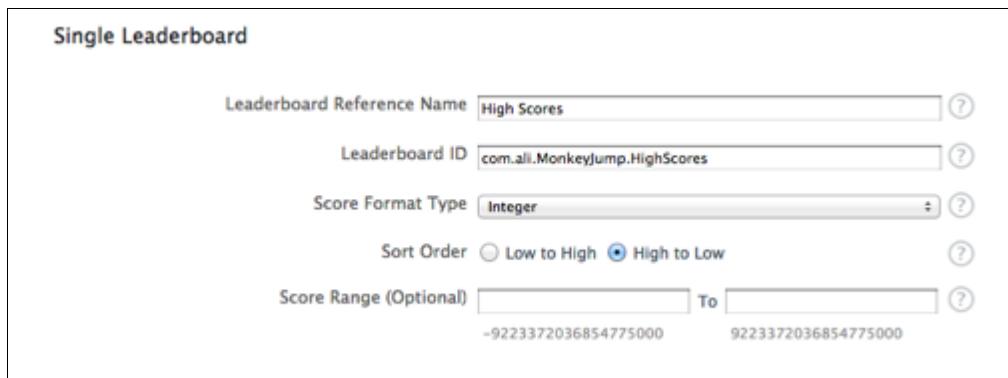
Click the blue **Manage Game Center** button and then click the **Enable for Single Game** button. Awesome! You have just enabled Game Center for your game. Yep, it is as simple as clicking a button – but you still have a bunch of code to write ☺

Note: If you're curious about the Enable for Group of Games button, as the name implies, it allows you to enable Game Center for a group of games that share the same leaderboards and achievements. This way, you can have bunch of apps with shared scores.



You aren't done with this section yet – you still need to add leaderboards and achievements. You might wonder why you have to bother with leaderboards and achievements since this chapter is about challenges – don't worry, you will see why later!

For now, take my word at it and add a leaderboard and some challenges, starting with a leaderboard. Click the **Add Leaderboard** button and select the **Single Leaderboard** type. You will be presented with a form, like so:



| | |
|----------------------------|--|
| Leaderboard Reference Name | High Scores |
| Leaderboard ID | com.ali.MonkeyJump.HighScores |
| Score Format Type | Integer |
| Sort Order | <input checked="" type="radio"/> High to Low |
| Score Range (Optional) | -9223372036854775000 To 9223372036854775000 |

Enter the leaderboard reference name as **High Scores** and the leaderboard ID as **HighScores**.

Note: I generally recommend you keep the leaderboard/achievement ID as an extension of the package name. For example, in the above case for me it would be com.ali.MonkeyJump.HighScores (you would need to replace the com.ali part to match your own setup). But for the purposes of this chapter, just name it plain **HighScores** (without the reverse domain name prefix) to keep things simple.

Set the **Sort Order** as **High to Low** and the **Score Format Type** as **Integer**. Finally, click on the **Add Language** button. Enter the language details as shown below:



Adding an image is not mandatory, but is always a good practice. The one used above is included in the iTunes ZIP file (it's named icon_leaderboard_512.png), and you can use it here for the high scores leaderboard. When you're done, click **Save**.

Finally, click on the Done button. For now, one leaderboard is enough, but in the future if you want to add more, you now know the drill.

Note: Wondering why you might want more than one leaderboard? Maybe you want to have different leaderboards for easy mode and hard mode, or a different leaderboard for each level. It's up to your own creativity how you configure your leaderboards!

Next, add a few achievements to the app. Click the **Add Achievements** button on the Manage Game Center screen. The three achievements you are going to create are:

- **Novice Runner:** Achieved when the player completes a distance of 200 meters.
- **Athlete:** Achieved when the player completes a distance of 1000 meters.
- **2000 meters:** Achieved when the player completes 2000 meters.

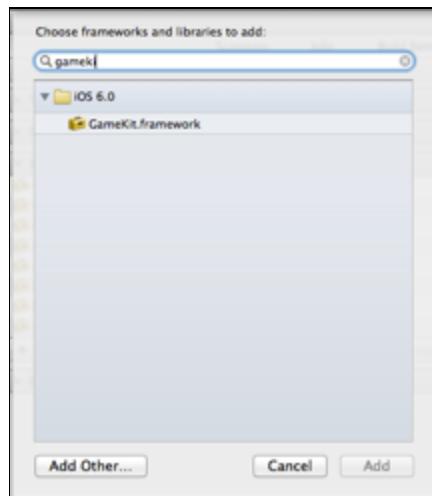
Creating achievements is similar to creating a leaderboard, so I won't go into details again. Just be sure while creating the above achievements to keep their respective achievement IDs as **NoviceRunner**, **Athlete** and **2000meters**, so that you'll be in sync with the code. You will find achievement icons for each in the iTunes.zip file; feel free to use them when you create each achievement.

Note: There are a few values for achievements that you might not be sure about: Point Value, Hidden, and Achievable More Than Once. It doesn't really matter what you put for these for the purposes of this chapter. ☺ If you want more information, please click the help icon next to each field to get a detailed description.

That's it! You have enabled Game Center features, achievements and leaderboards. Now it's time to write some code to use those features!

Authenticating the local player

Before you start writing code, you need to import the GameKit framework. Open the MonkeyJump project in Xcode 4.5 and navigate to the target settings. Open the **Build Phases** tab and expand the **Link Binary With Libraries** section. Click the "+" button and add the GameKit framework to the project.



Next you will write code to authenticate the player. Without authenticating the player, you cannot use any of the awesome features that Game Center provides.

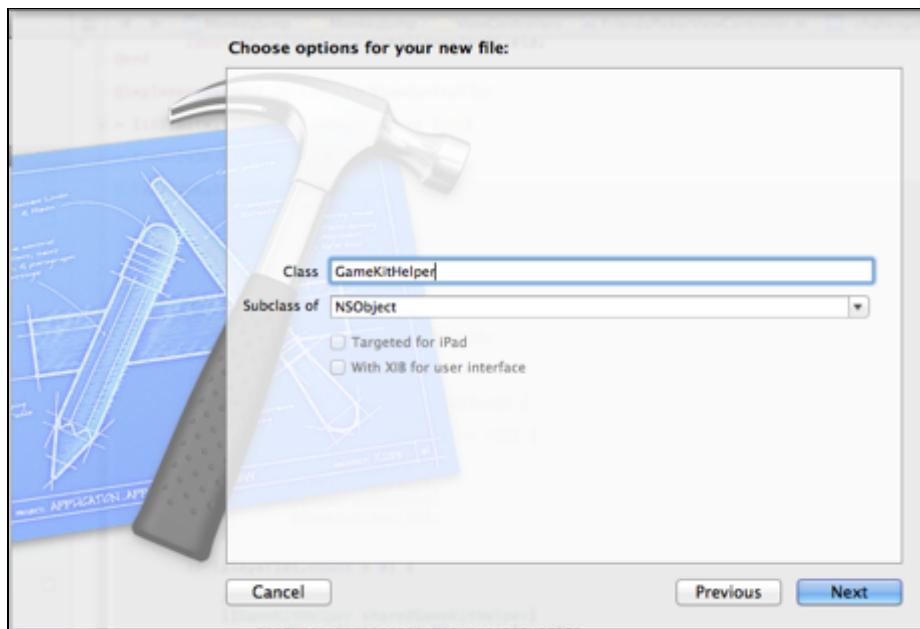
Player here means the user who is playing your game. In Game Center terms, this is the `GKLocalPlayer`.

Player authentication is a simple two-step process:

1. First you make an authenticate call to the Game Center platform.
2. The platform then asynchronously calls you back when authentication is complete. If the player was already logged in (95% of the time), a welcome banner is presented. If not, then a login screen is presented which also allows the player to register.

Let's write some code. You are going to use a singleton pattern, so that all the Game Center code is in one class.

Right-click on the MonkeyJump group in Xcode and select **New Group**. Name the group **GameKitFiles**. Next, right-click on the newly-created group and select **New File...**, then select the **Objective-C Class** template. Name the class **GameKitHelper** and make sure it extends `NSObject`.



Replace the contents of **GameKitHelper.h** with the following:

```
//  Include the GameKit framework
#import <GameKit/GameKit.h>

//  Protocol to notify external
//  objects when Game Center events occur or
//  when Game Center async tasks are completed
@protocol GameKitHelperProtocol<NSObject>
-(void) onAchievementsLoaded:(NSDictionary*)achievements;
@end
```

```
@interface GameKitHelper : NSObject

@property (nonatomic, assign)
    id<GameKitHelperProtocol> delegate;

// This property holds the last known error
// that occurred while using the Game Center API's
@property (nonatomic, readonly) NSError* lastError;

// This property holds Game Center achievements
@property (nonatomic, readonly)
    NSMutableDictionary* achievements;

+ (id) sharedGameKitHelper;

// Player authentication, info
-(void) authenticateLocalPlayer;
@end
```

The code is self-explanatory and is heavily commented. All you are doing here is declaring two methods and two properties – one is the delegate, and the other will hold the last error that occurred while using the GameKit framework.

Switch to **GameKitHelper.m** and replace its contents with the following:

```
#import "GameKitHelper.h"
#import "GameConstants.h"

@interface GameKitHelper ()<GKGameCenterControllerDelegate> {
    BOOL _gameCenterFeaturesEnabled;
}

@implementation GameKitHelper

#pragma mark Singleton stuff

+(id) sharedGameKitHelper {
    static GameKitHelper *sharedGameKitHelper;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedGameKitHelper =
            [[GameKitHelper alloc] init];
    });
}
```

```
    });
    return sharedGameKitHelper;
}

#pragma mark Player Authentication

-(void) authenticateLocalPlayer {

    GKLocalPlayer* localPlayer =
    [GKLocalPlayer localPlayer];

    localPlayer.authenticateHandler =
    ^(UIViewController *viewController,
       NSError *error) {

        [self setError:error];

        if ([[CCDirector sharedDirector].isPaused)
            [[CCDirector sharedDirector] resume];

        if (localPlayer.authenticated) {
            _gameCenterFeaturesEnabled = YES;
        } else if(viewController) {
            [[CCDirector sharedDirector] pause];
            [self presentViewController:viewController];
        } else {
            _gameCenterFeaturesEnabled = NO;
        }
    };
}
@end
```

You have declared a variable called `_gameCenterFeaturesEnabled`. This BOOL variable will be true if authentication was successful, and in any other case will be false.

The way to authenticate a player has changed in iOS 6.0. All you need to do now is set the `authenticationHandler` of the `GKLocalPlayer` object, as seen in the `-authenticateLocalPlayer` method. The `authenticationHandler` is a block that takes two parameters and is called by the Game Center platform.

This block is called by the system on a number of scenarios:

- When you set the `authenticationHandler` and request for the player to be authenticated.
- When the app moves to the foreground.

- On sign-in, i.e., if the player has not signed in before a sign-in screen is presented, any interaction on that screen leads to the `authenticationHandler` being called.

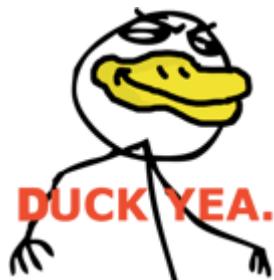
The `authenticationHandler` has two arguments:

- A `UIViewController` representing the login view controller if you need to present it; and
- An `NSError` object in case authentication happens to fail.

Notice that in the block, you first check if the local player is authenticated. If the player has already been authenticated, then all you need to do is set the `_gameCenterEnabled` flag as true and get on with your game.

If the login view controller (i.e., the first parameter of the `authenticationHandler`) is not `nil`, it means that the player has not logged into Game Center. If so, you first pause the game and then present the login view controller to the player. If the player logs in or selects the Cancel button on the login view controller, this handler is called again.

In earlier versions of Game Center, the developer did not have the power to decide when to present the login view controller. This new method gives developers more control, which is always a good thing. ☺



Finally, if authentication fails you need to gracefully fall back and disable all Game Center features. This is achieved in this app by setting the `_gameCenterFeaturesEnabled` flag to false.

In order for the code in `authenticateLocalPlayer` to work, you need a few more bits of code. Add the following to `GameKitHelper.m`:

```
#pragma mark Property setters

-(void) setError:(NSError*)error {
    _lastError = [error copy];
    if (_lastError) {
        NSLog(@"GameKitHelper ERROR: %@", [_lastError userInfo]
              description);
    }
}
```

```
#pragma mark UIViewController stuff

-(UIViewController*) getRootViewController {
    return [UIApplication
        sharedApplication].keyWindow.rootViewController;
}

-(void)presentViewController:(UIViewController*)vc {
    UIViewController* rootVC = [self getRootViewController];
    [rootVC presentViewController:vc animated:YES
        completion:nil];
}
```

The above three methods set up a few things needed by `authenticateLocalPlayer`:

1. The `lastError` property is declared as a `readonly` property. Hence, you cannot assign to it directly. So you need a setter method that will take care of setting the `lastError` property. That's what `setLastError:` is.
2. The Game Center login controller needs to be displayed to the user so that s/he can actually login. The `presentViewController:` and `getRootViewController` methods handle discovering the root view controller for the application and then displaying the login view via the root view controller.

Awesome! Now it's time to put `GameKitHelper` to the test. Open **Prefix.pch** and add the necessary import:

```
#import "GameKitHelper.h"
```

Next, open **MenuLayer.m** and add the following code to `onEnter` (right below the initial `[super onEnter]`). This will authenticate the player every time the menu screen is presented.

```
[[GameKitHelper sharedGameKitHelper]
    authenticateLocalPlayer];
```

Build and run the application, and now when the menu view controller appears you will see one of the following screens:



The image on the left shows the login view controller, presented in case the user was not logged in with Game Center. The image on the right shows the welcome banner, displayed every time an authentication call is successful.

Note: To test the authentication, first logout of Game Center and then login through the MonkeyJump app. This will run Game Center in sandbox mode. Additionally, this probably will not work on the Simulator (at least, it didn't work at the time of writing). You would need to test this on an actual device.

A unified experience

Players who use Game Center tend to use the Game Center app to look at the achievements they have earned, compare their scores with friends – and now with iOS 6, they will be using it to view the challenges they have received as well.

But what if they want to view all this information without leaving your game? Prior to iOS 6, they were out of luck – a user would have to leave your app to go to Game Center!

Good news – in iOS 6, you can now give a unified experience to your players by using the new `GKGameCenterViewController` class. Not only does this present all the Game Center information like leaderboards and achievements, but also allows the player to rate your app and like it on Facebook. Not bad, eh?

Let's try this out. Open **GameKitHelper.h** and add the following method declaration:

```
// Game Center UI
-(void) showGameCenterViewController;
```

Now open **GameKitHelper.m** and define this method as follows:

```
#pragma mark Game Center UI method

-(void) showGameCenterViewController {
```

```

//1
GKGameCenterViewController *gameCenterViewController
    = [[GKGameCenterViewController alloc] init];

//2
gameCenterViewController.gameCenterDelegate = self;

//3
gameCenterViewController.viewState
    = GKGameCenterViewControllerStateDefault;

//4
[self presentViewController:gameCenterViewController];
}

```

Here is a step-wise explanation of the above method:

1. First the method creates an instance of `GKGameCenterViewController`.
2. It then sets itself as the delegate for `GKGameCenterViewController`.
3. The view state of the `GKGameCenterViewController` is set to the default. The view state is basically the initial view that is presented to the user. You can choose to show leaderboards, achievements, or challenges as the initial view, or let iOS decide the default screen to show (currently leaderboards).
4. Finally, the view controller is presented to the player.

You've already set `GameKitHelper` as implementing `GKGameControllerDelegate`. If you're wondering where, check the class extension at the top of **GameKitHelper.m**.

Now add the `gameCenterViewControllerDidFinish:` method as shown below to support the protocol:

```

#pragma mark GKGameControllerDelegate method

- (void)gameCenterViewControllerDidFinish:
    (GKGameCenterViewController *)gameCenterViewController {

    [self dismissModalViewControllerAnimated];
}

```

Notice that the delegate method calls a new method that you haven't implemented yet. So add that now:

```

-(void) dismissModalViewControllerAnimated {
    UIViewController* rootVC = [self getRootViewController];
    [rootVC dismissViewControllerAnimated:YES completion:nil];
}

```

```
}
```

You can see that this is basically the same as the previous helper methods you added. It simply dismisses the view controller on top, by way of the root view controller.

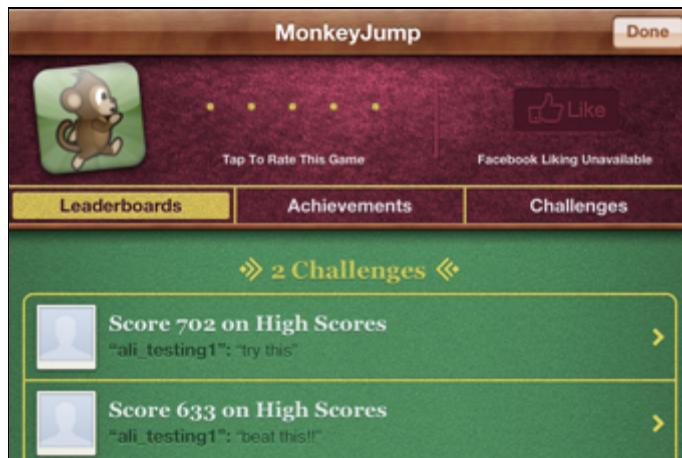
That's it! You've added support to present the `GKGameCenterViewController` to the player. Now let's use this newly-created method in your game.

Open **MenuLayer.m** and replace the log that prints, "Show the Game Center UI" in `menuButtonPressed:` with the following:

```
[ [GameKitHelper sharedGameKitHelper]
    showGameCenterViewController];
```

The code above is executed when the player taps the Game Center button on the menu screen.

Build and run the app. Now when the user presses the Game Center button, a screen like the one below will appear:



Play around with the view state of the `GKGameCenterViewController` (by setting it to leaderboards, achievements, challenges, etc.) just to see how the initial display varies as you change the state.

Submitting scores to Game Center

To send a score to Game Center, use the `GKScore` class. This class holds information about the player's score and the category to which it belongs.

The category refers to the leaderboard ID. For example, if you wish to submit a score to the High Scores leaderboard, then the category of the `GKScore` object would be the leaderboard ID that you set in iTunes Connect, which in your case is `HighScores`.

Open **GameKitHelper.h** and add the following method declaration:

```
// Scores
-(void) submitScore:(int64_t)score
    category:(NSString*)category;
```

Next, add the following method declaration to `GameKitHelperProtocol`:

```
-(void) onScoresSubmitted:(BOOL)success;
```

Now open **GameKitHelper.m** and add the following code:

```
-(void) submitScore:(int64_t)score
    category:(NSString*)category {
    //1: Check if Game Center
    //    features are enabled
    if (![_gameCenterFeaturesEnabled]) {
        CCLOG(@"Player not authenticated");
        return;
    }

    //2: Create a GKScore object
    GKScore* gkScore =
        [[GKScore alloc]
            initWithCategory:category];

    //3: Set the score value
    gkScore.value = score;

    //4: Send the score to Game Center
    [gkScore reportScoreWithCompletionHandler:
        ^(NSError* error) {

            [self setError:error];

            BOOL success = (error == nil);

            if ([_delegate
                respondsToSelector:
                @selector(onScoresSubmitted:)]) {

                [_delegate onScoresSubmitted:success];
            }
        }];
}
```

Here's a step-by-step explanation of the above method:

1. Check to see if Game Center features are enabled, and execution proceeds only if they are.
2. Create an instance of `GKScore`. The category used to create an object of `GKScore` is passed as an argument to the method.
3. Set the value of the `GKScore` object.
4. Send the `GKScore` object to Game Center using the `reportScoreWithCompletionHandler:` method. Once the score is sent, the platform calls the completion handler. The completion handler is a block that has one argument, in this case an `NSError` object that you can use to find out if the score was submitted successfully.

Now that you have defined a method to submit a score to Game Center, it's time to use it! But before you do that, open **GameConstants.h** and add the following define statement at the end (before the last `#endif`):

```
#define kHighScoreLeaderboardCategory @"HighScores"
```

Next, open **GameLayer.m** and find the method named `monkeyDead`. As the name implies, this method is called when the monkey dies. In other words, this is where the game would end.

Add the following code as the first line in the method:

```
[ [GameKitHelper sharedGameKitHelper]
    submitScore:(int64_t)_distance
    category:kHighScoreLeaderboardCategory];
```

Now build and run the app. Play the game until the monkey dies. Poor little dude!

When you finish playing, your score will be sent to Game Center. To verify that everything is working, open the Game Center app, tap on the Games tab and select the `MonkeyJump` game. The leaderboard should show your score. Here is a screenshot of Game Center showing the HighScores leaderboard:



Did you beat my score? No hacking the source code, now! ☺

Of course, since you currently have a Game Center view in the MonkeyJump app itself, you don't even have to exit the app to go to Game Center if you don't want to. You can check the leaderboards from within the app by accessing the Game Center option from the main menu.

Reporting achievements

To report achievements, you use the `GKAchievement` class. The `GKAchievement` class has two important properties:

- **identifier:** This property is similar to the identifier/category for a `GKScore`. It is used to identify the type of achievement. In this case, since you've defined three achievements in iTunes Connect, this value would be set to any of those three IDs.
- **percentComplete:** This property is of type double and it helps the player earn an achievement over time. For example, let's say that you have an achievement for the total distance run over multiple games. If the achievement is for 5000 meters and if the player completes a distance of 2500 meters in the current game, you would set this property to 50%.

Next you will write a method to load all the achievements earned by the local player and store them in a property called `achievements`. These achievements can even be those that have been partially earned.

Your reason for storing the achievements is to avoid reporting any that have already been achieved. For example, if a player X has earned the *Novice Runner* achievement already, then you should not report it again.

Open **GameKitHelper.m** and add the following method to it:

```
-(void) loadAchievements {

    //1
    if (![_gameCenterFeaturesEnabled]) {
        CCLOG(@"Player not authenticated");
        return;
    }

    //2
    [GKAchievement
        loadAchievementsWithCompletionHandler:
        ^NSArray* loadedAchievements, NSError* error) {

        [self SetLastError:error];

        if (_achievements == nil) {
            _achievements =
                [[NSMutableDictionary alloc] init];
        } else {
            [_achievements removeAllObjects];
        }

        for (GKAchievement* achievement
            in loadedAchievements) {
            achievement.showsCompletionBanner = YES;
            _achievements[achievement.identifier]
                = achievement;
        }
        if ([_delegate respondsToSelector:
            @selector(onAchievementsLoaded:)]) {
            [_delegate
                onAchievementsLoaded:_achievements];
        }
    }];
}
```

Here is a step-by-step explanation of the method:

1. First the method checks to see if Game Center features are enabled, and proceeds only if they are.

2. Next the method uses the `loadAchievementsWithCompletionHandler:` method of the `GKAchievement` class. This method loads all the achievements for the local player asynchronously, and when done, it calls the completion block. The completion block has two arguments: an array of achievements and an `NSError` object. The block stores the array in the `achievements` property of the `GameKitHelper` object and notifies the delegate.

The above method should be executed as soon as the local player is authenticated. That way, when you report an achievement, the `achievements` property already has the required entry.

To do this, modify `authenticateLocalPlayer` by adding the following after the `_gameCenterFeaturesEnabled = YES` line in the second `if` condition:

```
[self loadAchievements];
```

Great! Now that you've got all the achievements stored in the `achievements` array, you can write the method to report an achievement.

Open `GameKitHelper.h` and add the following method declaration:

```
-(void) reportAchievementWithID:  
    (NSString*)identifier  
    percentComplete:(float)percent;
```

Also add the following method declaration to `GameKitHelperProtocol`:

```
-(void) onAchievementReported:(GKAchievement*)achievement;
```

Now open `GameKitHelper.m` and add the following methods:

```
-(GKAchievement*) getAchievementByID:  
    (NSString*)identifier {  
  
    //1  
    GKAchievement* achievement =  
        _achievements[identifier];  
  
    //2  
    if (achievement == nil) {  
        // Create a new achievement object  
        achievement = [[GKAchievement alloc]  
                        initWithIdentifier:identifier];  
        achievement.showsCompletionBanner = YES;  
        _achievements[achievement.identifier]  
            = achievement;  
    }  
    return achievement;  
}
```

```
}

-(void) reportAchievementWithID:
    (NSString*)identifier
    percentComplete:(float)percent {
    //1
    if (![_gameCenterFeaturesEnabled]) {
        CCLOG(@"Player not authenticated");
        return;
    }

    //2
    GKAchievement* achievement =
        [self getAchievementByID:identifier];

    //3
    if (achievement != nil
        && achievement.percentComplete < percent) {

        achievement.percentComplete = percent;

        [achievement
            reportAchievementWithCompletionHandler:
            ^(NSError* error) {

                [self setError:error];

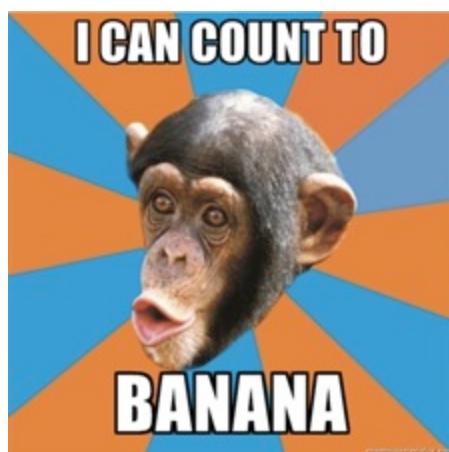
                if ([_delegate
                    respondsToSelector:
                    @selector(onAchievementReported:)])
                {
                    [_delegate
                        onAchievementReported:achievement];
                }
            }];
    }
}
```

The `getAchievementByID:` method is responsible for returning an achievement from either the achievements array or as a newly-created `GKAchievement` instance. This method is called from the `reportAchievementWithID:percentComplete:` method.

The `reportAchievementWithID:percentComplete:` method takes two arguments. The first is the achievement identifier and the second is a float value that represents the percentage of completion of an achievement. The internals of the method are pretty straightforward, but nonetheless here is a brief breakdown:

1. The method checks if Game Center features are enabled and continues only if they are.
2. The `getAchievementByID:` method is called with the passed-in achievement ID, which returns a `GKAchievement` matching that ID (or it creates a new achievement for that ID).
3. If the achievement returned from the previous step is not `nil` and the player has completed more than the current value of the `percentComplete` property, the achievement is reported to Game Center using the `reportAchievementWithCompletionHandler:` method. If the achievement is reported successfully, the delegate is informed using the `onAchievementReported:` method.

Now that you have added all the code needed to report an achievement, it's time to start using this new method. But first you need to write a method that will convert distance into percentage. The percent calculation will vary according to each achievement. For example, if the player ran 200 meters, the percentage for the *Novice Runner* achievement would be 100%, for *Athlete* it would be 20%, and for *2000meters* it would be 10%.



To create this method, first create a plist file called **Achievements.plist** in the plists group. Change the type of the root element to Array. Then add a dictionary to the root element and add two new items to it, each of type String. Name them **achievementId** and **distanceToRun**. Enter their values as *NoviceRunner* and 200.

Repeat the same process for the other two achievements. When you're done, the plist should look like the one shown below:

| Key | Type | Value |
|---------------|------------|--------------|
| Root | Array | (3 items) |
| Item 0 | Dictionary | (2 items) |
| achievementId | String | NoviceRunner |
| distanceToRun | String | 200 |
| Item 1 | Dictionary | (2 items) |
| achievementId | String | Athlete |
| distanceToRun | String | 1000 |
| Item 2 | Dictionary | (2 items) |
| achievementId | String | 2000meters |
| distanceToRun | String | 2000 |

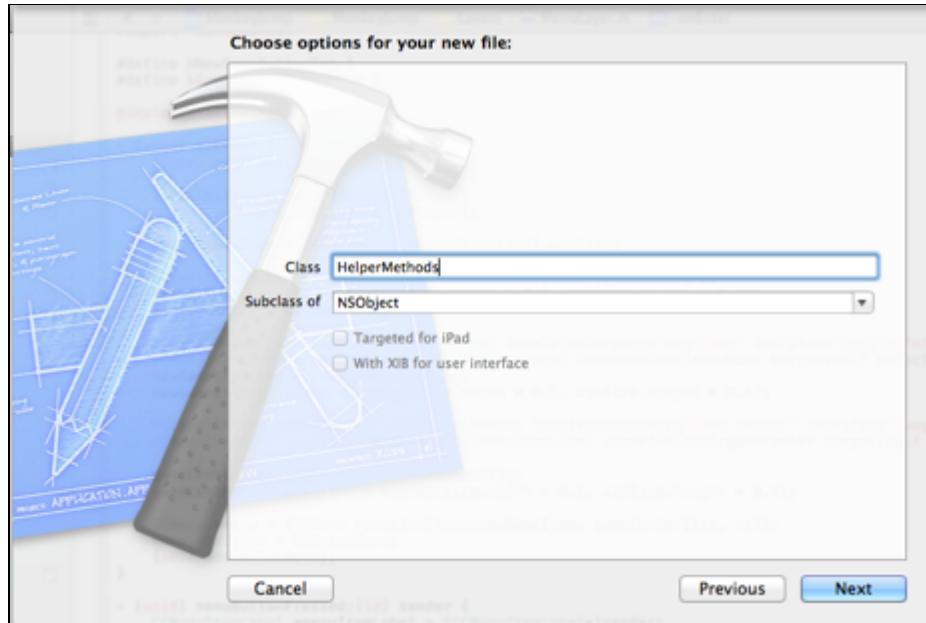
Can you guess why you added all the achievement-related information to the plist? If tomorrow you decide to add more achievements, all you have to do is make an entry in this file. No code change required!

Open **GameConstants.h** and add the following define statements at the end:

```
#define kAchievementsFileName @"Achievements.plist"
#define kAchievementsResourceName @"Achievements"
```

Now let's write a helper method that, for each achievement in the Achievements.plist file, will convert the distance completed by the player into a percentage.

Create a new class in the **GameKitFiles** group, name it **HelperMethods**, and make sure it extends **NSObject**.



Declare the following class method in **HelperMethods.h**:

```
+ (void) reportAchievementsForDistance:
    (int64_t) distance;
```

Next, open **HelperMethods.m** and include the **GameConstants.h** file as shown below:

```
#import "GameConstants.h"
```

Add the following method to the same file:

```
+ (void) reportAchievementsForDistance
    :(int64_t) distance {

    // 1
    NSString *rootPath =
        NSSearchPathForDirectoriesInDomains
        (NSDocumentDirectory,
         NSUserDomainMask,
         YES)[0];

    NSString *plistPath =
        [rootPath
         stringByAppendingPathComponent:
         kAchievementsFileName];

    if (![[NSFileManager defaultManager]
          fileExistsAtPath:plistPath]) {

        plistPath =
            [[NSBundle mainBundle]
             pathForResource:kAchievementsResourceName
             ofType:@"plist"];
    }

    NSArray *achievements =
        [NSArray arrayWithContentsOfFile:plistPath];

    if (achievements == nil) {
        CCLOG(@"Error reading plist: %@", kAchievementsFileName);
        return;
    }

    // 2
    for (NSDictionary *achievementDetail
         in achievements) {

        NSString *achievementId =
            achievementDetail[@"achievementId"];
```

```
NSString *distanceToRun =
    achievementDetail[@"distanceToRun"];

float percentComplete =
    (distance * 1.0f/[distanceToRun intValue])
    * 100;

if (percentComplete > 100)
    percentComplete = 100;

[[GameKitHelper sharedGameKitHelper]
    reportAchievementWithID:achievementId
    percentComplete:percentComplete];
}

}
```

Below is a brief explanation of the method:

1. The first part of the method reads the **Achievements.plist** file and populates an array with all the entries in the plist.
2. The next part calculates the percentage of each achievement that has been completed, and sends it to Game Center.

You are now going to add a call to `reportAchievementsForDistance:` when the game is over. Open **GameLayer.m** and add the following code to `monkeyDead` (below the previous code you added to report the score):

```
[HelperMethods
    reportAchievementsForDistance:(int64_t)_distance];
```

Of course, you also need to add an import to the top of **GameLayer.m**:

```
#import "HelperMethods.h"
```

Build and run the app, and try your hand once more at jumping the monkey. Whenever you complete an achievement, you should see an achievement banner displayed at the top of the screen. And you should now be able to track your achievement progress using the Game Center screen in your app (or the Game Center app if you prefer).

Sharing scores and achievements

The `GKGameCenterViewController` also allows the user to share scores and achievements to any social network supported by iOS, such as Twitter, Facebook, or Sina Weibo, as well as via messages and email.

Below is a screenshot showing the `GKGameCenterViewController`'s share feature:



The sharing feature of the `GKGameCenterViewController` is great, but what if you want to allow the player to immediately share their scores and achievements within your game without having to go to the `GKGameCenterViewController` screen?

In iOS 6, the solution is simple: you can pop up a `UIActivityViewController` to let the user share their scores/achievements through any of the available social channels – the same as you learned about in Chapter 11, “Beginning Social Framework.” Sharing has never been so easy!

Open `GameKitHelper.h` and add the following method declaration:

```
- (void) shareScore:  
    (int64_t)score  
    category:(NSString*)category;
```

Now add the method to `GameKitHelper.m`:

```
- (void) shareScore:(int64_t)score  
    category:(NSString*)category {  
    //1  
    GKScore* gkScore =  
        [[GKScore alloc]  
            initWithCategory:category];  
  
    gkScore.value = score;  
  
    //2  
    UIActivityViewController  
    *activityViewController =  
        [[UIActivityViewController alloc]  
            initWithActivityItems:@[gkScore]  
            applicationActivities:nil];
```

```
//3
activityViewController.completionHandler =
^(NSString *activityType, BOOL completed) {

    if (completed)
        [self dismissModalViewControllerAnimated];
};

//4
[self presentViewController:
    activityViewController];
}
```

Here is a step-by-step explanation of the method:

1. First a `GKScore` object is created with a category and value.
2. The method then sets up a `UIActivityViewController` and passes the `GKScore` object to it through the `init` method.
3. Then a completion handler is setup for the `UIActivityViewController`. This completion handler is called when the sharing action has been completed, so that the method can dismiss the `UIActivityViewController`.
4. Finally, the `UIActivityViewController` is presented to the player.

To use this method in your game, open **GameOverLayer.m** and replace the `CCLOG(@"Share button pressed");` line in `menuButtonPressed:` with the following:

```
[[GameKitHelper sharedGameKitHelper]
    shareScore:_score catergory:
    kHighScoreLeaderboardCategory];
```

Also add the following import to the file:

```
#import "GameConstants.h"
```

Compile and run the game. Play around with the monkey. When the game is over, tap on the Share Score button, and a `UIActivityViewController` should pop up. Hurray – you can now share your score with the entire world!



Note: The number of actions available on the activity view will vary depending on the capabilities of your device (for instance, whether it can send text messages), as well as the social networks you've configured on the device. So you might not see the same icons as in the image above.

Game Center challenges

Finally - the section you've been waiting for!

Game Center challenges is the biggest new Game Center feature introduced in iOS 6.0. Challenges can help make your game go viral, and they increase user retention tremendously.

The only problem is, integrating challenges is extremely difficult since the API is vast and complicated.



Just kidding! To integrate challenges into your game, all you have to do is... ABSOLUTELY NOTHING! ☺ If your game supports leaderboards and achievements (which it currently does), it will automatically support challenges without you having to do any extra work.



To test this, open the Game Center application (make sure you are in sandbox mode). Go to the Games tab and open the MonkeyJump game. (Alternatively, you can simply use the Game Center screen in your own app since that shows you all Game Center information for your app.)

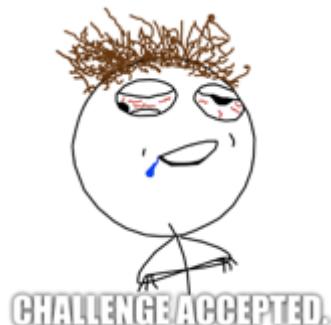
If you have unlocked an achievement, go to the achievements section and open the achievement. You will see a **Challenge Friends** button. Tap on it, enter the names of friends you want to challenge and press Send (the same can be done with leaderboards). When the challenge is sent, your friends will receive a push notification.

Note: To test challenges, you will need two devices running iOS 6.0, each logged into Game Center with a different account, and the accounts need to have added each other as friends.

Challenges are not mere push notifications. Let me briefly explain how challenges work with an example.

Suppose I send a score challenge to Ray of 500 meters. Ray will receive a push notification on his device informing him of the challenge. Let's suppose he gets a score of 1000 meters when he plays the game. In other words, Ray wipes the floor with the challenge. And he definitely wants me to know about that.

Since the game reports all scores to Game Center, it knows automatically that Ray killed the challenge, so it will send a challenge completion push notification to both the devices. Ray can then challenge me with his 1000 meter score. Little does he know that I can do 1000 meters in my sleep.



This process can go on indefinitely, with each party repeatedly topping the other's score. It is because of this addictive, self-perpetuating use pattern that every game developer should integrate challenges into his/her games!

Up until now, you've tested challenges using the Game Center application or the built-in Game Center view in your app, but what if you want to allow the user to challenge his/her friends from within your game?

That's exactly what you're going to do next. ☺ You will add this functionality to your game and allow the player to select which friends s/he wants to challenge using a friend picker.

Open **GameKitHelper.h** and add a new property to it.

```
@property (nonatomic, readwrite)  
BOOL includeLocalPlayerScore;
```

Next add the following method declarations to **GameKitHelperProtocol**:

```
-(void) onScoresOfFriendsToChallengeListReceived:  
      (NSArray*) scores;  
-(void) onPlayerInfoReceived:  
      (NSArray*) players;
```

Also add these method declarations to **GameKitHelper**:

```
-(void) findScoresOfFriendsToChallenge;  
  
-(void) getPlayerInfo:(NSArray*)playerList;  
  
-(void) sendScoreChallengeToPlayers:  
      (NSArray*)players  
      withScore:(int64_t)score  
      message:(NSString*)message;
```

Next you're going to define each one of the above methods in **GameKitHelper.m**. Let's start with **findScoresOfFriendsToChallenge**. Add the following lines of code:

```
-(void) findScoresOfFriendsToChallenge {  
    //1  
    GKLeaderboard *leaderboard =  
        [[GKLeaderboard alloc] init];  
  
    //2  
    leaderboard.category =  
        kHighScoreLeaderboardCategory;  
  
    //3
```

```
leaderboard.playerScope =
    GKLeaderboardPlayerScopeFriendsOnly;

//4
leaderboard.range = NSMakeRange(1, 100);

//5
[leaderboard
    loadScoresWithCompletionHandler:
    ^(NSArray *scores, NSError *error) {

        [self setError:error];

        BOOL success = (error == nil);

        if (success) {
            if (!_includeLocalPlayerScore) {
                NSMutableArray *friendsScores =
                    [NSMutableArray array];

                for (GKScore *score in scores) {
                    if (![score.playerID
                        isEqualToString:
                        [GKLocalPlayer localPlayer]
                        .playerID]) {
                        [friendsScores addObject:score];
                    }
                }
                scores = friendsScores;
            }
            if ([_delegate
                respondsToSelector:
                @selector
                (onScoresOfFriendsToChallengeListReceived:)])
            {
                [_delegate
                onScoresOfFriendsToChallengeListReceived:scores];
            }
        }
    }];
}
```

This method is responsible for fetching the scores of all the player's friends. To do this, the method queries the HighScores leaderboard for scores of the local player's friends.

Every time you request scores, Game Center adds the score of the local player to the results by default. For example, in the above method when you request the scores of all the player's friends, Game Center returns an array containing not only the scores of the player's friends, but the player's score as well. So, you use the `includeLocalPlayerScore` property to decide whether or not to remove the local player's score from the scores array. By default, this is NO (don't include the player's score).

Now add the following method:

```
- (void) getPlayerInfo:(NSArray*)playerList {
    //1
    if (_gameCenterFeaturesEnabled == NO)
        return;

    //2
    if ([playerList count] > 0) {
        [GKPlayer
            loadPlayersForIdentifiers:
            playerList
            withCompletionHandler:
            ^(NSArray* players, NSError* error) {

                [self setError:error];

                if ([_delegate
                    respondsToSelector:
                    @selector(onPlayerInfoReceived:)])
                {

                    [_delegate onPlayerInfoReceived:players];
                }
            }];
    }
}
```

This method gets player information for a list of players by passing in an array of player IDs.

One final method – add the following code:

```
- (void) sendScoreChallengeToPlayers:
    (NSArray*)players
    withScore:(int64_t)score
    message:(NSString*)message {

    //1
    GKScore *gkScore =
```

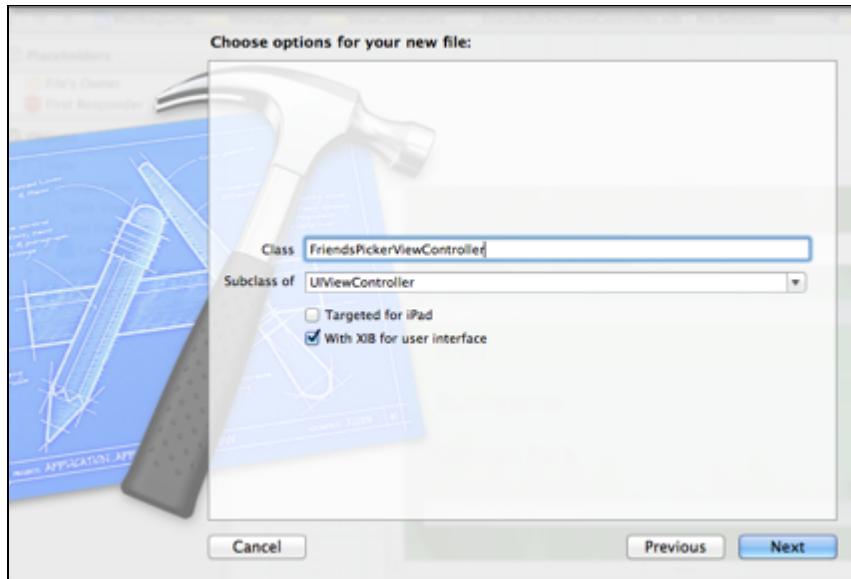
```
[ [GKScore alloc]
    initWithCategory:
    kHighScoreLeaderboardCategory];
gkScore.value = score;

//2
[gkScore issueChallengeToPlayers:
    players message:message];
}
```

This method sends out a score challenge to a list of players, accompanied by a message from the player.

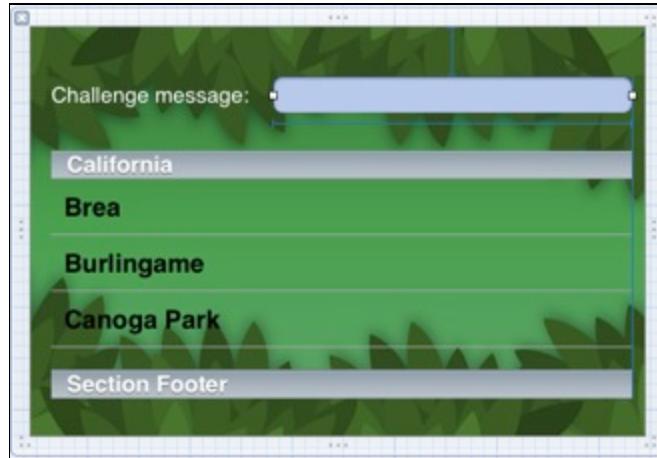
Great! Next, you need a friend picker. The friend picker will allow the player to enter a custom message and select the friends that s/he wants to challenge. By default, it will select those friends that have a score lower than the local player's score, since these are the ones that the player should definitely challenge. After all, the player wants to win! ☺

Create a new group in Xcode and name it **ViewControllers**. Then create a new file in that group that extends `UIViewController` and name it **FriendsPickerController**. Make sure you check the "With XIB for user interface" checkbox, as shown below.



Open the **FriendsPickerController.xib** file, set the view's orientation to landscape, drag a `UITableView`, a `UITextField` and a `UILabel` onto the canvas, and set the `text` property of the label as "Challenge message".

Also, to make sure that this view controller has the same look and feel as the rest of the game, add **bg_menu.png** as the background image. The final view controller should look like this:



Open **FriendsPickerController.h** and add the following statements above the `@interface` line:

```
typedef void (^FriendsPickerCancelButtonPressed)();  
typedef void (^FriendsPickerChallengeButtonPressed)();
```

These two new data types, `FriendsPickerCancelButtonPressed` and `FriendsPickerChallengeButtonPressed`, describe the blocks you'll be using. A block is like a C function; it has a return type (in this case `void`) and zero or more parameters. The `typedef` makes it a bit easier to refer to this block in the code.

Add the following properties to the `@interface` section:

```
//1  
@property (nonatomic, copy)  
    FriendsPickerCancelButtonPressed  
    cancelButtonPressedBlock;  
  
//2  
@property (nonatomic, copy)  
    FriendsPickerChallengeButtonPressed  
    challengeButtonPressedBlock;
```

These properties represent blocks of code that will be executed when either the Cancel or the Challenge buttons are pressed.

Next add the Cancel and Challenge buttons to the view controller. Open **FriendsPickerController.m** and replace `viewDidLoad` with the following code:

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

```

UIBarButtonItem *cancelButton =
[ [UIBarButtonItem alloc]
    initWithTitle:@"Cancel"
    style:UIBarButtonItemStylePlain
    target:self
    action:@selector(cancelButtonPressed:)];

UIBarButtonItem *challengeButton =
[ [UIBarButtonItem alloc]
    initWithTitle:@"Challenge"
    style:UIBarButtonItemStylePlain
    target:self
    action:@selector(challengeButtonPressed:)];

self.navigationItem.leftBarButtonItem =
cancelButton;
self.navigationItem.rightBarButtonItem =
challengeButton;
}

```

The method adds two `UIBarButtonItem`s to the view controller, representing the Cancel and Challenge buttons. Now add the methods that will be called when these buttons are tapped.

```

- (void)cancelButtonPressed:(id) sender {
    if (self.cancelButtonPressedBlock != nil) {
        self.cancelButtonPressedBlock();
    }
}

- (void)challengeButtonPressed:(id) sender {
    if (self.challengeButtonPressedBlock) {
        self.challengeButtonPressedBlock();
    }
}

```

The above methods are easy to understand – all you do is execute the code in the challenge and cancel blocks.

Before you can integrate this view controller into the game and test to see if everything works, you first need to write an initialization method that takes the score of the local player. But before you do this, you must define a variable to hold the score.

Add the following variable to the class extension at the top of **FriendsPickerController.m** – and remember to enclose the variable in curly brackets so that the final class extension looks like this:

```
@interface FriendsPickerController () {
    int64_t _score;
}

@end
```

Now add the following initialization method:

```
- (id)initWithScore:(int64_t) score {
    self = [super
        initWithNibName:
        @"FriendsPickerController"
        bundle:nil];

    if (self) {
        _score = score;
    }
    return self;
}
```

Add the method declaration for the above to **FriendsPickerController.h**, as shown below:

```
- (id)initWithScore:(int64_t) score;
```

Now you are ready to test this view controller to see if everything works as expected. Open **GameKitHelper.h** and define a method as follows:

```
-(void)
showFriendsPickerControllerForScore:
(int64_t)score;
```

Then open **GameKitHelper.m** and add the following import statement:

```
#import "FriendsPickerController.h"
```

Next, add the method as follows:

```
-(void)
showFriendsPickerControllerForScore:
(int64_t)score {

    FriendsPickerController
    *friendsPickerController =
        [[FriendsPickerController alloc]
        initWithScore:score];
```

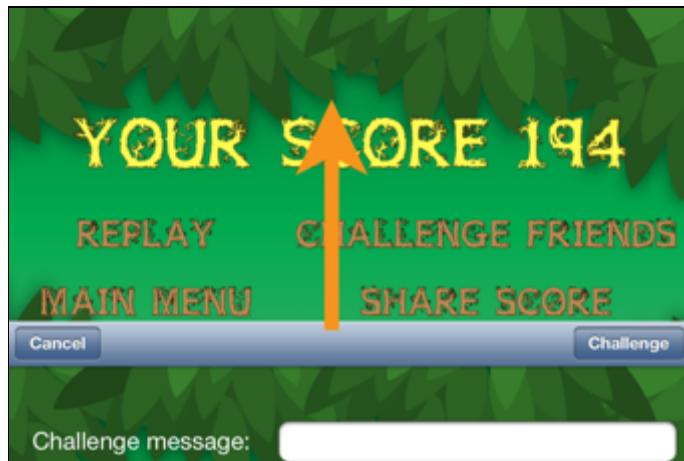
```
friendsPickerController.  
    cancelButtonTitle = ^() {  
        [self dismissModalViewControllerAnimated];  
    };  
  
friendsPickerController.  
    challengeButtonPressedBlock = ^() {  
        [self dismissModalViewControllerAnimated];  
    };  
  
UINavigationController *navigationController =  
    [[UINavigationController alloc]  
        initWithRootViewController:  
            friendsPickerController];  
  
[self presentViewController:navigationController];  
}
```

This method presents the `FriendPickerController` modally. It also defines the blocks that will be executed when the Challenge and Cancel buttons are pressed. In this case, all that happens is that the view controller is dismissed.

Now open `GameOverLayer.m` and replace the `CCLOG(@"Challenge button pressed");` line in `menuButtonPressed:` with the following:

```
[ [GameKitHelper sharedGameKitHelper]  
    showFriendsPickerControllerForScore:_score];
```

Here is the moment of truth! Build and run the game, play a round of `MonkeyJump`, and when you press the Challenge Friends button on the game over screen, you will be presented with the `FriendsPickerController`. If you tap on either the Challenge or the Cancel button the view controller will be dismissed.



Great! Your game now has the ability to show the friends picker view controller. But the view controller does not show any friends, which kind of defeats the purpose.

No need to feel lonely – let's add this functionality!

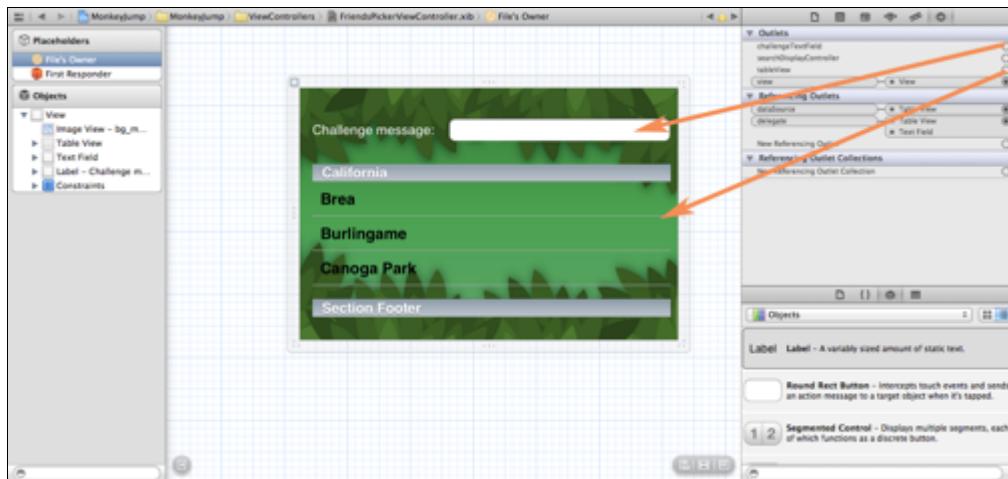
Open **FriendsPickerController.m** and replace the class extension with the following:

```
@interface FriendsPickerController : UIViewController
<UITableViewDataSource,
UITableViewDelegate,
UITextFieldDelegate,
GameKitHelperProtocol>

NSMutableDictionary *_dataSource;
int64_t _score;

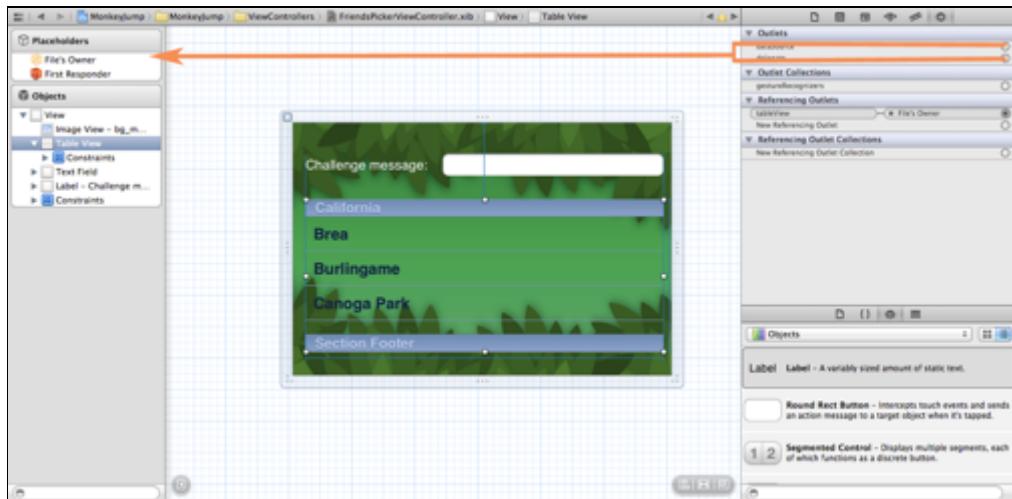
@property (nonatomic, weak)
IBOutlet UITableView *tableView;
@property (nonatomic, weak)
IBOutlet UITextField *challengeTextField;
@end
```

Notice that the interface now implements a bunch of protocols. Along with that, it also has two `IBOutlets`, one for the `UITableView` and the other for the `UITextField`. Connect these properties to their respective views using Interface Builder, as shown below:



Next set the delegate and the data source of the `UITableView`, and the delegate of the `UITextField`, as the File's Owner in Interface Builder.

To do this, select the `UITableView` and in the Connections inspector, drag the data source and delegate outlets to the File's Owner on the left, as shown in the image below:



Repeat the process for the UITextField.

Switch to **FriendsPickerController.m** and add the following code within the if condition in `initWithScore:`, below the `_score = score;` line:

```
_dataSource = [NSMutableDictionary dictionary];

GameKitHelper *gameKitHelper = [GameKitHelper
sharedGameKitHelper];

gameKitHelper.delegate = self;
[gameKitHelper findScoresOfFriendsToChallenge];
```

This method initializes the data source, sets itself as the delegate for GameKitHelper, and calls `findScoresOfFriendsToChallenge`. If you recall, this method is used to find the scores of all the friends of the local player. Implement the `onScoresOfFriendsToChallengeListReceived:` delegate method to handle what happens after the player's friends' scores are fetched:

```
-(void)
onScoresOfFriendsToChallengeListReceived:
(NSArray*) scores {
//1
NSMutableArray *playerIds =
[NSMutableArray array];

//2
[scores enumerateObjectsUsingBlock:
^(id obj, NSUInteger idx, BOOL *stop) {

    GKScore *score = (GKScore*) obj;
```

```

//3
if(_dataSource[score.playerID]
    == nil) {
    _dataSource[score.playerID] =
        [NSMutableDictionary dictionaryWithDictionary];
    [playerIds addObject:score.playerID];
}

//4
if (score.value < _score) {
    [_dataSource[score.playerID]
        setObject:[NSNumber numberWithBool:YES]
        forKey:kIsChallengedKey];
}

//5
[_dataSource[score.playerID]
    setObject:score forKey:kScoreKey];
};

//6
[[GameKitHelper sharedGameKitHelper]
    getPlayerInfo:playerIds];
[self.tableView reloadData];
}

```

The code is quite self-explanatory, but here's an explanation anyway:

1. An array named `playerIds` is created to hold the IDs of the local player's friends.
2. Then the method starts iterating over the returned scores.
3. For each score, an entry in the data source is created and the player ID is stored in the `playerIds` array.
4. If the score is less than the local player's score, the entry is marked in the data source.
5. The score is stored in the data source dictionary.
6. The `GameKitHelper`'s `getPlayerInfo:` method is invoked with the `playerIds` array. This will return the details of each friend, such as the player's name and profile picture.

The above method requires a few `#defines` in order to work. Add those (and a few others you'll need later on in the code) to the top of the file below the `#import` line:

```

#define kPlayerKey @"player"
#define kScoreKey @"score"
#define kIsChallengedKey @"isChallenged"

```

```
#define kCheckMarkTag 4
```

Next you need to implement the `onPlayerInfoReceived:` delegate method. This method is called when information for all the local player's friends has been received.

```
- (void) onPlayerInfoReceived:(NSArray*)players {
    //1

    [players
        enumerateObjectsUsingBlock:
        ^(id obj, NSUInteger idx, BOOL *stop) {

            GKPlayer *player = (GKPlayer*)obj;

            //2
            if (_dataSource[player.playerID]
                == nil) {
                _dataSource[player.playerID] =
                    [NSMutableDictionary dictionary];
            }
            [_dataSource[player.playerID]
                setObject:player forKey:kPlayerKey];

            //3
            [self.tableView reloadData];
        }];
}
```

This method is also quite straightforward; since you have the details of each player, you just update the `_dataSource` dictionary with each player's information.

The `_datasource` dictionary is used to populate the table view. Implement the table view's data source methods as shown below:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return _dataSource.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell identifier";
    static int ScoreLabelTag = 1;
```

```
static int PlayerImageTag = 2;
static int PlayerNameTag = 3;

UITableViewCell *tableViewCell =
    [tableView
        dequeueReusableCellWithIdentifier:
        CellIdentifier];

if (!tableViewCell) {

    tableViewCell =
        [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    tableViewCell.selectionStyle =
        UITableViewCellStyleGrayBackground;
    tableViewCell.textLabel.textColor =
        [UIColor whiteColor];

    UILabel *playerName =
        [[UILabel alloc] initWithFrame:
        CGRectMake(50, 0, 150, 44)];
    playerName.tag = PlayerNameTag;
    playerName.font = [UIFont systemFontOfSize:18];
    playerName.backgroundColor =
        [UIColor clearColor];
    playerName.textAlignment =
        UIControlContentVerticalAlignmentCenter;
    [tableViewCell addSubview:playerName];

    UIImageView *playerImage =
        [[UIImageView alloc]
            initWithFrame:CGRectMake(0, 0, 44, 44)];
    playerImage.tag = PlayerImageTag;
    [tableViewCell addSubview:playerImage];

    UILabel *scoreLabel =
        [[UILabel alloc]
            initWithFrame:
            CGRectMake(395, 0, 30,
                      tableViewCell.frame.size.height)];
    scoreLabel.tag = ScoreLabelTag;
    scoreLabel.backgroundColor =
        [UIColor clearColor];
```

```
scoreLabel.textColor =
    [UIColor whiteColor];
[tableViewCell.contentView
    addSubview:scoreLabel];

UIImageView *checkmark =
    [[UIImageView alloc]
        initWithImage:[UIImage
            imageNamed:@"checkmark.png"]];
checkmark.tag = kCheckMarkTag;
checkmark.hidden = YES;
CGRect frame = checkmark.frame;
frame.origin =
    CGPointMake(tableView.frame.size.width - 16, 13);
checkmark.frame = frame;
[tableViewCell.contentView
    addSubview:checkmark];
}

NSDictionary *dict =
[_dataSource allValues][indexPath.row];
GKScore *score = dict[kScoreKey];
GKPlayer *player = dict[kPlayerKey];

NSNumber *number = dict[kIsChallengedKey];

UIImageView *checkmark =
    (UIImageView*)[tableViewCell
        viewWithTag:kCheckMarkTag];

if ([number boolValue] == YES) {
    checkmark.hidden = NO;
} else {
    checkmark.hidden = YES;
}

[player
    loadPhotoForSize:GKPhotoSizeSmall
    withCompletionHandler:
    ^(UIImage *photo, NSError *error) {
        if (!error) {
            UIImageView *playerImage =
                (UIImageView*)[tableView
                    viewWithTag:PlayerImageTag];
            playerImage.image = photo;
        } else {
    }
}
}
```

```
        NSLog(@"Error loading image");
    }
}];

UILabel *playerName =
    (UILabel*)[tableViewCell
        viewWithTag:PlayerNameTag];
playerName.text = player.displayName;

UILabel *scoreLabel =
    (UILabel*)[tableViewCell
        viewWithTag:ScoreLabelTag];
scoreLabel.text = score.formattedValue;
return tableViewCell;
}
```

That's a lot of code. ☺ If you have used a UITableView before, the code should not be new to you. The `tableView:cellForRowAtIndex:` method creates a new UITableViewCell. Each cell of the table view will contain a profile picture, the player's name and the player's score.

Now add `tableView:didSelectRowAtIndexPath:` to handle the user selecting a row in the table view:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:
(NSIndexPath *)indexPath {

    BOOL isChallenged = NO;

    //1
    UITableViewCell *tableViewCell =
        [tableView cellForRowAtIndexPath:
            indexPath];

    //2
    UIImageView *checkmark =
        (UIImageView*)[tableViewCell
            viewWithTag:kCheckMarkTag];

    //3
    if (checkmark.isHidden == NO) {
        checkmark.hidden = YES;
    } else {
        checkmark.hidden = NO;
        isChallenged = YES;
    }
}
```

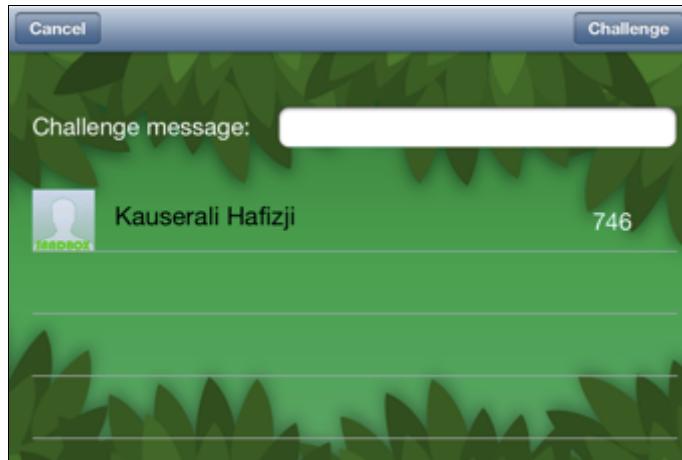
```
    }
    NSArray *array =
        [_dataSource allValues];

    NSMutableDictionary *dict =
        array[indexPath.row];

    //4
    [dict setObject:[NSNumber
                    numberWithInt:isChallenged]
                    forKey:kIsChallengedKey];
    [tableView deselectRowAtIndexPath:indexPath
                                animated:YES];
}
```

All the method does is set an entry in the `_datasource` to YES or NO.

Build and run the app. Now when the `FriendsPickerController` is launched the `UITableView` is populated with the local player's friends. The details for each friend, such as their name and profile picture, are also displayed in each cell. Here's what it looks like:



The final thing left to do is to actually send the challenge. Replace `challengeButtonPressed:` in `FriendsPickerController.m` with the following:

```
- (void)challengeButtonPressed:
    (id) sender {

    //1
    if(self.challengeTextField.text.
        length > 0) {

        //2
    }
}
```

```
NSMutableArray *playerIds =
    [NSMutableArray array];
NSArray *allValues =
    [_dataSource allValues];

for (NSDictionary *dict in allValues) {
    if ([dict[kIsChallengedKey]
        boolValue] == YES) {

        GKPlayer *player =
            dict[kPlayerKey];
        [playerIds addObject:
            player.playerID];
    }
}
if (playerIds.count > 0) {

    //3
    [[GameKitHelper sharedGameKitHelper]
        sendScoreChallengeToPlayers:playerIds
        withScore:_score message:
            self.challengeTextField.text];
}

if (self.challengeButtonPressedBlock) {
    self.challengeButtonPressedBlock();
}
} else {
    self.challengeTextField.layer.
        borderWidth = 2;
    self.challengeTextField.layer.
        borderColor =
            [UIColor redColor].CGColor;
}
}
```

Here is a step-by-step explanation of the above method:

1. The method first checks to see if the user has entered a message. If not, the border of the `challengeTextField` is turned red.
2. If the user has entered text, the method finds the player IDs of all the selected players and stores them in an array called `playerIds`.
3. If the user has selected players to challenge, then `sendScoreChallengeToPlayers:withScore:` is called from `GameKitHelper` with those player IDs. This will send the challenge to all the selected players.

Note: `sendScoreChallengeToPlayers:withScore:` will only send a score challenge. You can also enable the player to send achievement challenges if you wish. You'd just make a similar method to `sendScoreChallengeToPlayers:withScore` in GameKitHelper that uses `GKAchievement's issueChallengeToPlayers:message` method. I leave it up to you as a challenge, should you wish to accept it. ☺

Build and run the game. Now when you press the Challenge Friends button on the `FriendsPickerController`, it will send out a score challenge. If you have two devices you can easily test this to see if it works.

Challenges unlocked! You can now send challenges through code.



Where to go from here?

If you have been following this chapter from the start, then you have spent a lot of time messing with Game Center APIs. I hope you now have a clear understanding of how to use Game Center and work with the new APIs. I also hope you are as excited about the challenge APIs as I am. I would love to see these incorporated into your awesome games!

But wait, there's more! In the next chapter, you will learn to do some really cool things using the challenges API. To get a sneak peak, take a look at the beginning of the 2012 WWDC video titled "Game Technologies Kickoff", where it shows a cool demo of a player racing against a "ghost player" in Jetpack Joyride. You'll be implementing something like that into MonkeyJump in the next chapter! ☺

Chapter 14: Intermediate Challenges with GameKit

By Kauserali Hafizji

Creating a good game takes a lot of time and effort, not only from a design standpoint, but also from the development side. There is just a lot of work that needs to go into it!

But unfortunately, and in spite of all the work we put in, most of the good games in the App Store don't make it big.

You don't... love... my game?



The ones that do often have something in common: they use Game Center in a unique way. Take the famous *Temple Run* as an example. When the player runs by labels representing scores of his/her friends, they have a visceral experience that increases user retention tremendously.

In the previous chapter, you learned how to add Game Center features such as leaderboards, achievements and challenges to a simple Cocos2d game. These are all great ways to increase the loyalty of your app's users, but you can do even better and add a bit of extra effort to make your app stand above the rest!

In this chapter, you'll implement a practical example of doing this, by using Game Center to implement a "ghost challenge" feature. When challenging a friend, instead of simply sending your friend a score to beat, the friend will see a ghost representing how you played against the level that they can play against! This makes the challenge much more personal and satisfying.

To implement this, you will track a player's moves as he/she plays the game, send it to a private server, and when the player sends a challenge to a friend you will include a reference to the data on the server via the `context` property of `GKScore` and `GKAchievement`. Then when the friend accepts the challenge, you will look up the data on the private server and replay the original's player's moves as a "ghost."

Short of offering real money for game achievements, I can hardly think of a more powerful motivating factor to keep fans playing your game! ☺

This chapter continues with the MonkeyJump project from where you left off last time, so open the project or grab a copy from this chapter's resources before continuing.

Getting started

To go through this chapter, you will need access to a webserver. If you don't have your own webserver, you can set one up on your Mac by following the instructions given at <http://www.apachefriends.org/en/xampp-macosx.html>.

Note: Setting up a webserver is beyond the scope of this chapter. Please follow the instructions given on the page linked above.

Along with a webserver, you will also need to know PHP and some SQL syntax. So long as you are equipped with those three things, the rest of this section will take you through setting up your server and database for use with the MonkeyJump game.

Ready? Let's get started with your server set up!

Setting up your server

Open the **ServerScripts.zip** file found in this chapter's resources. Here you will find two files:

- **ChallengesAPI.php**: This file contains the PHP code that implements the web service you'll use in the sections to come. It supports a GET and a POST call.
- **DatabaseScripts.sql**: This file contains the SQL statements to create all the necessary databases and tables.

Step 1: Creating the database

You are going to create a database on your server to store gameplay information from the challenger, which you can then retrieve to generate an entire level, as well as reproduce each and every move of the player. You need to store the gameplay information in a database because you can only store a very limited

amount of information in a `GKScore` object – definitely not enough for gameplay tracking like this.

For this chapter, your database needs just one table, **ChallengesInfo**, to hold the gameplay info. This table has the following schema:

- *ChallengeId*: An identifier representing a challenge being sent to another player. It is used as a key to reference gameplay information.
- *JSON*: A text field used to store JSON data that represents gameplay information, such as when the player jumps or dies.

Note: Curious why the table does not have a fixed structure defining different gameplay elements? Well, I wanted to create a schema such that you can use the same table for any game.

The only drawback I see with this approach is that you cannot search for a particular item. For example, say the game sends an array of jump times to the server. You cannot run a SQL query asking for all entries having more than three jumps.

To create the database, you need to have MySQL installed on your webserver. If you are using a XAMPP server, MySQL comes along with it.

The **DatabaseScripts.sql** file contains SQL statements that will create a database with the structures you need. Here are the contents of the file:

```
# Remove the challenges database
drop database if exists challenges;

# Create a fresh challenges database
create database challenges;

# Use the new challenges database
use challenges;

# Remove the challengesInfo Table
drop table if exists ChallengesInfo;

# Create a fresh table
CREATE TABLE `ChallengesInfo` (
  `ChallengeId` varchar(255) NOT NULL,
  `JSON` text,
  PRIMARY KEY (`ChallengeId`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

If you are familiar with SQL syntax, then these statements should be fairly easy to understand. In essence, all that these statements do is create a new database called **challenges**, and then create the **ChallengesInfo** table (introduced above) within that database.

Note: The next step is to run the above SQL commands using a MySQL client application – this will create the database and table. The process for doing this is beyond the scope of this chapter, but you should be able to figure out what you need to do by referring to this tutorial:

<http://www.raywenderlich.com/2941/how-to-write-a-simple-phpmysql-web-service-for-an-ios-app>

Step 2: Creating and publishing the API

OK, now that you're done creating the database, you need a way to access it via the network. This means writing a web service, which is an API that is accessible via HTTP.

Though you can choose to write your web service in any language you fancy, to make things easier I have already created a PHP script that you can find in the file called **ChallengesAPI.php**. The contents of the file are shown below:

```
<?PHP
$request_method =
    strtolower($_SERVER['REQUEST_METHOD']);

# This method is used to get the
# game details for a specific Challenge
# Storage used is MySql

function GetData($challengeId){
    # 1: Create a connection to the database

    $link = mysql_connect('localhost','root','')
        or die('Cannot connect to the DB');

    # 2: Select the database

    mysql_select_db('challenges',$link)
        or die('Cannot select the DB');

    # 3: Run the query

    $query = "SELECT * from
```

```
    ChallengesInfo WHERE
    ChallengeId = '".$challengeId ."'';

$result = mysql_query($query,$link)
    or die('Errant query:  '.$query);

$data = '';
if(mysql_num_rows($result)) {
    while($info = mysql_fetch_array( $result )) {
        $data = $info[ 'JSON' ];
    }
}
@mysql_close($link);

# 4: Return the results
return $data;
}

# Check if the challenge id is set

if(isset($_GET['challengeId'])) {

    # Retrieve the challengeId from the url
    $challengeId = $_GET['challengeId'];

    // In case of "GET" call
    if($request_method == 'get'){
        $data = GetData($challengeId);

        if($data == ''){
            header("HTTP/1.0 404 Not Found");
        } else{
            header('Content-type: text/json');
            echo $data;
        }
    } else {

        $data = file_get_contents('php://input');
        if($data != ''){
            $existingData = GetData($challengeId);
            $link =
                mysql_connect('localhost','root','')
                or die('Cannot connect to the DB');

            mysql_select_db('challenges',$link)
        }
    }
}
```

```

        or die('Cannot select the DB');

        if($existingData == ''){
            $query = "INSERT INTO
                ChallengesInfo values (''
                .$challengeId.'', ''.$data.'')";

            $message = "Data Inserted";
        } else {

            $query = "UPDATE ChallengesInfo set JSON = ''
                .$data.'' where ChallengeId = ''
                .$challengeId .''";
            $message = "Data updated";
        }

        mysql_query($query,$link)
            or die('Errant query:  '.$query);
        header('Content-type: text/json');
        echo '{"__message" : "'.$message.'"}';
    }
}

} else {
    header('STATUS 400');
    header('Content-type: text/json');
    echo '{"__message" : "Please provide a challenge id"}';
}
?>

```

Here is a step-by-step explanation of what's happening:

- The script first checks the type of request being sent. Since we are adhering to the REST way of doing things, the request could be of type GET, POST, or PUT. In this script, you will be using GETs to retrieve gameplay data, and POSTs to store gameplay data.
- It then checks to see if a `challengeId` has been sent in the request, which represents a unique id for the challenge. This is done using `isset`, a handy PHP function used to check if a variable has been set.
- If the `challengeId` is not set, a response is sent with an HTTP Status Code of 400 (standing for BAD request).
- In case of a GET request, the script pulls a record from the database with the same `challengeId` as the one sent.
- In case of a POST request, the script saves all the data received in the body of the request to the database. The `challengeId` sent via the request is used to

determine whether it there is an existing record that should be updated, or if a new record should be created.

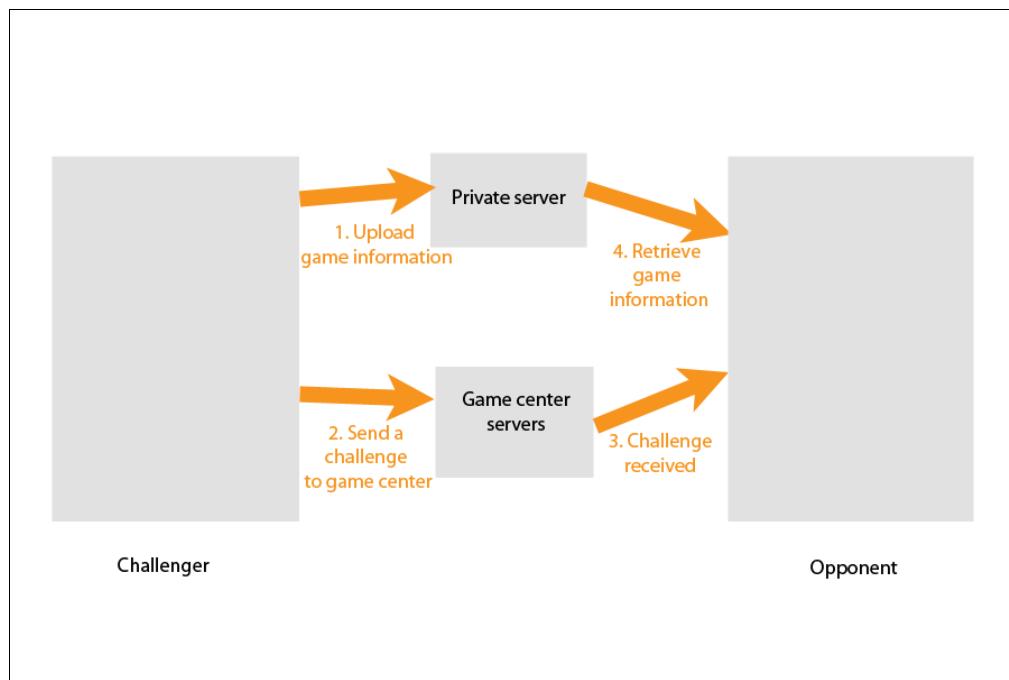
Now that you have a basic idea of what the web service does, create a new directory on your webserver and copy **ChallengesAPI.php** into it. To test if it is working, you can use any REST client (I prefer the Advanced REST Client available from the Chrome webstore).

Of course, you can do a simple test of the web service by typing the address of the ChallengesAPI.php file into your browser. You will receive a JSON response that will either contain an error message (for instance, if you didn't include the challenge ID) or some data (if you included a valid challenge ID).

How things will tie together

Up until now, your game has used a very simple model to send challenges to the opponent. All it does is make use of the functionality available via the Game Center APIs. You are now going to add your own server into the mix.

The flow diagram below illustrates the interaction you want to achieve between the challenger and opponent, via your private server and the Game Center servers:



Here is a step by step explanation:

1. Before sending a challenge to Game Center, first upload the challenger's gameplay information to your server.
2. After uploading the gameplay information, send a challenge request.

3. The challenge is received by the challenged player (the opponent).
4. The app on the challenged player's device retrieves all the gameplay information stored on the server.

At this point, the challenged player starts playing against his challenger's recorded game in real time - and starts having amazing fun! But how exactly will this work? How will the challenged player see her challenger's awesome moves replicated onscreen?

Enter the Ghost Monkey!

Adding a ghost monkey

This chapter's resources should include a Resources.zip file. Extract it and you will find two sprite sheets and their plist files. Copy these into your project's **MonkeyJump** folder, replacing the existing files.

These new sprite sheets have an extra monkey character along with all the animation frames – the only difference is that the new monkey has a very low contrast. From now on, I will refer to this new character as the "Ghost Monkey."

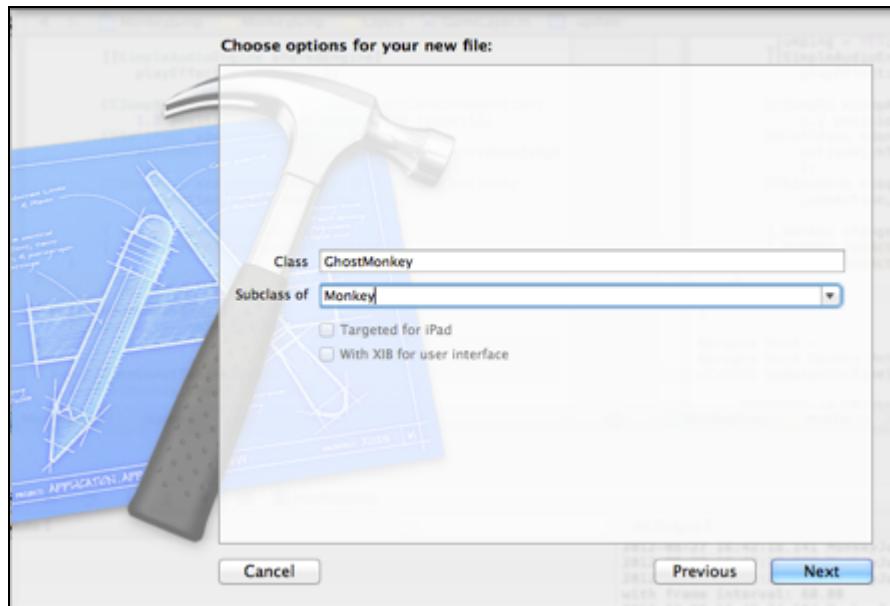


The purpose of the Ghost Monkey is to literally reenact the challenger's game on the client's device, using the gameplay information retrieved from the server. The Ghost Monkey will play side-by-side with the challenged player's monkey – but, being a well-behaved "ghost," will not interact with any sprites in the live game. It is there merely to "haunt" the challenged player with a display of how much better (or worse!) his friend did!

This particular game is pretty simple – all you do is jump over obstacles. So the idea is you'll see when your friend jumps (or dies), and see if you can get further than your friend. In other games, this ghost concept could be even more interesting – imagine a top-down racing game, and being able to see the path your friend takes along the track.

Open your MonkeyJump project in Xcode and create a new file by right-clicking on the **GameObjects** group in the project navigator, selecting **New File...** and then

the **iOS\Cocoa Touch\Objective-C class** template, and finally clicking **Next**. Name the class **GhostMonkey**, and make it a subclass of **Monkey**.



Since the **Monkey** class has all the methods needed to create the walk and jump animations, you don't need to add any additional functionality to it. The only thing you need to do is create a plist file to describe the animations for the ghost monkey.

Create a new file in the plist group and call it **GhostMonkey**. As with the **Monkey.plist** file, create two dictionaries, one for the *jumpAnim* and the other for the *walkAnim*. To make things easy, you can simply copy over the contents of the **Monkey.plist** file here, but do make sure to change the *filenamePrefix*.

Here is how the final plist should look:

| Key | Type | Value |
|-----------------|------------|-------------------|
| Root | Dictionary | (2 items) |
| jumpAnim | Dictionary | (3 items) |
| filenamePrefix | String | monkey_ghost_jump |
| delay | String | 0.1 |
| animationFrames | String | 1,2,3,4 |
| walkAnim | Dictionary | (3 items) |
| filenamePrefix | String | monkey_ghost_run |
| delay | String | 0.08 |
| animationFrames | String | 1,2,3,4,5,6,7,8 |

Now that you have added a plist for the animations, open the **GhostMonkey.m** file and override `changeState:` by adding the following code to the file:

```
- (void) changeState:(MonkeyState) newState {
```

```
[super changeState:newState];

if(newState == kDead) {
    [self stopAllActions];
    [self setDisplayFrame:
        [[CCSpriteFrameCache
            sharedSpriteFrameCache]
            spriteFrameByName:
            @"monkey_ghost_dead.png"]];
}
```

The reason you're overriding this method is to set the `spriteFrame` to `monkey_ghost_dead.png`. If you had not done this, then the sprite frame would have been `monkey_dead.png`. (Close, but not quite!)

To check if the new `GhostMonkey` class works, open **GameLayer.m** and add the following import statement:

```
#import "GhostMonkey.h"
```

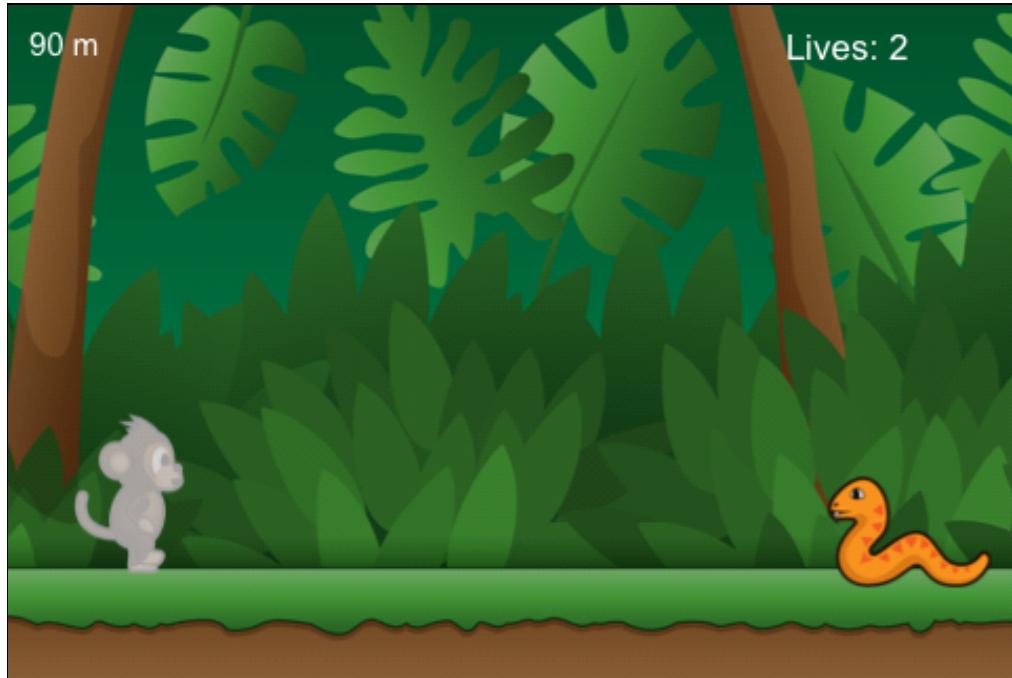
Replace the reference to the `Monkey` class in the class extension with `GhostMonkey`.

```
GhostMonkey *_monkey;
```

And in `onEnter`, change the way the `_monkey` object is initialized to the following:

```
_monkey =
    [GhostMonkey
        spriteWithSpriteFrameName:
        @"monkey_ghost_run1.png"];
```

Build and run the game. Our beloved monkey should now appear as a ghost!



Note: Before you proceed with the next section, please reverse all the changes to the **GameLayer.m** file. These changes were only made to test the **GhostMonkey** class.

Tracking the player's every move

Now that you have a class to represent the challenger, you need a way to track the player's every move. This data will later be sent to your server, a topic we will cover shortly.

The strategy you follow to track gameplay data needs to be lightweight. Here's why:

- The game should not hinder its own run loop while it is tracking the player's movements. In other words, you don't want the game to stutter or pause while you track what the player is doing. Everything should work smoothly without the player ever noticing any performance issues.
- The data that you send to the server needs to be small in size so that you can ensure very little lag while sending and receiving data.

Keeping the above points in mind, here's your strategy:

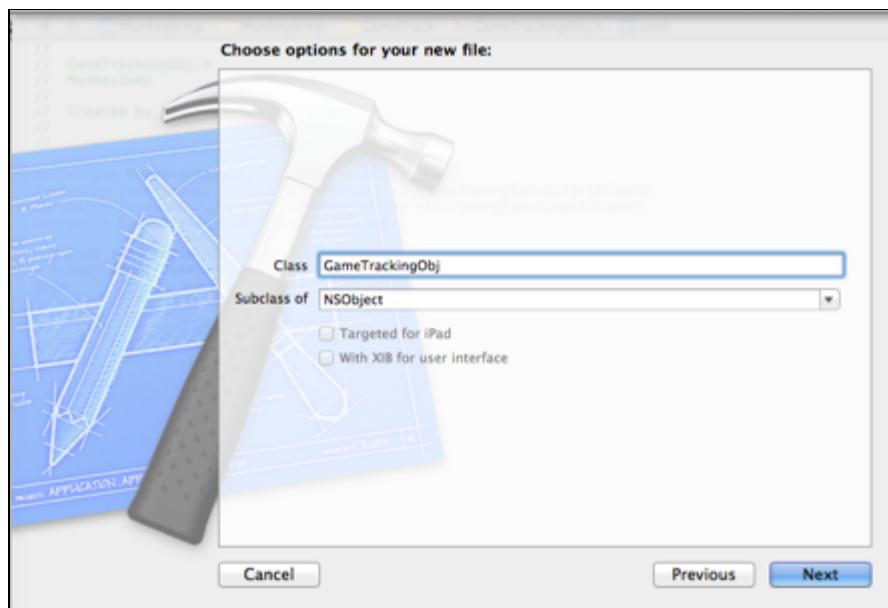
- To make sure that the challenged opponent gets the exact same level as the challenger, i.e., the sequence of enemies that are generated are the same, you need to send the seed that you use for generating random numbers. Since the seed is one long variable, it is extremely lightweight.

- To track where the player jumps, you are going to store the time that has elapsed from the start of the game until the jump. Since the player can jump multiple times, you'll have an array to store all the values.
- You'll have another array to track where the player got hurt. Just as for jump data, you'll store the time that has elapsed from the game start until each hit to the player.

OK! Now there is a clear strategy on the table, let's start implementing it.

Create a new group under the **MonkeyJump** group. To do this, right-click on the **MonkeyJump** group, select **New Group** and name it **GameTrack**.

Now create a new file in this group by right-clicking on the **GameTrack** group, selecting **New File...** and then the **iOS\Cocoa Touch\Objective-C class** template, and finally clicking **Next**. Name the class **GameTrackingObj** and make sure it extends **NSObject**.



Open the **GameTrackingObj.h** file and replace the contents of the file with the following:

```
@interface GameTrackingObj : NSObject<NSCopying>

@property (nonatomic, strong)
NSMutableArray *jumpTimingSinceStartOfGame;

@property (nonatomic, strong)
NSMutableArray *hitTimingSinceStartOfGame;

@property (nonatomic, readonly) long seed;
```

```
- (void) addJumpTime:(double) jumpTime;
- (void) addHitTime:(double) hitTime;
@end
```

Here you have added properties to store the seed value used by the random function, and defined arrays to store the jump and the hit times. Remember you are going to store the elapsed times from the start of the game.

You have also declared two helper methods: one to add a jump time, and the other to add a hit time to their respective arrays.

Now open **GameTrackingObj.m** and add the following lines of code between the `@interface` and `@end` lines:

```
- (id) init {
    self = [super init];
    if (self) {
        self.seed = 1;
    }
    return self;
}

- (void) addJumpTime:(double) jumpTime {

    if (self.jumpTimingSinceStartOfGame
        == nil) {
        self.jumpTimingSinceStartOfGame =
            [NSMutableArray array];
    }
    [self.jumpTimingSinceStartOfGame
        addObject:[NSNumber numberWithDouble:
            jumpTime]];
}

- (void) addHitTime:(double) hitTime {

    if (self.hitTimingSinceStartOfGame
        == nil) {
        self.hitTimingSinceStartOfGame =
            [NSMutableArray array];
    }
    [self.hitTimingSinceStartOfGame
        addObject:[NSNumber numberWithDouble:
            hitTime]];
}
```

The above code is fairly self-explanatory – all you've done is define an `init` method that sets the seed value. You've also added the helper methods for adding jump times and hit times.

Next you need to implement `copyWithZone:`, which is used in case you want to copy the `GameTrackingObj`.

```
- (id)copyWithZone:(NSZone *)zone {
    GameTrackingObj *copy =
        [ [GameTrackingObj allocWithZone:zone]
            init];
    copy.jumpTimingSinceStartOfGame =
        [self.jumpTimingSinceStartOfGame
            mutableCopy];
    copy.hitTimingSinceStartOfGame =
        [self.hitTimingSinceStartOfGame
            mutableCopy];
    copy.seed = self.seed;
    return copy;
}
```

For easy debugging, it's always best practice to override the `description` method of `NSObject`, so go ahead and do that:

```
- (NSString*) description {
    return [NSString stringWithFormat:
        @"Jump times: %@", Hit times: %@, seed: %ld",
        self.jumpTimingSinceStartOfGame.description,
        self.hitTimingSinceStartOfGame.description,
        self.seed];
}
```

Awesome!! Now let's put this new class to use! Open `GameLayer.m` and add the following import statement:

```
#import "GameTrackingObj.h"
```

Next, add the following variables to the class extension of the `GameLayer.m` file.

```
@interface GameLayer() {
    .
    .
    .
    double _startTime;
    long seed;
```

```
    GameTrackingObj *_gameTrackingObj;  
}  
@end
```

You will use these variables to store the start time, the seed value, and the gameplay information (stored inside the new game tracking object you just created).

Now change `initWithHud:` to the following:

```
- (id) initWithHud:(HudLayer*) hud {  
    self = [super init];  
    if (self) {  
        seed = time(NULL);  
        srand(seed);  
  
        . . .  
        _gameTrackingObj =  
            [[GameTrackingObj alloc] init];  
        _gameTrackingObj.seed = seed;  
    }  
    return self;  
}
```

The only new things happening here are the initialization of the `GameTrackingObj` and the seed value.

Next, initialize the `_startTime` variable in `onEnterTransitionDidFinish`. You're doing this here because the game's run loop is started in this method.

```
- (void) onEnterTransitionDidFinish {  
    [super onEnterTransitionDidFinish];  
  
    . . .  
    NSDate *startDate = [NSDate date];  
    _startTime = [startDate timeIntervalSince1970];  
  
    . . .  
}
```

To store the times when the player jumps, change `ccTouchBegan:withEvent:` to the following:

```
- (void) ccTouchesBegan:(NSSet *) touches withEvent:(UIEvent *) event  
{  
    if (! _jumping) {  
        _jumping = YES;
```

```
NSDate *jumpDate = [NSDate date];

double jumpTimeSinceStart =
    [jumpDate timeIntervalSince1970] - _startTime;

[_gameTrackingObj addJumpTime:
    jumpTimeSinceStart];

. . .

}
```

Since the monkey jumps when the player taps the screen, it makes sense to record the jump times in `ccTouchesBegan:withEvent:`.

To record the times when the player gets hurt, change `update:` to the following:

```
- (void) update:(ccTime)deltaTime {

    . . .

    //check for collisions
    if (!_isInvincible) {
        for (CCSprite *sprite in _spriteBatchNode.children) {
            if (sprite.tag == kMonkeyTag)
                continue;
            else {
                . . .
                if (CGRectIntersectsRect
                    (_monkey.boundingBox, enemyRect)) {

                    _isInvincible = YES;
                    _monkey.lives -= 1;

                    NSDate *hitDate = [NSDate date];

                    double hitTimeSinceStart =
                        [hitDate timeIntervalSince1970]
                            - _startTime;
                    [_gameTrackingObj
                        addHitTime:hitTimeSinceStart];
                }
            }
        }
    }
}
```

```
        }
    }
}
```

Now that you can track gameplay information, you need a way to send this information to your web server.

Adding the AFNetworking library

For network communication, you're going to use a library called **AFNetworking**. This is a popular library created by the makers of Gowalla that makes networking calls much easier.

To add this library to your project, follow these steps:

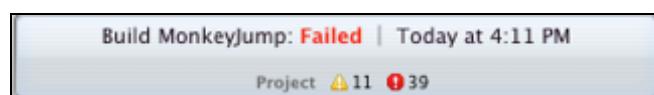
1. Download the latest version of the library from the AFNetworking GitHub page (<https://github.com/AFNetworking>) – you can use the convenient ZIP download option and extract the ZIP file to a location on your hard drive.
2. Switch to your project in Xcode, select the project root in the project navigator, and then select File > Add files to “MonkeyJump”.

You will be presented with a file browser dialog. In the file browser, go to the folder where you extracted AFNetworking and select the AFNetworking sub-folder within the main folder (be sure to select the folder itself, rather than any of the files in the directory).

3. Next, check these options below the file browser:
 - a. Destination: Copy items into destination group’s folder (if needed) [checked].
 - b. Folders: Create group for any added folders [Top item selected].
 - c. Add to Targets: [All targets selected].

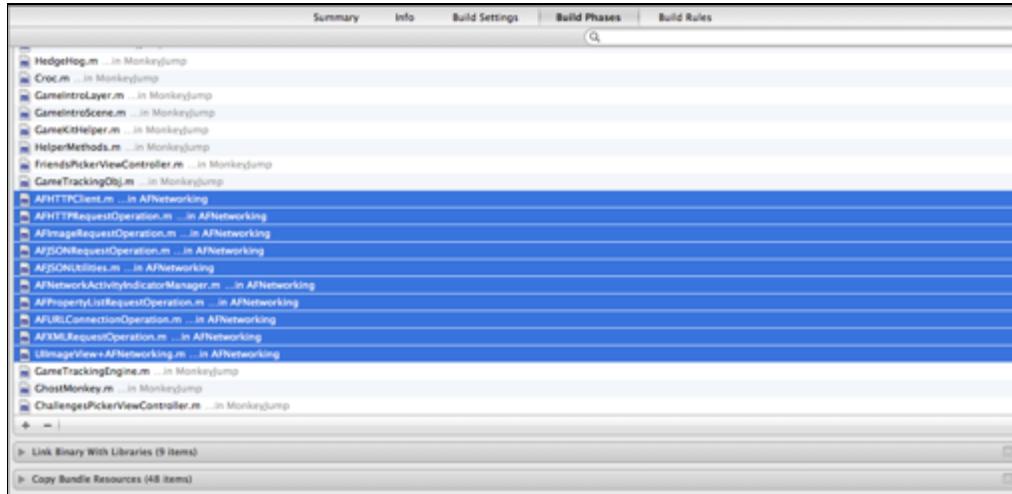
Jeez, a lot of steps, huh? But there are no more. ☺ You've now successfully added the AFNetworking library to the MonkeyJump project.

Try building the application – and oh no, there are a bunch of errors!



Do not panic! ☺ This is just because the library does not support ARC, which you can fix easily by disabling ARC for just the AFNetworking files.

To do this, click on the project root in the project navigator and open the **Build Phases** tab. Next expand the **Compile Sources** section and select all the AFNetworking files, as shown below:



Tap Enter on the keyboard and a popup text window should open. Add the `-fno-objc-arc` flag in this window. This will disable ARC only for the selected files.



Now build the application, and everything should build smoothly without any errors.

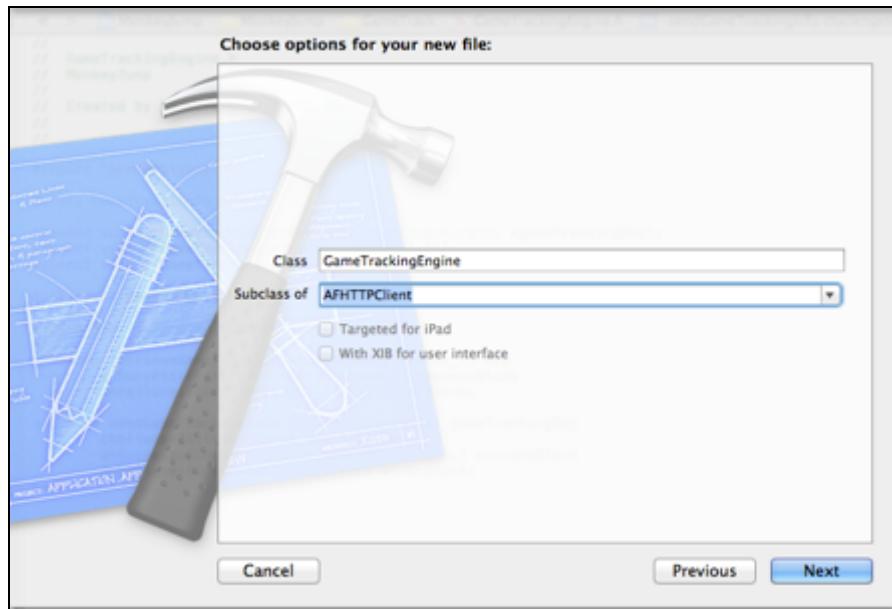
I knew it would all be OK.



Adding network communication

Now that you have integrated a networking library, you need to write the code to use it so that you can send and receive data from your web server.

To do this, create a new class under the **GameTrack** group using the Objective-C class template. Name the class **GameTrackingEngine**, and make sure that this class extends **AFHTTPClient**, as shown below:



Open **GameTrackingEngine.h** and replace the contents with the following:

```
#import "AFHTTPClient.h"

@class GameTrackingObj;

typedef void (^GameTrackingObjResponse)
    (GameTrackingObj *gameTrackingObj);

typedef void
(^GameTrackingObjSentSuccessfully)();

typedef void
(^GameTrackingObjError)(NSError *error);

@interface GameTrackingEngine : AFHTTPClient

+ (GameTrackingEngine *)sharedClient;

- (void) retrieveGameTrackingDetailsForKey:(int64_t) key
    onSuccess:(GameTrackingObjResponse) responseBlock
    onFailure:(GameTrackingObjError) errorBlock;

- (void) sendGameTrackingInfo:(GameTrackingObj*) gameTrackingObj
    challengeId:(NSNumber*) challengeId
    onSuccess:(GameTrackingObjSentSuccessfully) successBlock
    onFailure:(GameTrackingObjError) errorBlock;

@end
```

Here you have added three blocks:

1. `GameTrackingObjResponse`: Since the request to the server will be sent asynchronously, this block is used to send the parsed response back to the caller in the form of a `GameTrackingObj`.
2. `GameTrackingObjSentSuccessfully`: Sending data to your server is again asynchronous. So, this block is used to tell the caller that the request was sent successfully.
3. `GameTrackingObjError`: This block is used to inform the caller when an error has occurred.

Along with these three blocks, you have also added helper methods to send and receive data. Note that this class uses a singleton pattern.

Next, open **GameTrackingEngine.m** and define each method. First define the class method as follows:

```
+ (GameTrackingEngine *)sharedClient {

    static GameTrackingEngine *sharedClient = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedClient = [[GameTrackingEngine alloc]
                       initWithBaseURL:[NSURL
                                     URLWithString:
                                     BASE_URL]];
    });

    return sharedClient;
}
```

Here you initialize a single `GameTrackingEngine` object. Since the `GameTrackingEngine` class extends `AFHTTPClient`, you will use `initWithBaseURL:` for initialization.

Note: `AFHTTPClient` provides a convenient interface to interact with web services using default headers, authentication, network reachability monitoring, batched operations, query string parameter serialization, and multipart form requests.

The `BASE_URL` used above will change depending on how your web server is set up. Since I deployed my server locally, I have defined this in the **Prefix.pch** file as

```
#define BASE_URL @"http://192.168.0.100"
```

Add the above line to **Prefix.pch**, but modify the URL to match your server configuration.

Next, override `initWithBaseURL:` in **GameTrackingEngine.m**:

```
- (id) initWithBaseURL:(NSURL *)url {
    self = [super initWithBaseURL:url];
    if (self) {
        self.parameterEncoding =
            AFJSONParameterEncoding;
        [self registerHTTPOperationClass:
            [AFJSONRequestOperation class]];
        [self setDefaultHeader:@"Accept"
            value:@"application/json"];
    }
    return self;
}
```

Since the format of the data you send and receive is going to be JSON, you have specified `AFJSONParameterEncoding` as the encoding type. You've also registered `AFJSONRequestOperation`, which will make sure that every time a request is sent or a response is received, the data will be in JSON.

Add the following imports to the top of the file:

```
#import "AFJSONRequestOperation.h"
#import "GameTrackingObj.h"
```

Next you will define `retrieveGameDetailsForKey:onSuccess:onFailure:`. Add the following code:

```
- (void) retrieveGameTrackingDetailsForKey:(int64_t) key
    onSuccess:(GameTrackingObjResponse) responseBlock
    onFailure:(GameTrackingObjError) errorBlock {

    NSDictionary *params = [NSDictionary
        dictionaryWithObject:
        [NSNumber numberWithInt:key]
        forKey:@"challengeId"];

    [self getPath:@"ChallengesAPI.php"
        parameters:params
        success:^(AFHTTPRequestOperation *operation,
        id responseObject) {
```

```
if (responseBlock != nil &&
    responseObject != nil &&
    [responseObject isKindOfClass:
     [NSDictionary class]]) {

    GameTrackingObj *gameTrackingObj =
        [[GameTrackingObj alloc] init];

    gameTrackingObj.jumpTimingSinceStartOfGame =
        [responseObject objectForKey:
         kJumpArrayKey];

    gameTrackingObj.hitTimingSinceStartOfGame =
        [responseObject objectForKey:
         kHitArrayKey];

    gameTrackingObj.seed =
        [[responseObject objectForKey:
         kSeekKey] intValue];

    responseBlock(gameTrackingObj);
}

} failure:^(AFHTTPRequestOperation *operation,
           NSError *error) {
    if (errorBlock != nil) {
        errorBlock(error);
    }
}];
}
```

This method is used to retrieve information from your server. It queries the server with a GET request and a challengeId. Then it parses the response it receives and creates a new GameTrackObj. Finally, it gives this object to the caller using the block defined previously.

Add the following `#defines` below the `#import` lines to define constant values needed by the code above:

```
#define kSeekKey @"Seed"
#define kJumpArrayKey @"JumpArray"
#define kHitArrayKey @"HitArray"
```

You now have a method to retrieve information. Next, write the method to send game play information. Add the following code:

```
- (void) sendGameTrackingInfo:  
    (GameTrackingObj*) gameTrackingObj  
challengeId:  
    (NSNumber*) challengeId  
onSuccess:  
    (GameTrackingObjSentSuccessfully) successBlock  
onFailure:  
    (GameTrackingObjError) errorBlock {  
  
    NSDictionary *params =  
        [NSDictionary dictionaryWithObjectsAndKeys:  
            [NSNumber numberWithInt:gameTrackingObj.seed],  
            kSeekKey,  
            gameTrackingObj.jumpTimingSinceStartOfGame,  
            kJumpArrayKey,  
            gameTrackingObj.hitTimingSinceStartOfGame,  
            kHitArrayKey, nil];  
  
    NSString *postPath = [  
        NSString stringWithFormat:  
            @"ChallengesAPI.php?challengeId=%d",  
            challengeId.intValue];  
  
    [self postPath:postPath  
        parameters:params  
        success:^(AFHTTPRequestOperation *operation,  
                 id responceObject){  
        if (successBlock != nil) {  
            successBlock();  
        }  
    } failure:^(AFHTTPRequestOperation *operation,  
                NSError *error){  
        if (errorBlock != nil) {  
            errorBlock(error);  
        }  
    }];  
}
```

This method takes a `GameTrackingObj` as a parameter and creates an `NSDictionary` of all the properties of that `GameTrackingObj`. This dictionary is sent in the body of the POST request.

Awesome! You're armed and ready to use this class in your game to send and receive gameplay data.

Sending gameplay information

In order to send gameplay information to your server, you need to make a few changes to the way the game was written. These changes are required to pass the `GameTrackingObj` between different scenes and layers.

First open `GameOverScene.h` and forward-declare the `GameTrackingObj` class:

```
@class GameTrackingObj;
```

Next change the `initWithScore:` method declaration to the following:

```
- (id) initWithScore:(int64_t)score
    gameTrackingObj:(GameTrackingObj*)gameTrackingObj;
```

Now open `GameOverScene.m` and replace the initialization method with the following:

```
- (id) initWithScore:(int64_t)score
    gameTrackingObj:(GameTrackingObj*)gameTrackingObj {
    self = [super init];
    if (self) {
        GameOverLayer *gameOverLayer =
            [[GameOverLayer alloc]
                initWithScore:score
                gameTrackObj:gameTrackingObj];
        [self addChild:gameOverLayer];
    }
    return self;
}
```

Notice here that the code initializes not only the `GameOverScene`, but also the `GameOverLayer` using a new method that accepts a `GameTrackingObj`. Since you haven't yet written the new initialization method for the `GameOverLayer`, you should do that next – but first, switch to `GameLayer.m` and replace the last line in `monkeyDead` with the following, so as to use the new initializer for `GameOverScene`:

```
[[CCDirector sharedDirector] replaceScene:[[GameOverScene alloc]
    initWithScore:(int64_t)_distance
    gameTrackingObj:_gameTrackingObj]];
```

Now open `GameOverLayer.h` and add a forward class declaration again:

```
@class GameTrackingObj;
```

Next, change the previous initialization method to the following:

```
- (id) initWithScore:(int64_t)score
    gameTrackObj:(GameTrackingObj*) gameTrackObj;
```

Now open **GameOverLayer.m** and add the following import:

```
#import "GameTrackingObj.h"
```

Then, declare a GameTrackingObj variable in the class extension:

```
@interface GameOverLayer() {
    . . .
    GameTrackingObj *_gameTrackObj;
}
@end
```

And replace the old initialization method with the following:

```
- (id) initWithScore:(int64_t)score
    gameTrackObj:(GameTrackingObj*) gameTrackObj {
    self = [super init];
    if (self) {
        _score = score;
        _gameTrackObj = gameTrackObj;
    }
    return self;
}
```

This new method only stores a reference to the GameTrackingObj. You now need to pass this reference to the FriendsPickerController, since ultimately the challenge is sent through that view controller.

Open **GameKitHelper.h** and add a forward class declaration again:

```
@class GameTrackingObj;
```

Then change the method declaration for `showFriendsPickerControllerForScore:` to:

```
-(void)showFriendsPickerControllerForScore:(int64_t)score
    gameTrackObj:(GameTrackingObj*)gameTrackObj;
```

Next, open **GameKitHelper.m** and replace the `showFriendsPickerControllerForScore:` method with the following:

```
-(void)
    showFriendsPickerControllerForScore:
    (int64_t)score gameTrackObj:(GameTrackingObj*) gameTrackObj {
```

```

FriendsPickerController
    *friendsPickerController =
        [[FriendsPickerController alloc]
            initWithScore:score gameTrackObj:gameTrackObj];

friendsPickerController.
    cancelButtonPressedBlock = ^{
        [self dismissModalViewControllerAnimated];
};

friendsPickerController.
    challengeButtonPressedBlock = ^{
        [self dismissModalViewControllerAnimated];
};

UINavigationController *navigationController =
    [[UINavigationController alloc]
        initWithRootViewController:
            friendsPickerController];

[self presentViewController:navigationController];
}

```

This method now accepts the `GameTrackingObj` and passes it to the `FriendsPickerController`.

Yes, the `FriendsPickerController` still doesn't support initializing using a `GameTrackobj`. You'll fix that in a moment, but first, open **GameOverLayer.m**, find `menuButtonPressed:`, and replace the code used to display the `FriendsPickerController` with the following:

```

- (void) menuButtonPressed:(id) sender {
    CCMenuItemLabel *menuItem = (CCMenuItemLabel*) sender;

    . . .

    else if(menuItem.tag == kChallengeFriendsButtonTag) {
        [[GameKitHelper sharedGameKitHelper]
            showFriendsPickerControllerForScore:
                _score
                gameTrackObj:_gameTrackObj];
    }
    . . .
}

```

Now open **FriendsPickerController.h** and add another forward declaration:

```
@class GameTrackingObj;
```

Then, change the initialization method declaration.

```
- (id)initWithScore:(int64_t) score
    gameTrackObj:(GameTrackingObj*) gameTrackObj;
```

Next open **FriendsPickerController.m** and add an instance variable to the class extension:

```
@interface FriendsPickerController ()  
    <UITableViewDataSource,  
     UITableViewDelegate,  
     UITextFieldDelegate,  
     GameKitHelperProtocol> {  
  
    . . .  
    GameTrackingObj *_gameTrackObj;  
    . . .  
}  
. . .  
@end
```

And change the initialization routine as follows:

```
- (id)initWithScore:(int64_t) score
    gameTrackObj:
        (GameTrackingObj*) gameTrackObj {
    self = [super  
        initWithNibName:  
        @"FriendsPickerController"  
        bundle:nil];  
  
    if (self) {
        . . .
        _gameTrackObj = gameTrackObj;
        . . .
    }
    return self;
}
```

Now you need to change the method the game uses to send the challenge to selected friends in the `FriendsPickerController` – the `sendScoreChallengeToPlayers:withScore:message:` method found in the `GameKitHelper` class.

Instead of modifying this method, you are going to add a new method to **GameKitHelper.m** that will accept the `GameTrackingObj` as a parameter. Add this code:

```
- (void) sendScoreChallengeToPlayers:  
    (NSArray *)players  
    withScore:(int64_t)score  
    message:(NSString *)message  
    withGameTrackingObj:  
        (GameTrackingObj*)gameTrackingObj {  
  
    // 1  
    if (gameTrackingObj != nil) {  
  
        // 2  
        NSNumber *challengeId =  
            [NSNumber numberWithInt:  
                (arc4random() % 10000 + 1)];  
  
        // 3  
        [[GameTrackingEngine sharedClient]  
            sendGameTrackingInfo:gameTrackingObj  
            challengeId:challengeId  
            onSuccess:^(){  
                // 4  
                [self sendScoreChallengeToPlayers:players  
                    score:score  
                    context:challengeId.integerValue  
                    message:message];  
            } onFailure:^(NSError *error){  
                // 5  
                [self sendScoreChallengeToPlayers:players  
                    score:score  
                    context:0  
                    message:message];  
            }];  
    } else {  
        // 6  
        [self sendScoreChallengeToPlayers:players  
            score:score context:0  
            message:message];  
    }  
}  
  
- (void) sendScoreChallengeToPlayers:(NSArray*)players  
    score:(uint64_t) score
```

```
        context:(uint64_t) context
        message:(NSString*) message {

    GKScore *gkScore =
        [[GKScore alloc]
         initWithCategory:
         kHighScoreLeaderboardCategory];

    gkScore.context = context;
    gkScore.value = score;

    [gkScore issueChallengeToPlayers:players
        message:message];
}
```

I'll explain this code shortly, but first, import the `GameTrackingEngine` class at the top of `GameKitHelper.m`:

```
#import "GameTrackingEngine.h"
```

And don't forget to add the method declaration/prototype for this method to `GameKitHelper.h`:

```
-(void)sendScoreChallengeToPlayers:(NSArray *)players
withScore:(int64_t)score message:(NSString *)message
withGameTrackingObj:(GameTrackingObj*)gameTrackingObj;
```

Now to the step-by-step explanation of the `sendScoreChallengeToPlayers:withScore:message:withGameTrackingObj:` method:

1. The method first checks to see if gameplay information is being sent. If not, it sends a challenge with the `context` set to 0.
2. If gameplay information *is* being sent, the method generates a random `challengeId`. This ID is then sent in the `context` property of the challenge.
3. The method then stores the gameplay information on your private server with the random `challengeId` generated in step #2.
4. If gameplay information is sent to the server successfully, then the challenge is sent, keeping the `context` as the `challengeId`.
5. If an error occurred while sending gameplay information, the challenge is still sent, keeping the `context` as 0.

Note: The `context` property allows your game to associate an arbitrary 64-bit unsigned integer value with the score data reported to Game Center. You decide how this integer value is interpreted by your game.

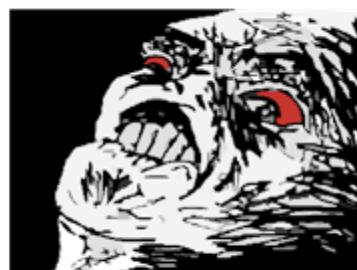
When you send out a `GKScoreChallenge` with the `context` value set, you're guaranteed that the same value is available at the querying end. Hence, you send out `challengeId` as a context.

You now have a method to send gameplay information to your web server and to send a reference (`challengeId`) of that information through a `GKScoreChallenge`. Let's put this method to the test.

Open **FriendsPickerController.m** and change `challengeButtonPressed:` so that it uses the new method you added in `GameKitHelper`.

```
- (void)challengeButtonPressed:  
    (id) sender {  
    . . .  
    if (playerIds.count > 0) {  
  
        [ [GameKitHelper sharedGameKitHelper]  
            sendScoreChallengeToPlayers:playerIds  
            withScore:_score  
            message:self.challengeTextField.text  
            withGameTrackingObj:_gameTrackObj];  
    }  
  
    . . .  
}  
. . .  
}
```

Awesome! You now have all the code in place. Ready to test it out?



OH how I have been waiting for this moment ...

Build and run the application, make sure that your server is up and running and that the database tables have been created. Set a breakpoint at the place where the `challengeId` is generated (in **GameKitHelper.m**) by clicking in the side gutter on the line you can see in the screenshot below:

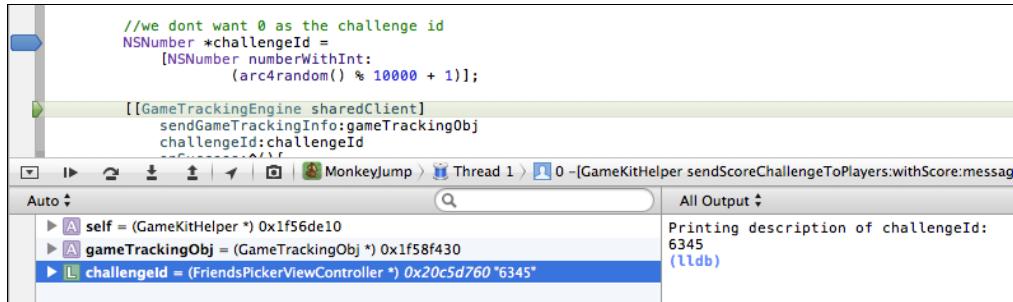
```

-(void) sendScoreChallengeToPlayers:
    (NSArray *)players
    withScore:(int64_t)score
    message:(NSString *)message
    withGameTrackingObj:
        (GameTrackingObj*)gameTrackingObj {
    if (gameTrackingObj != nil) {
        //we dont want 0 as the challenge id
        NSNumber *challengeId =
            [NSNumber numberWithInt:
                (arc4random() % 10000 + 1)];
        [[GameTrackingEngine sharedClient]
            sendGameTrackingInfo:gameTrackingObj
            challengeId:challengeId
            onSuccess:^({
                [self sendScoreChallengeToPlayers:players
                    score:score
                    context:challengeId.integerValue
                    message:message];
            } onFailure:^(NSError *error){
                [self sendScoreChallengeToPlayers:players
                    score:score
                    context:0
                    message:message];
            });
    } else {
        [self sendScoreChallengeToPlayers:players
            score:score context:0
            message:message];
    }
}

```

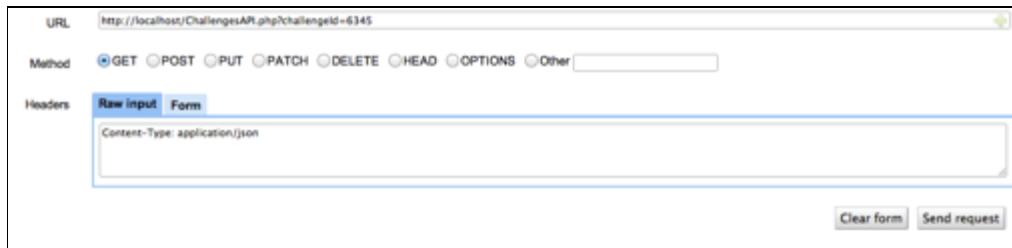
Play a round of the game, and then challenge a few of your friends using the FriendsPickerController. When you send the challenge, the debugger will break at the breakpoint you just added.

Note the challengeId that was indeed generated, as shown below (you can simply right-click on the challengeId variable in the variable view and select Print Description of “variable”):



With the challengeId noted, let execution continue by clicking the “continue” button in the debugger. Next open up a REST client like the Advanced REST Client mentioned earlier, and execute a GET request with the same challengeId.

Here is a screenshot showing the request:



And the response received should be something like this:

Status code **200 OK**

Time 5 ms

Headers Date: Mon, 27 Aug 2012 07:15:10 GMT
X-Powered-By: PHP/5.3.1
Connection: Keep-Alive
Content-Length: 145
Server: Apache/2.2.14 (Unix) DAV/2 mod_ssl/2.2.14 OpenSSL/0.9.8i PHP/5.3.1 mod_perl/2.0.4 Perl/v5.10.1
Content-Type: text/json
Keep-Alive: timeout=5, max=100

Body Raw response JSON

```
{
  "HitArray": [size(3)
    2.138778924942017,
    6.129381895065308,
    9.74910593032837
  ],
  "Seed": 0,
  "JumpArray": [size(3)
    0.2882349491119385,
    2.134745836257935,
    6.48747181892396
  ]
}
```

If you get a valid response, then stop reading, take a break, and do a small victory dance because you have finished a major chunk of the chapter successfully!

If things didn't go according to plan, you might get an error message indicating what went wrong. One possibility is that you didn't create the MySQL table or the database. Or, something might have been named wrong. Check things and do some debugging until you get it right. Then do your victory dance. ☺

Next up, you'll write some routines to retrieve the gameplay data when a challenge is received and to replay the challenger's moves.

Retrieving gameplay information

You can now send a challenge to an opponent, but you need a way to retrieve the gameplay information at the opponent's end. There are two ways to do this:

1. Show a notification to the user when a challenge is received. When the user taps on the notification, begin downloading gameplay information and start the game in challenge mode.
2. Have an option in the game that shows all of the challenges that have been sent to the local player (even those that have been sent in the past). When s/he taps

on one of the challenges, download the gameplay information for that challenge and start the game.

In this chapter, you are going to implement the second method – you will present a `ChallengePickerController` where the user can select a challenge. This view controller is similar to the `FriendsPickerController`, so I will not explain it in detail.

However, to help you get going with the project quickly, I have added a **ChallengePicker.zip** file to the chapter resources that contains the necessary files for this view controller. Add them to your project, and also be sure to go through the code to see how things are implemented.

To query all the challenges received by the local player, you will use the `loadReceivedChallengesWithCompletionHandler:` method of `GKChallenge`. This will act as the data source for the `ChallengePickerController`.

Here is how the `challengePickerController` UI looks:



When the user selects a challenge from the `ChallengePickerController`, a block is called just as with the `FriendsPickerController`. And that block is encompassed in a `showChallengePickerController` method in **GameKitHelper.m**. Add this method to the file:

```
-(void) showChallengePickerController {
    //1
    ChallengesPickerController
    *challengesPickerController =
        [[ChallengesPickerController alloc]
            initWithNibName:
            @"ChallengesPickerController"
            bundle:nil];
```

```
//2
challengesPickerController.
    cancelButtonTitle = ^{
        [self dismissModalViewControllerAnimated];
};

challengesPickerController.
    challengeSelectedBlock =
    ^(GKScoreChallenge *challenge) {
        [self dismissModalViewControllerAnimated];

//3
if (challenge.score.context != 0) {
    //valid challenge id
    [[GameTrackingEngine sharedClient]
     retrieveGameTrackingDetailsForKey:
     challenge.score.context
     onSuccess:^(GameTrackingObj *gameTrackingObj){
        //4
        [[CCDirector sharedDirector]
         replaceScene:[CCTransitionProgressRadialCCW
                     transitionWithDuration:1.0f
                     scene:[[GameScene alloc]
                            initWithGameTrackingObj:
                            gameTrackingObj]]];
    } onFailure:^(NSError *error){
        //5
        [[CCDirector sharedDirector]
         replaceScene:[CCTransitionProgressRadialCCW
                     transitionWithDuration:1.0f
                     scene:
                     [[GameScene alloc] init]]];
    }];
} else {
    //6
    [[CCDirector sharedDirector]
     replaceScene:[CCTransitionProgressRadialCCW
                 transitionWithDuration:1.0f
                 scene:[[GameScene alloc] init]]];
}
};

//7
```

```
UINavigationController *navigationController =
    [ [UINavigationController alloc]
        initWithRootViewController:
            challengesPickerController];
    [self presentViewController:navigationController];
}
```

Don't forget to add the following imports to **GameKitHelper.m**:

```
#import "ChallengesPickerController.h"
#import "GameScene.h"
```

Here is a step-wise explanation of the new method:

1. Create the Challenge Picker View Controller.
2. Set up the Cancel button handler, which simply dismisses the Challenge Picker.
3. When a challenge is selected, the block checks the `context` property of the challenge. If it is 0, then it starts the game normally.
4. If the context is not 0, then the block retrieves gameplay information using the `challengeId` as the `context` property of the challenge.
5. Once it receives the gameplay information, the block creates a new instance of `GameScene` in challenge mode.
6. If an error was encountered while receiving the data, then the game is started in normal mode.
7. Finally, display the Challenge Picker so that the user can actually pick a challenge.

The Challenge Picker also needs a helper method added to `GameKitHelper`. So add the method declaration to **GameKitHelper.h** first:

```
-(void)loadChallenges;
```

Also add a delegate method declaration to `GameKitHelperProtocol`:

```
-(void)onChallengesReceived:(NSArray*)challenges;
```

Then add the method implementation to **GameKitHelper.m**:

```
-(void) loadChallenges {
    if (_gameCenterFeaturesEnabled == NO) {
        return;
    }
    [GKChallenge
        loadReceivedChallengesWithCompletionHandler:
```

```
^(NSArray *challenges, NSError *error) {  
  
    [self setError:error];  
  
    BOOL success = (error == nil);  
    if (success &&  
        [_delegate respondsToSelector:  
            @selector(onChallengesReceived:)]) {  
  
        [_delegate onChallengesReceived:challenges];  
    }  
}  
};  
}
```

Now that you have the gameplay information for the challenger, you just need to make sure that the `GhostMonkey` follows the challenger's every move. You do this by allowing the game to start in challenge mode when the game will have accompanying gameplay data.

To implement this, open `GameScene.h` and add a new initialization method as follows:

```
- (id) initWithGameTrackingObj:  
    (GameTrackingObj*) gameTrackingObj;
```

Define this method in the `GameScene.m` file.

```
- (id) initWithGameTrackingObj:  
    (GameTrackingObj*) gameTrackingObj {  
  
    if (self = [super init]) {  
        HudLayer *hudLayer = [HudLayer node];  
        [self addChild:hudLayer z:1];  
  
        GameLayer *gameLayer =  
            [[GameLayer alloc]  
                initWithHud:hudLayer  
                gameTrackObj:gameTrackingObj];  
  
        [self addChild:gameLayer];  
    }  
    return self;  
}
```

There's not much of a difference here – all that's changed is that the `GameScene` now accepts a `GameTrackingObj`, which is then passed to the `GameLayer`.

Open **GameLayer.h** and declare a new initialization method:

```
- (id) initWithHud:(HudLayer *)hud
              gameTrackObj:(GameTrackingObj*) gameTrackObj;
```

Next, add a property to store the `GameTrackingObj` in the class extension in **GameLayer.m**:

```
@property (nonatomic, copy) GameTrackingObj *challengerGame;
```

Now add the method in **GameLayer.m** as follows:

```
- (id) initWithHud:(HudLayer *)hud
              gameTrackObj:(GameTrackingObj*)
              gameTrackObj {
    self = [self initWithHud:hud];
    if (self) {
        self.challengerGame = gameTrackObj;

        seed = gameTrackObj.seed;
        srand(seed);
        _gameTrackingObj.seed = seed;
    }
    return self;
}
```

That sets up everything you need to start a new game with a `GameTrackingObj` instance containing gameplay data. Now you need to use that gameplay information to display the Ghost Monkey on screen.

Add the following code to the end of `onEnter` in **GameLayer.m**:

```
if (self.challengerGame) {
    _ghostMonkey =
        [GhostMonkey spriteWithSpriteFrameName:
            @"monkey_ghost_run1.png"];
    _ghostMonkey.position =
        ccp(0.125 * _winSize.width,
            0.271 * _winSize.height);
    [_ghostMonkey setTag:kGhostMonkeyTag];
    [_spriteBatchNode addChild:_ghostMonkey];
}
[_spriteBatchNode addChild:_monkey];
```

The changes above will add `GhostMonkey` to the game layer only when the challenger's gameplay data is available.

But that code introduces the need for a new instance variable for the Ghost Monkey. So add the following import first:

```
#import "GhostMonkey.h"
```

Next, add the instance variable to the class extension at the top:

```
GhostMonkey *_ghostMonkey;
```

Also add the following `#define` to the other `defines` at the top of the file:

```
#define kGhostMonkeyTag 5
```

Next add the following code to the end of `onEnterTransitionDidFinish`:

```
if (self.challengerGame) {
    [_ghostMonkey changeState:kWalking];
    [self schedule:@selector(updateGhostMonkeyMoves:)];
}
[self scheduleUpdate];
```

Notice that the code schedules a method named `updateGhostMonkeyMoves:`, which will execute every frame. This is where you will write the code to control the `GhostMonkey`.

Add that method to **GameLayer.m**:

```
- (void) updateGhostMonkeyMoves:
    (ccTime) deltaTime {

    // 1
    NSDate *date = [NSDate date];
    double currentTimeSinceStart =
        [date timeIntervalSince1970]
        - _startTime;

    //2
    [self.challengerGame.jumpTimingSinceStartOfGame
    enumerateObjectsUsingBlock:
     ^(id obj, NSUInteger idx, BOOL *stop){

        NSNumber *jumpTime = (NSNumber*) obj;
        if (jumpTime.doubleValue
            <= currentTimeSinceStart) {

            *stop = YES;
            [self.challengerGame.jumpTimingSinceStartOfGame
```

```
removeObject:jumpTime];

CCJumpBy *jumpAction =
[CCJumpBy actionWithDuration:1.2
    position:ccp(0,0)
    height:120
    jumps:1];

CCCallFunc *doneJumpAction =
[CCCallFunc actionWithTarget:self
    selector:@selector(doneGhostJump)];

CCSequence *sequenceAction =
[CCSequence actions:
    jumpAction,doneJumpAction, nil];

[_ghostMonkey changeState:kJumping];
[_ghostMonkey runAction:sequenceAction];
}

};

//3
[self.challengerGame.hitTimingSinceStartOfGame
enumerateObjectsUsingBlock:
^(id obj, NSUInteger idx, BOOL *stop){

NSNumber *hitTime = (NSNumber*)obj;
if (hitTime.doubleValue <= currentTimeSinceStart) {

    [self.challengerGame.hitTimingSinceStartOfGame
        removeObject:hitTime];
    *stop = YES;

    id action =
    [CCSequence actions:
        [CCBlink
            actionWithDuration:1.5
            blinks:4],
        [CCShow action],
        [CCCallFunc
            actionWithTarget:self
            selector:
            @selector(doneGhostTakingDamage)],
        nil];
}
```

```
[_ghostMonkey runAction:action];

// 4
_ghostMonkey.lives -= 1;

if(_ghostMonkey.lives <= 0) {
    _ghostMonkey.position =
        ccp(0.125 * _winSize.width,
            0.271 * _winSize.height);

    [_ghostMonkey changeState:kDead];
    [self unschedule:
        @selector(updateGhostMonkeyMoves:)];
    [self scheduleOnce:
        @selector(ghostMonkeyDead) delay:0.5];
}
}

};

}
```

Even though this routine looks big, it's not very difficult to understand. Here is a step-by-step explanation of the method:

1. First you calculate the time elapsed since the start of the game.
2. The method enumerates over all the jump times of the challenger. If it finds a time that is equal to or lesser than the current time from start, it will make the GhostMonkey jump. Of course, the code also removes that particular time value from the `jumpTimingSinceStartOfGame` array, so that it doesn't get processed again.
3. The method enumerates over all the hit times of the challenger. If it finds a time that is equal to or lesser than the current time from start, it will reduce the life of the Ghost Monkey by 1. The code also removes the time value from the hit timing array.

Now add some helper methods, which get called when the `GhostMonkey` finishes the jump and hit animation, to **GameLayer.m**:

```
- (void)doneGhostJump {
    [_ghostMonkey changeState:kWalking];
}

- (void)doneGhostTakingDamage {
    [_ghostMonkey changeState:kWalking];
}

- (void) ghostMonkeyDead {
```

```

[_ghostMonkey removeFromParentAndCleanup:YES];
}

```

Also make a slight change to the `update:` method

```

- (void) update:(ccTime)deltaTime {

    . . .
    //check for collisions
    if (! _isInvincible) {
        for (CCSprite *sprite in _spriteBatchNode.children) {
            if (sprite.tag == kMonkeyTag
                || sprite.tag == kGhostMonkeyTag)
                continue;
            else {
                . . .
            }
        }
    }
}

```

The change here is a check added for the `GhostMonkey` sprite. This will prevent the `update` method from checking for collisions between the `GhostMonkey` and other sprites. (It's a ghost, after all!)

Are you tired of making changes to the game? Don't worry, there aren't any more! ☺ You've now got all the code you need to run the game in challenge mode with the Ghost Monkey enacting the challenger's gameplay data.

The only thing that's missing is a button to launch the `ChallengePickerController`. Let's add it so that you can test challenges.

You'll add the launch button to the `MenuScene`. Open `MenuLayer.m` and add the following `define` statement above the implementation section:

```
#define kChallengesReceivedButtonTag 3
```

Next, change the `onEnter` method to the following in order to add a menu item to the current `CCMenu`:

```

- (void) onEnter {
    . . .
    CCLabelBMFont *challengesReceived =
        [CCLabelBMFont
            labelWithString:@"Challenges received"
            fntFile:@"jungle.fnt"];
}

```

```

CCMenuItemLabel *challengesReceivedItem =
    [CCMenuItemLabel
        itemWithLabel:challengesReceived
        target:self
        selector:@selector(menuButtonPressed:)];

challengesReceivedItem.tag =
    kChallengesReceivedButtonTag;
challengesReceivedItem.position =
    ccp(winSize.width * 0.5, winSize.height * 0.3);

CCMenu *menu =
    [CCMenu menuWithItems:newGameItem,
        gameCenterItem,
        challengesReceivedItem,
        nil];
menu.position = CGPointMakeZero;
[self addChild:menu];
}

```

The above code is quite self-explanatory – when the Challenges Received button is pressed, the `menuButtonPressed:` selector is invoked. So let's change the `menuButtonPressed` method to handle the button action and launch the `ChallengePickerController`:

```

- (void) menuButtonPressed:(id) sender {
    CCMenuItemLabel *menuItemLabel =
        (CCMenuItemLabel*)sender;

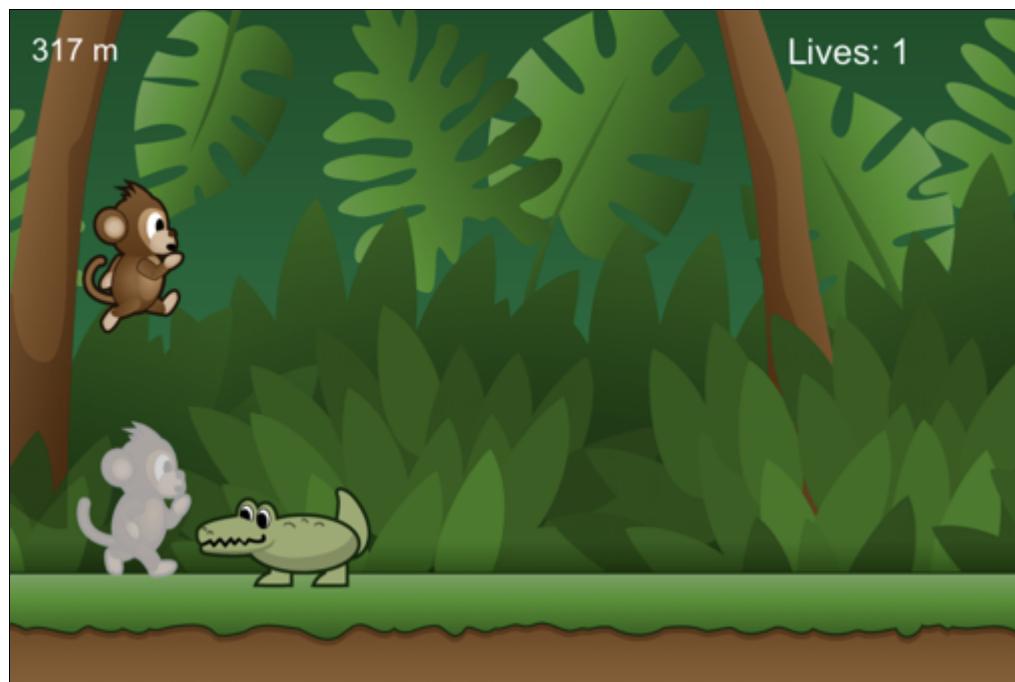
    if (menuItemLabel.tag == kNewGameButtonTag) {
        [[CCDirector sharedDirector]
            replaceScene:
                [CCTransitionProgressRadialCCW
                    transitionWithDuration:1.0f
                    scene:[[GameScene alloc] init]]];
    } else if (menuItemLabel.tag == kGameCenterButtonTag) {
        [[GameKitHelper sharedGameKitHelper]
            showGameCenterViewController];
    } else {
        [[GameKitHelper sharedGameKitHelper]
            showChallengePickerController];
    }
}

```

Build and run the game. Now you'll see a new button on the menu page, which when pressed will launch the `challengePickerController`.



That's it! It time to test if everything works. To do this, you will need two devices. Send out a challenge from one to the other, and if you accept the challenge on the other using the `challengePickerController`, you will see your moves being repeated as you try to beat your earlier score.



When you see a monkey ghost, in your neighborhood – who you gonna call? ☺

Where to go from here?

Wow, you have accepted and trounced all the challenges I've thrown your way over these two chapters, and they weren't all easy! You now have a good familiarity with the new APIs introduced in iOS 6 for GameKit, and I bet you've already started thinking about how to make them work for you.

Let's review your achievements. From the beginning of the last chapter through the end of this one, you've learned how to:

- Set up everything needed to activate Game Center features on iTunes Connect.
- Use leaderboards and achievements in the game.
- Show leaderboards and achievements using the built-in `GKGameCenterViewController`.
- Share scores and achievements using the new `UIActivityViewController`.
- Send score challenges to friends using `GKScoreChallenge`.
- Use the `context` property to store a value related to a `GKScore` or `GKAchievement`.
- Track gameplay information, and send and retrieve it from a private server.
- Replay each move of the opponent based on the retrieved information.

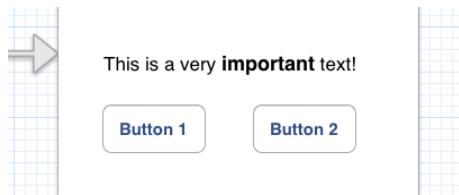
I hope you are excited about the new changes added to the GameKit framework, and inspired to implement some of this new technology in your own games. Please drop us a line if you found these chapters useful, or if you just want to show off your game! ☺

Chapter 15: What's New with Attributed Strings

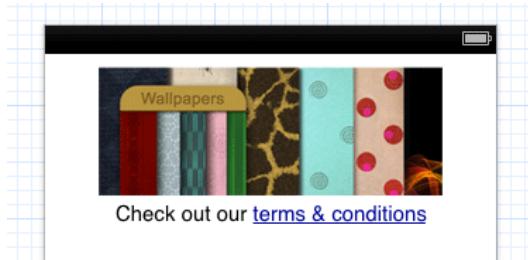
By Marin Todorov

Cocoa Touch is absolutely amazing if you need to complete a sophisticated task, like finding out where in the world the user is located based on their GPS coordinates, making an HD video and sending it over to your web server, or getting the compass direction that the user is facing.

But for quite some time now, it's been relatively complicated to do what should be simple text formatting – like putting a bold word in the middle of a sentence when using a `UILabel`, for example:



And I've often found myself wishing for a button that mimics a hyperlink, like the one in the screenshot below:



Until iOS 6, this was only possible using Core Text, which is indeed a very powerful text layout framework. But Core Text is not so straightforward to use – mostly because it's not tightly integrated with UIKit, but also because it's a C-based, rather than Objective-C-based, framework.

However, now performing simple text formatting with UIKit controls is now super-easy, thanks to the new attributed strings functionality in iOS 6!



The new attributed strings functionality in iOS 6 allows you to use Interface Builder to easily apply rich text formatting to different text elements like `UITextView`, `UIButton`, `UILabel`, and more.

You can also create rich text via code and display it using UIKit elements. Last but definitely not least, users can easily apply formatting (such as bolding or italics) to their own text as well, due to the new rich text editing capability in controls like `UITextView`!

In this chapter, you'll try out the new attributed strings functionality and learn how to integrate it into your own apps. This chapter is split into two parts:

1. In the **first part** of the chapter, you will learn the basics of using attributed strings. You'll learn how to create them and how to apply many different formatting styles to them. You will also learn about displaying attributed strings with `UILabel`.
2. In the **second part** of the chapter, you will learn how to allow the user to richly format their own text. You'll build a project similar to the Notes app that comes with iOS 6 – except it will be a real rich text editor for notes! You will learn in detail about nearly all of the new features of `NSAttributedString`.

At this point you're probably eager to get started, so let's dive right in!

Formatting text with attributed strings

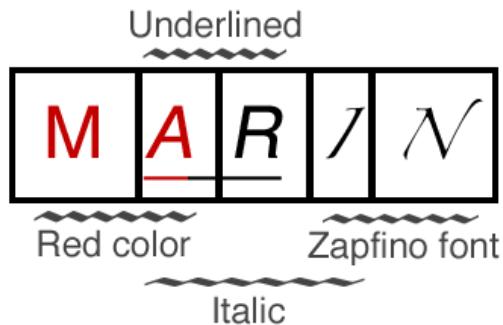
You've probably encountered the `NSString` object a billion times, and it has served you well throughout your Objective-C projects.

Some of you may also have used `NSAttributedString`, which is a special class that can work with rich text. `NSAttributedString` is not a subclass of `NSString` – each class has a very different purpose, and each inherits from `NSObject`.

`NSString`'s main aim is to store information like this:



On the other hand, `NSAttributedString`'s task is to store this kind of data:

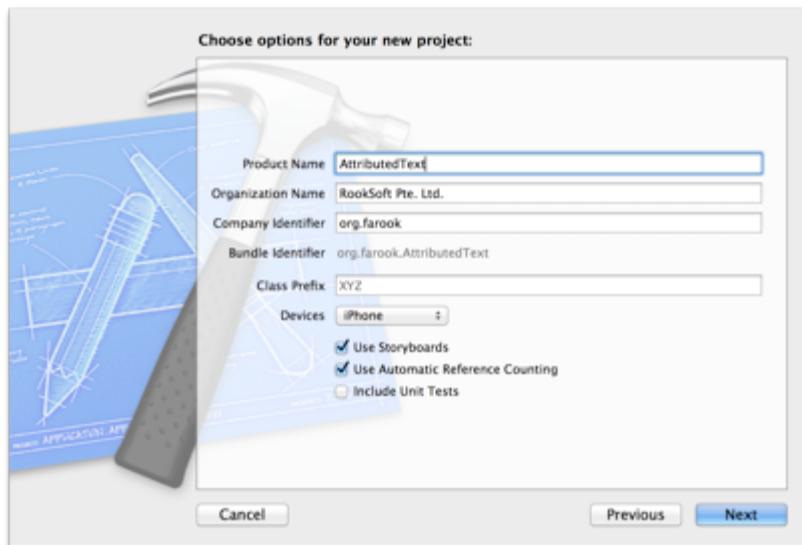


As you can see, `NSAttributedString` hold strings just like `NSString`, but it can also associate formatting information like fonts, colors, and more to parts of the string. Between the two, it's clear that `NSAttributedString` is the class to go with if you are looking to work with formatted text.

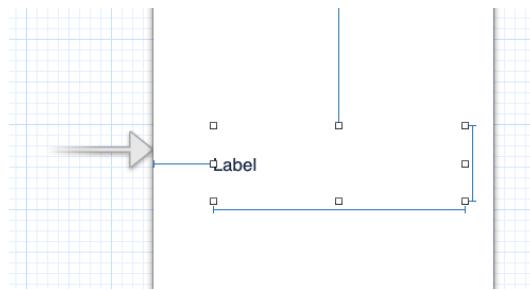
The only problem is prior to iOS 6 this class has been out of the spotlight because you could only use it with the Core Text framework. But now with iOS 6, `NSAttributedString` moves into prime time, as now most of the most critical UIKit controls make direct use of it!

Since using `NSAttributedString` in an iOS 6 project is incredibly easy, I'll skip the long introduction and just walk you through trying out different things in the iPhone Simulator. After all, the best way to learn is by trying it out for yourself. Let's go!

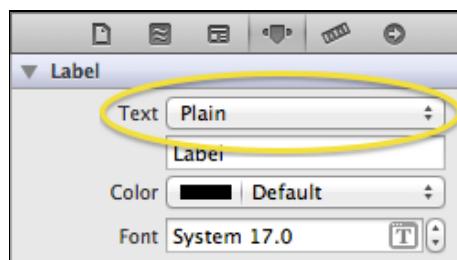
From Xcode's menu, choose **File/New/Project...** and create an iOS application based on the **Single View** template. Call the project **AttributedText**, ensure that Storyboards and ARC are enabled, that the Device is set to iPhone, and save it.



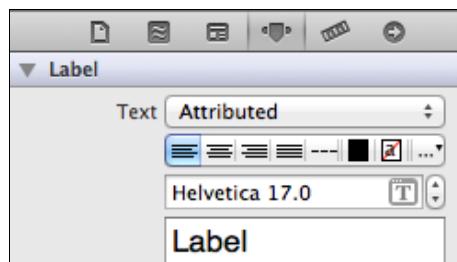
Click on **MainStoryboard.storyboard** in the project navigator and drag a label from the Object Library on the right onto your main view. Make it a bit bigger so that it can show 2-3 lines of text, like so:



Now look at the Attribute inspector on the right sidebar; there's a new property for the label component at the top of the palette (make sure the label is selected to see its properties):



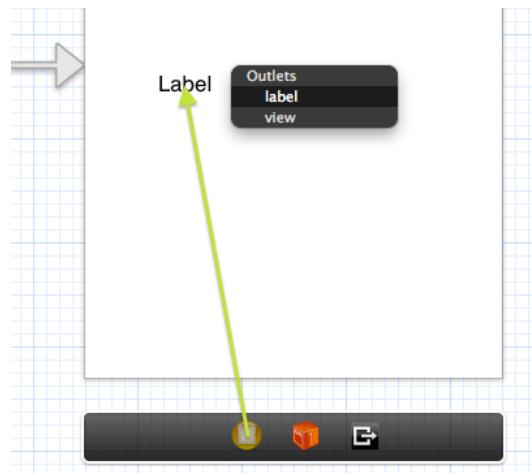
From the drop-down menu (highlighted in the picture above) select **Attributed**. The UI should change a bit and should now look like this:



OK, now you need to connect the label to your view controller class so you can start playing with attributed strings. Open **ViewController.m** and in the class extension (the `@interface ViewController()` section at the top) add this code:

```
{  
    IBOutlet UILabel* label;  
}
```

Switch back to **MainStoryboard.storyboard** and while holding the Ctrl key, drag-and-drop from the view controller icon on the bar below the view onto the label, as in the screenshot below:



From the menu that pops up, select **label**. Now the label is connected!

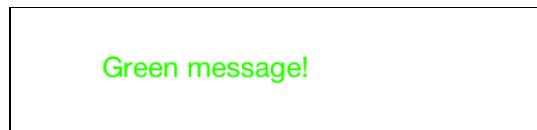
Open **ViewController.m** and add your first attributed strings UIKit code to the end of `viewDidLoad`:

```
// 1
NSDictionary* attributes = @{
    NSForegroundColorAttributeName: [UIColor greenColor]
};

// 2
NSAttributedString* aString =
    [[NSAttributedString alloc] initWithString:@"Green message!"
                                         attributes:attributes];

// 3
label.attributedText = aString;
```

I'll explain this in a moment, but for now click Run and give the project a try:



Congrats – you've made your first attributed string and displayed it in a label! Let's have a look at the code:

1. First you create a dictionary with one key-value pair. For the key you use the new UIKit constant `NSForegroundColorAttributeName` and you set the value to `greenColor` as provided by `UIColor`.
2. Then you create a new `NSAttributedString` and use its custom initializer, which takes in a text and a dictionary of attributes to apply to that text.
3. Finally you set the new property of `UILabel` – **attributedText**.

That's about it! You create a dictionary – the keys are the text style properties you want to set – and you supply the proper values. Then you apply this dictionary to

the attributed text object, and you're done! You've just applied a foreground color to the text of a `UILabel`.

"But wait - I have easily done before by just setting the label's color!", I am sure some of you are (correctly) protesting.

But my young Padawan, we're just getting started. ☺ Let's try applying some more interesting attributes. Replace the line declaring the `attributes` dictionary (section #1) with this new declaration:

```
NSDictionary* attributes = @{
    NSForegroundColorAttributeName: [UIColor greenColor],
    NSUnderlineStyleAttributeName: [NSNumber numberWithInt:
        NSUnderlineStyleSingle]
};
```

This time you add an additional property for underline – the number you pass in represents the width of the underline (1 in this case). As a reminder, you have to put an @ symbol before the 1 to convert it into an `NSNumber` with the new Objective-C literal syntax – see Chapter 2, “Programming in Modern Objective-C” for more information.

Build and run, and you'll see the following:



Green message!

Great! This is text formatting you didn't have at your fingertips in UIKit before iOS6. Let's try a few more style attributes just to see what is possible. Replace section #1 again with the following code:

```
NSDictionary* attributes = @{
    NSForegroundColorAttributeName: [UIColor blueColor],
    NSFontAttributeName:
        [UIFont fontWithName:@"Zapfino" size:24.0],
    NSBackgroundColorAttributeName: [UIColor orangeColor],
    NSKernAttributeName: @-3.0
};
```

I am sure you are getting the hang of it already. There are a few new style properties in this piece of code, so let's check them out:

- `NSFontAttributeName` sets the font of the text.
- `NSBackgroundColorAttributeName` sets the background color of the text. Note the difference compared to setting the background color of the entire label – this attribute will only set the background color for the part of the label that actually renders some text.
- `NSKernAttributeName` sets the horizontal offset between characters in the text.

If you run the project now, you should see this:



Yes, the message is a bit misleading now because it isn't green any more, but that's computers for you – they obediently follow your instructions. ☺

Some of you might not be familiar with text kerning (after all, it's something that designers use more often than developers do). Have a look at the example below and you will see the effect of different kerning values on the same text (positive and negative offset):



As you can see, kerning effects adjusts the spacing in-between characters. It is often most useful to apply fine-grained adjustments to subsets of characters to get a particular effect.

At this point you've seen how to set a few different style attributes of the characters in a given text, and that's already pretty awesome compared to what you were able to achieve in iOS 5.

But of course there are many more attributes! You already know how to apply character formatting to a piece of text, so here's a quick reference of all the styles you can use, which you can refer to for your own projects:

| | |
|--------------------------------|--|
| NSBackgroundColorAttributeName | (UIColor*) Sets the text background color. |
|--------------------------------|--|

| | |
|-----------------------------------|--|
| NSFontAttributeName | <p>(UIFont*) Sets the font to render the text. If you want bold or italic text, provide the correct name for each given font. These vary depending on the font family.</p> <p>For example, for the Helvetica Neue font, the bolded version is HelveticaNeue-Bold, and the italicized version is HelveticaNeue-Italic.</p> <p>But if you would like to use Courier New, the font names are CourierNewPS-ItalicMT for italic and CourierNewPS-BoldMT for bold.</p> <p>To see all the fonts available in iOS 6, you can consult: http://www.ios6-fonts.com.</p> |
| NSForegroundColorAttributeName | (UIColor*) Sets the color to use when rendering the text. |
| NSKernAttributeName | (NSNumber*) Sets the floating point value to adjust the space between the characters in the text. |
| NSLigatureAttributeName | (NSNumber*) If you provide the value @1, this enables the use of ligatures when rendering the font. If you would like to learn more about typographic ligatures, read up here: http://en.wikipedia.org/wiki/Typographic_ligature . |
| NSParagraphStyleAttributeName | (NSParagraphStyle*) For this key you provide a new object that can set a number of styling properties for the whole paragraph (vs. setting the style of the text characters). More about this class later on. |
| NSShadowAttributeName | <p>(NSShadow*) For this key you provide another class new to iOS 6, NSShadow.</p> <p>You can set the color, blur radius and the offset of the shadow. A quick demo is coming in a moment!</p> |
| NSStrikethroughStyleAttributeName | (NSNumber*) Enable or disable the text strikethrough style. Use one of the two pre-defined constants: NSUnderlineStyleNone or NSUnderlineStyleSingle. Look at the example after this table for example usage. |
| NSStrokeColorAttributeName | (UIColor*) Sets the color to use for the |

| | |
|-------------------------------|--|
| | <p>character stroke. In case this isn't clear, have a look at the example below showing foreground color vs. stroke color:</p>  |
| NSStrokeWidthAttributeName | <p>(NSNumber*) Sets the width to use for the stroke. You can also use floating numbers like 1.5, 2.75, etc.</p> |
| NSUnderlineStyleAttributeName | <p>(NSNumber*) Enable or disable the text underlining style. Use one of the two pre-defined constants: NSUnderlineStyleNone or NSUnderlineStyleSingle. Look at the next example below for the proper usage.</p> <p><i>Off-record:</i> You can also use values like @2,@3, etc to set the actual thickness of the line, or even @10 to set a double underlining. However these are undocumented features, which might be coming in officially in a coming minor update to iOS6.</p> |

Let's have a quick look at the shadow definition class before moving on to paragraphs and even more formatting. You've probably already used the shadowing properties on `UILabel`, so the new `NSShadow` class should feel familiar.

Go back to `ViewController.m` and again replace section #1 with this code:

```

NSShadow* myShadow = [[NSShadow alloc] init];
myShadow.shadowColor = [UIColor grayColor];
myShadow.shadowBlurRadius = 5.0;
myShadow.shadowOffset = CGSizeMake(1,1);

NSDictionary* attributes = @{
    NSUnderlineStyleAttributeName: [NSNumber numberWithInt:
        NSUnderlineStyleSingle],
    NSStrikethroughStyleAttributeName: [NSNumber numberWithInt:
        NSUnderlineStyleSingle],
    NSForegroundColorAttributeName: [UIColor blueColor],
    NSShadowAttributeName: myShadow
};

```

First you declare a new instance of `NSShadow`. You set the shadow color, the blur radius to apply to the shadow, and finally, the offset from the text as a `CGSize`. Then, as before, you set a few different style properties discussed in the table above.

Build and run to have a look at the result:



Green Message!

Maybe that's slightly odd looking, but also impressive tech-wise! ☺ Time to turn your attention to styling paragraphs.

Paragraphs also come in style this year!

Take a look at this label, which contains an entire paragraph of text:

(UIFont*) Sets the font to render the text. If you want bold or italic text provide the correct name for each given font. These vary depending on the font family. For example for the "Helvetica Neue" font you need to provide "HelveticaNeue-Bold" name for a bolded font, and "HelveticaNeue-Italic" for italic font. However, if you would like to use "Courier New", the font names are: "Cou..."

You can see that some text styles are already applied to the text – like color, shadow and font. However, there are other aspects of the paragraph that you might want to tweak to improve readability and the general look and feel. These include the spacing between the lines in the paragraph, the line height, and the offset from the text bounding box.

Let's get cracking. Switch to **MainStoryboard.storyboard** in your Xcode project and resize the label so that it covers most of the screen. Also set the **Lines** property of the label to **0**, so that the text will display across multiple lines.

Now go back to **ViewController.m** and replace sections #1-3 in `viewDidLoad` with this new code:

```
NSShadow* myShadow = [[NSShadow alloc] init];
myShadow.shadowBlurRadius = 2.0;
myShadow.shadowColor = [UIColor grayColor];
myShadow.shadowOffset = CGSizeMake(1,1);
```

```

NSDictionary* attributes = @{
    NSForegroundColorAttributeName: [UIColor colorWithRed:0.2
green:0.239 blue:0.451 alpha:1] /*#333d73*/,
    NSShadowAttributeName: myShadow
};

NSString* txt = @"(UIFont*) Sets the font to render the text.
If you want bold or italic text provide the correct name for each
given font. These vary depending on the font family.\nFor example
for the \"Helvetica Neue\" font you need to provide
\"HelveticaNeue-Bold\" name for a bolded font, and
\"HelveticaNeue-Italic\" for italic font.\nHowever, if you would
like to use \"Courier New\", the font names are: \"CourierNewPS-
ItalicMT\" for italic and \"CourierNewPS-BoldMT\" for bold.";

NSAttributedutedString* aString = [[NSAttributedutedString alloc]
initWithString: txt attributes: attributes];

label.attributedText = aString;

```

If you hit Run now, you will see the text formatted as in the previous image.

Now to the paragraph styling. Add the following right below the `[super viewDidLoad]` in `viewDidLoad`:

```

NSMutableParagraphStyle* paragraph =
[[NSMutableParagraphStyle alloc] init];
paragraph.alignment = NSTextAlignmentJustified;

```

This code creates a new `NSMutableParagraphStyle` instance and sets the text alignment to justified. A justified alignment means the layout engine will stretch the space between the words so that the text takes up the whole width of the lines in the text paragraph.

Now add the following below the above bit of code:

```

paragraph.firstLineHeadIndent = 20.0;
paragraph.paragraphSpacingBefore = 16.0;

```

The first property sets the offset of the first word in the paragraph from the beginning of the line, and the second property sets the offset of the first line of the paragraph from the previous line of text (see the image below).

Now insert a new key into the `attributes` dictionary that's already in the code:

```

NSParagraphStyleAttributeName: paragraph

```

This will apply the paragraph style you defined above to the label's text. Hit Run and have a look at the result. Wow!

The diagram shows a block of attributed text with several annotations:

- A red double-headed arrow at the top indicates the text is justified to fill up the line width.
- An annotation above the first line reads: "(UIFont*) Sets the font to render the text. If you want bold or italic text provide the correct name for each given font. These vary depending on the font family." with a red underline under "render the text".
- An annotation below the first line reads: "First word is offset by 16pt paragraph margin" with a red underline under "paragraph margin".
- An annotation below the first line reads: "For example for the "Helvetica Neue" font you need to provide "HelveticaNeue-Bold" name for a" with a red underline under "Helvetica Neue".

The text is already looking good, but it can be refined even further.

Look at that first line – to be honest, there's a little too much spacing between some of the words. That makes reading uncomfortable, because the eye needs to jump ahead to find the next word. What's more, I personally would enjoy a bit more air between the lines.

Find the line in your code where you set the `paragraphSpacingBefore` property, and paste this code directly under it:

```
paragraph.lineSpacing = 4;
paragraph.hyphenationFactor = 1.0;
```

The `lineSpacing` style property will add extra four pixels between lines (you can also use negative values to bring lines closer together).

The `hyphenationFactor` instructs the layout engine about the paragraph's hyphenation threshold. This factor's value can vary between 0.0 and 1.0; values closer to 0 will make the engine do less hyphenation, while values closer to 1 will make it do more. A factor of 1 will results in as much hyphenation as possible.

Hit Run and note the difference:

The diagram shows the same attributed text as before, but with different styling. The `lineSpacing` property has been set to 4px, creating more space between lines. The `hyphenationFactor` has been set to 1.0, resulting in extensive hyphenation throughout the text.

I am sure you are already pretty convinced that UIKit is now a super-powerful text-formatting engine! Let's quickly go over all the style properties you can set for paragraphs:

| | |
|-----------|--------------------------------|
| alignment | (NSTextAlignment) An NSInteger |
|-----------|--------------------------------|

| | |
|----------------------|---|
| | that accepts one of these pre-defined constants: NSTextAlignmentLeft, NSTextAlignmentCenter, NSTextAlignmentRight, NSTextAlignmentJustified or NSTextAlignmentNatural |
| baseWritingDirection | (NSWritingDirection) An NSInteger, that takes in: NSWritingDirectionLeftToRight, NSWritingDirectionRightToLeft, or NSWritingDirectionNatural. If you specify NSWritingDirectionNatural, then the system determines which direction to use. |
| firstLineHeadIndent | (CGFloat) The offset for the first word of the paragraph. |
| headIndent | (CGFloat) The first word's indent on any line other than the first one in the paragraph. |
| hyphenationFactor | (CGFloat) A value between 0.0 and 1.0, with 0.0 being no hyphenation at all, and 1.0 being hyphenation in all cases, where possible. |
| lineBreakMode | (NSLineBreakMode) One of the following: NSLineBreakByWordWrapping, NSLineBreakByCharWrapping, NSLineBreakByClipping, NSLineBreakByTruncatingHead, NSLineBreakByTruncatingTail, NSLineBreakByTruncatingMiddle. |
| lineHeightMultiple | (CGFloat) A line height multiplier. If you set it to 1.5 for example, then the height of the text lines will be 50% bigger than the natural line height for the given font size. |
| lineSpacing | (CGFloat) The margin between |

| | |
|------------------------|--|
| | the lines in the text. |
| maximumLineHeight | (CGFloat) The maximum line height. |
| minimumLineHeight | (CGFloat) The minimum line height. |
| paragraphSpacing | (CGFloat) The paragraph's bottom margin space. |
| paragraphSpacingBefore | (CGFloat) The paragraph's top margin space. |

Cool! Your iOS designer is going to be really happy about these!

Now you've mastered almost everything you can do to an attributed string. But isn't it boring when the *whole* string is formatted the same way?

Mutating text formatting

In this section of the chapter you'll learn how to mutate your `NSAttributedString`s. But be careful, or you might end up with extra eyeballs! 😊

So far, `NSAttributedString` has been great, but once you create it, you can't change it. In addition, you've only been able to apply attributes to the entire string – not individual parts of a string.

But never fear... the `NSMutableAttributedString` comes to the rescue!

You can still use all the formatting knowledge you gained using the non-mutable attributed strings, but you do need to learn how to apply different formatting styles to different parts of the text.

Delete all the code from `viewDidLoad` (except for the call to `super`, of course) and paste in this to define two different styles:

```
NSDictionary* redAttrs = @{
    NSForegroundColorAttributeName: [UIColor redColor],
};

NSDictionary* greenAttrs = @{
    NSForegroundColorAttributeName: [UIColor greenColor],
};
```

Now you want to create a mutable attributed string and apply the `redAttrs` and `greenAttrs` styles to different parts of the text. Go on and add this code:

```
NSMutableAttributedString* aString =  
[ [NSMutableAttributedString alloc] initWithString:  
 @"Red AND green text!" ];  
  
[aString setAttributes:redAttrs range:NSMakeRange(0,3)];  
[aString setAttributes:greenAttrs range:NSMakeRange(8,5)];  
  
label.attributedText = aString;
```

Aha! First you declare the new object `aString` and load it up with the text "Red AND green text!" Then you call `setAttributes:range:` on the object and apply the attribute sets to different ranges of the text. Pretty easy, eh?

Red AND green text!

Now you can build just about any formatting you want – combine styles, overlap formatting, overwrite attributes – it's up to your imagination (and your naturally sophisticated sense of style, of course)!

There are certainly more interesting APIs of `NSMutableAttributedString` to look into. And you are going to do so, but to keep things lively and showcase some of the real-world applications of this material, you are going to learn on the fly while developing this chapter's iOS 6 project.



A Notes app on steroids

For the rest of this chapter, you will create a simple clone of the default Notes application that comes with every iOS device.

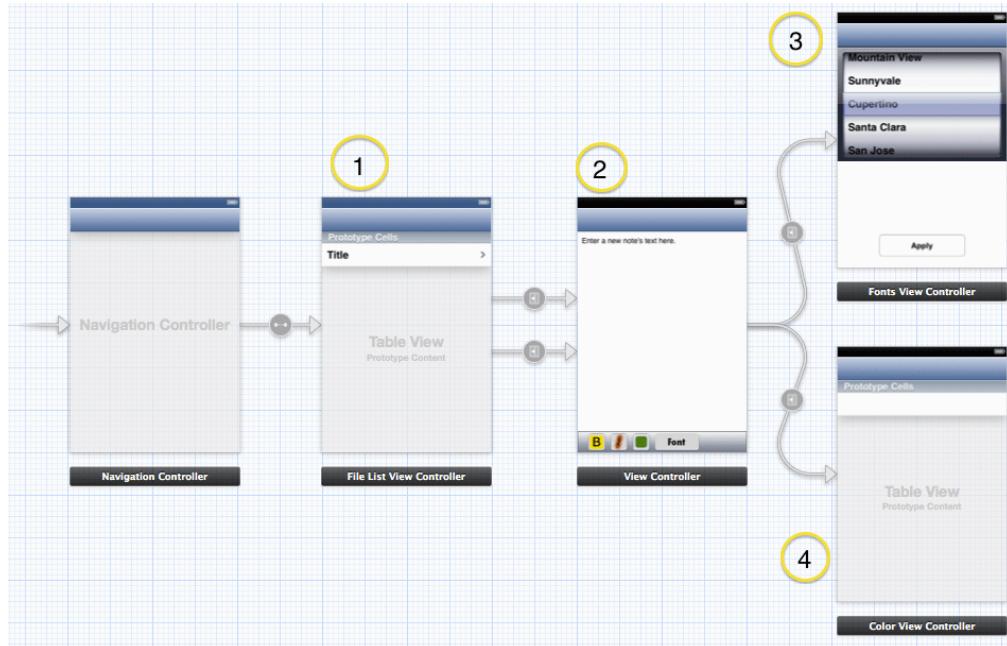
But your project is going to be even more awesome than the default Notes app – it's going to be powered by the new iOS 6 attributed strings functionality. It will be like the Notes app on steroids – yeah!

The finished project will look like this:



I've prepared a starter Xcode project for you with all the screens of the app already connected in a storyboard, and also hooked up to the necessary empty view controller classes. You should find the file among the resources for this chapter.

Open the **RichEditor.xcodeproj** file and then select **MainStoryboard.storyboard** to have a look at the project's storyboard:



And here's the plan for each of the screens of the app:

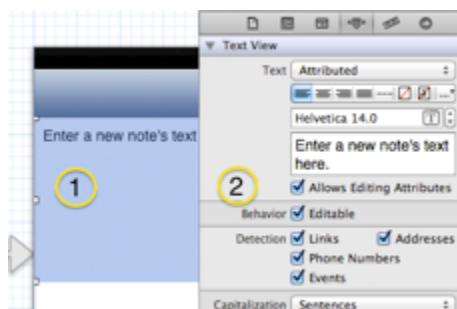
1. When opened, the app will show the list of already created notes; tapping on a note will open it up in the editor. There's also an Add button already in place for creating a new note.

2. The editor screen features a rich text editing text area, and text keyboard with a custom toolbar for applying styles to the text. In order from left to right, there are buttons to highlight text, apply a flame-like effect to text, choose a font, and change the text color. The view controller has code to make the toolbar scroll into screen when the keyboard appears, resting right above the keyboard.
3. When the user taps the Font button on the editor screen, they are taken to a separate screen, where they can choose from a list of pre-defined fonts and text sizes.
4. When the user taps the color selection button on the editor screen, a new screen pops up with a pre-defined list of colors to choose from.

You can actually hit Run in Xcode right now and have a look at the project. All the screens are connected, but no logic is implemented.

Built-in text view rich formatting

While the storyboard file is open, zoom in on the editor screen ("View Controller") and select the text area, which takes up the upper half of the screen. Then in the Attributes inspector on the right sidebar, check the box for **Allows Editing Attributes**.



Believe it or not – you just converted the startup plain-text editor project into a glorious rich-text editor project!



Let's have a look at the results!

Run the project and tap the (+) button to open the editor screen. Select the pre-filled text and have a look at the context menu – there's one new item as compared to before:



Tap the “**B / U**” menu item and a new submenu pops up, which you can use to format the text! Awesome! You’ve made it!



The context menu with the bold, italics, and underline option is the new rich-text formatting capability in `UITextField`, new to iOS 6. Not bad for just clicking a couple buttons. But of course you aim to do more, so let’s get cracking!

Warning: Unfortunately the final 4.5 version of Xcode was released with a bug causing the super handy “**Allows editing attributes**” checkbox to not actually work. Before the time Apple releases an update to Xcode you will have to use code instead of just ticking this checkbox. Just do the following:

- 1) Open up **ViewController.m** and find `viewDidLoad` method.
- 2) Paste in the following code, which does exactly the same as ticking the checkbox in Interface Builder:

```
editor.allowsEditingTextAttributes = YES;
```

Voila!

Custom rich text editing

As a reminder, the toolbar that appears on top of the keyboard with the yellow highlight, flame-like effect, and other buttons is completely custom – I added it into

the starter project for you. However, the buttons don't do anything yet – that's for you to implement now!

Let's start with the first button, which should apply a bright yellow background to the selected text.

Open **ViewController.m** and find the stub for the `btnBTapped:` method.

The B button (below) is already connected to the `btnBTapped:` action for you in the storyboard.



You will make it so the highlight button only works when there's a piece of text already selected, so you can apply the highlight to that text. Start by adding the code to check whether there's selected text:

```
NSRange selection = editor.selectedRange;

if (selection.length==0) {
    [[[UIAlertView alloc] initWithTitle:@"Warning"
                                message:@"Select text to highlight first"
                                delegate:nil
                                cancelButtonTitle:@"Close"
                                otherButtonTitles: nil] show];
    return;
}
```

`editor` is an outlet connected to the text area, so you just fetch its `selectedRange`, and store it in `selection`. Then if the selection `length` equals zero, you just fire up an alert and bail out.

Next you need to check the existing background color of the selected text, so you can decide whether to highlight the selected text or to clear its background color. In other words, you want the button to allow the user to switch the highlighting on and off. Add this to the end of the method:

```
UIColor* bgColor =
[editor.attributedText attribute:NSBackgroundColorAttributeName
                        atIndex:selection.location
                      effectiveRange:nil];

UIColor* newColor;
if (CGColorGetAlpha(bgColor.CGColor)==0.0) {
    newColor = [UIColor yellowColor];
} else {
    newColor = [UIColor clearColor];
}
```

```
NSDictionary* bStyle = @{
    NSBackgroundColorAttributeName:newColor
};

[self applyAttributesToTextArea:bStyle];
```

`NSAttributedString` has a method called `attributeAtIndex:effectiveRange:` that allows you to read only the styling attribute you are interested for a certain position in the text. This is pretty handy, as you have the position of the text selection, and you are interested in whether or not it is already highlighted.

On the first line, you store the current background color of the selection into `bgColor`. Then you declare a new variable `newColor`. You use `CGColorGetAlpha()` to read the alpha channel of the current background color, and if it has its alpha set to 0.0 (i.e., it's a fully transparent color), you set `newColor` to yellow, and otherwise to a clear color.

You build a dictionary with the new background color styling.

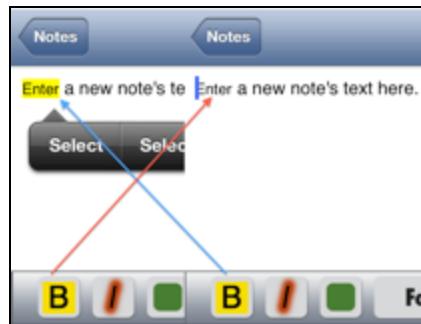
Finally, you call the `applyAttributesToTextArea:` method to apply the formatting – which is the next step!

Let's implement it in the most straightforward way – get the selection range, apply the attributes, and update the `attributedString` of the `UITextView` control. Add the following method:

```
-(void)applyAttributesToTextArea:(NSDictionary*)attrs
{
    NSRange selection = editor.selectedRange;
    NSMutableAttributedString* text =
        [[[NSMutableAttributedString alloc] initWithAttributedString:
          editor.attributedText];

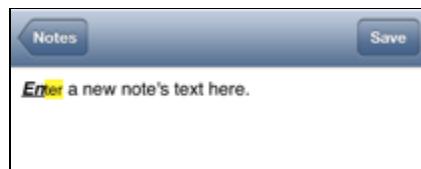
    [text setAttributes:attrs range:selection];
    editor.attributedText = text;
}
```

Give the project another build and run. You'll see that you can now select a word of text and highlight it by tapping on the highlight button. Then if you select the same word and tap the button again, its background color is cleared. Great!



That was almost too easy to believe, right?

Let's give the highlight function another try. This time, select the first word, and from the "BIU" menu apply bold, italic and underline styles. Now drag the left selector pin so that only a few characters (for example "ter") are selected, and press the highlight button.



OK, lesson learned – `setAttributes:range:` overwrites all the formatting with the new text style you provide. The method is really handy when you want to explicitly set the formatting on a few words in the text, but when you are editing you need another method.

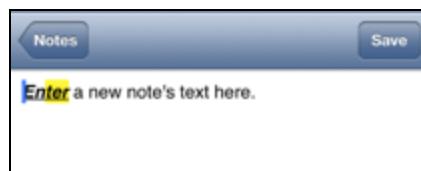
The method to add or replace a style on a given range of text is called `addAttributes:range:`, so go ahead and replace this line in `applyAttributesToTextArea::`:

```
[text setAttributes:attrs range:selection];
```

With the new API, call:

```
[text addAttributes:attrs range:selection];
```

Re-launch the project and try the same steps as before. Now the button should highlight the selected text without erasing all the previously-applied formatting, like so:



There is, however, still one thing that doesn't feel right. After you apply the highlight, the selection disappears and the cursor jumps to the beginning of the

text. Well, what if you're not done editing your selection? It's irritating to have to select it again.

The fix is really easy to implement. All you need to do is save the existing selection range before applying the new formatting, and re-select the same piece of text.

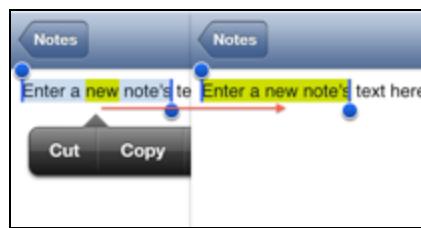
To do that, just add this one line at the end of the method:

```
editor.selectedRange = selection;
```

Now your selection persists while you change the formatting of the text in the editor!

Implementing context-dependent formatting

The highlight function in your editor is already kicking serious ass, but there's one thing that can be better. If you highlight a given word and then select more text that includes both highlighted and non-highlighted text, then tapping the button will "merge" the style of the selected words:



In other words, the current code either highlights or un-highlights the entire section as a group rather than toggling each piece of the text's current state.

This is OK for the highlight function, but for other formatting tools in your rich text editor, you might desire a different behavior.

For example, when you implement the next button in the formatting toolbar , you are going to make it look at the state of each part of the selection you have highlighted and toggle its current state accordingly. For example, here's what would happen if you selected all of the text on the left side and tapped the new button:



As you can see, this is different than the highlight button currently works – rather than modifying the text as a group, it treats each piece as a separate formatting area to toggle.

This button will use `nsshadow*` to apply a red shine to the selected text, giving it a nice flame-like effect. You will also learn several new tricks in text formatting, so everyone wins!

In **ViewController.m**, find the `btnITapped:` stub and add the following code to get started:

```
NSRange selection = editor.selectedRange;

if (selection.length>0) {

    //1
    NSMutableAttributedString* a =
        [[NSMutableAttributedString alloc]
         initWithAttributedString:editor.attributedText];

    //2
    [editor.attributedText
     enumerateAttributesInRange:selection
     options:kNilOptions
     usingBlock:^(NSDictionary *rangeAttrs, NSRange range,
                 BOOL *stop) {
        //apply formatting
    }];
}

//3
editor.attributedText = a;
editor.selectedRange = selection;

}
```

Let's look at the method's initial code. Just as before, you grab the current text selection and store it in the `selection` variable. If there's text selected, you go on to processing the selection, performing these three steps:

1. You copy the contents of the text editor into a mutable attributed string so you can modify the existing formatting.
2. `enumerateAttributesInRange:options:usingBlock:` loops over all pieces of the text that have continuous formatting, and invokes the supplied block for each of the formatting "runs".
3. Finally you update the editor text and restore the selection.

Let's have a look at what a "run" is in the text. Here's an example with two different types of formatting applied:



In the above text, there are three runs:

1. Blue foreground color.
2. Blue foreground color and underline.
3. Underlined text.

As you can see, there's a difference between applying text formatting and reading the formatting back. This attributed text was built by applying formatting to two overlapping ranges – the blue foreground color was applied to "Text with" and the underlining was applied to "with runs." But when you read the formatting, you get back three runs, each with a unique combination of formatting attributes applied.

Next let's implement the body of the block. Replace the //apply formatting comment with:

```
NSShadow* currentShadow = rangeAttrs[NSShadowAttributeName];
NSShadow* newShadow = [[NSShadow alloc] init];

if (!currentShadow || currentShadow.shadowBlurRadius==0.0) {
    newShadow.shadowColor = [UIColor redColor];
    newShadow.shadowBlurRadius = 6.0;
} else {
    newShadow.shadowColor = [UIColor clearColor];
    newShadow.shadowBlurRadius = 0;
}

[a addAttribute:NSShadowAttributeName
           value:newShadow
           range:range];
```

This is similar to what you've done already for the highlight button.

Your block receives the current text run attributes via the `rangeAttrs` dictionary parameter, so you fetch the existing shadow from it. Then you create a new blank shadow object – `newShadow`.

Then you check if there is an existing shadow applied to the text. If there is, you just create a clear shadow, which won't be visible; else you create a red shadow with a 6-point blur applied to it.

In the end, you apply the shadow using the same method as for the highlight.

That's it! This method will loop over the different parts of the text and switch the red glow effect on and off. Give it a try!



OK! Your rich text editor is getting more and more powerful! And in the meantime, you are learning a lot about working with attributed strings in iOS 6.

One little annoyance is that you always have to first select the text, and then click the button to make it glow in red. Wouldn't it be cool to be able to just turn the glow on and off as you type? Yes – it would!

This would involve modifying the text formatting as the user types, i.e., setting a kind of "pending" formatting, which gets applied to the next character the user types.

After the closing curly bracket of the `if` condition you just added, add the `else` part – to be executed if there's no text selection:

```
else {
    //1
    NSMutableDictionary* pendingAttrs =
        [[NSMutableDictionary alloc] initWithDictionary:
            editor.typingAttributes];

    //2
    NSShadow* currentShadow = pendingAttrs[NSShadowAttributeName];
    NSShadow* newShadow = [[NSShadow alloc] init];

    if (!currentShadow || currentShadow.shadowBlurRadius==0.0) {
        newShadow.shadowColor = [UIColor redColor];
        newShadow.shadowBlurRadius = 6.0;
    } else {
        newShadow.shadowColor = [UIColor clearColor];
        newShadow.shadowBlurRadius = 0;
    }

    //3
    [pendingAttrs setObject:newShadow
        forKey:NSShadowAttributeName];

    //4
    editor.typingAttributes = pendingAttrs;
}
```

Let's see what the code does:

1. You fetch the current typing attributes from the `UITextView` and store them in `pendingAttributes`. Aha! The new-in-iOS 6 `typingAttributes` property contains the formatting to be applied to newly-typed characters!
2. Next you have exactly the same block of code as before to determine whether the red glow has already been applied to the text, and build the proper shadow object to replace the existing one.
3. You just replace the shadow component in the current text format.
4. Finally, you assign the new formatting to the `typingAttributes` of the text area.

Believe it or not – that's all!

Run the project and type in a word. Then tap the "/" button (below) and type in another word. Then tap the button again and type some more. As you can see, it keeps track of your editing state appropriately.



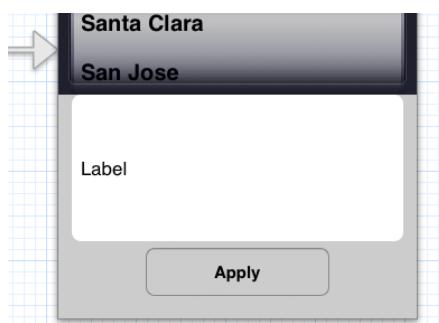
Congrats - now you know how apply custom formatting to entire selections of text or individual formatting runs, and how to edit the pending formatting for a `UITextView` component. But there's more - it's time to have a look at yet another cool feature available in iOS 6 and XCode 4.5!

Formatting text in Interface Builder

So far you know how to build attributed strings programatically and how to show them using labels. What about cases where you have static text that you want to lay out in your storyboard using Interface Builder, and you never want to change it again? You definitely don't want to use code to set that up, right?

If you run the project now and from the editor screen tap on the `Font` button, you are taken to the font selection screen. Right now not much is happening on this screen, but it's going to serve as a playground for your next endeavor.

Switch to the storyboard, select the Fonts View Controller, and add a label in between the picker view and the button, like so:

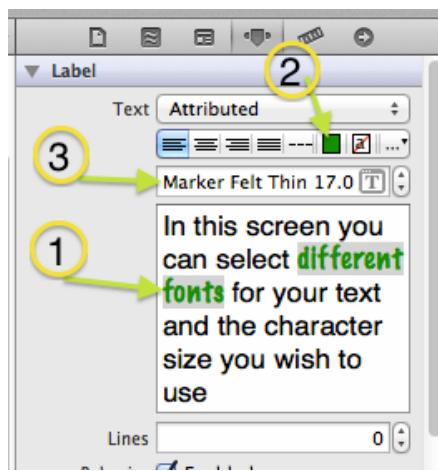


Set **0** for the **Lines** property and in the **Text** field, type, "In this screen you can select different fonts for your text and the character size you wish to use."

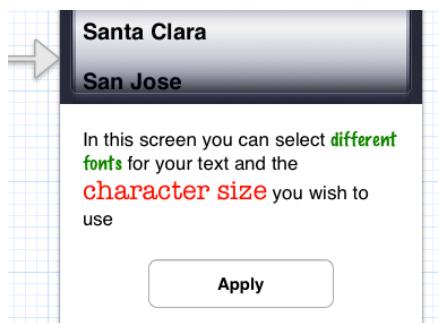
You should see the plain text showing up in the user interface design area.

Now choose **Attributed** from the drop-down box above the Text field (where you just pasted the text). This in the Text field, do the following:

1. Select the text "different fonts".
2. Choose a green color from the color selector (look at the picture below for its location).
3. Choose the Marker Felt font from the font selector. (I know, the designers among you are cringing now!) ☺



As you do each of these things, you will see the formatting being applied to the label preview. Select the words "character size" and apply a different color and font size to them as well. You should have the label looking something like this:



That wasn't hard at all! You can, in exactly the same way, also set attributed text for a text field's content, a button's title, and so on.

Now let's tackle implementing the font selector screen of your rich text editor!

Fonts, fonts and more fonts

First let's pre-fill the picker control with different font names and sizes for the user to choose.

Open **FontsViewController.m** and find the class extension at the top. You will need a new class variable to store the list of fonts. On the line below the second IBOutlet declaration, add:

```
NSArray* fontsDataSource;
```

Then in the `awakeFromNib` method (called before the view is loaded), add:

```
fontsDataSource = @[
    @[@12, @14, @18, @24],
    @{@"Arial":@"AmericanTypewriter", @"Helvetica",@"Zapfino"}
];
```

That's your fonts database for the font selector.

Now you need to implement a few delegate methods for the picker control. Find the `numberOfComponentsInPickerView:` method placeholder. Replace the code inside with the real code:

```
return fontsDataSource.count;
```

This tells the picker control to show two picker columns to the user.

Now you need to let the picker control know how many rows (or options) there should be in each column. Find the `pickerView:numberOfRowsInComponent:` method and replace the placeholder code with:

```
return ((NSArray*)fontsDataSource[component]).count;
```

Finally, you need to implement a new method that will provide the values from `fontsDataSource` to the picker control. Add this code:

```
- (NSAttributedString *)pickerView:(UIPickerView *)pickerView
attributedTitleForRow:(NSInteger)row
forComponent:(NSInteger)component
{
    //1
    id data = ((NSArray*)fontsDataSource[component])[row];

    //2
    NSMutableAttributedString* title =
        [[NSMutableAttributedString alloc] initWithString:
            [data description]];
}
```

```
if (component==0) {
    //3
    float pointSize = [(NSNumber*)data floatValue];
    NSDictionary* attr = @{@"NSFontAttributeName" :
        [UIFont fontWithName:@"Arial" size:pointSize] };

    [title setAttributes:attr
        range:NSMakeRange(0, title.length)];
}

if (component==1) {
    //4
    NSDictionary* attr = @{@"NSFontAttributeName" :
        [UIFont fontWithName:data size:16.0] };
    [title setAttributes:attr
        range:NSMakeRange(0, title.length)];
}

//5
return title;
}
```

The picker control calls this method for each of the options, on each column, and asks for the title it should show for that option. Let's go over the code:

1. First you fetch the correct piece of data from `fontsDataSource` – you use the `component` and `row` values provided to the method to fetch either a font name or a size.
2. Then you create a new attributed string, and you feed it the value of the grabbed piece of data from `fontsDataSource`.
3. If you are sending data to the first column (i.e., `component` equals zero), then you cast the fetched data to an `NSNumber` and get the float value. You create a new formatting style called `attr`, and you set the font as Arial and the font size as whatever you fetched in `pointSize`. Finally, you apply it to the `title` attributed string.
4. Similarly, for the second column, you fetch the font name from `fontsDataSource` then you generate a new formatting with that font and apply it to `title`.
5. In the end, you just return `title`. UIKit will take care of rendering the attributed string on the picker control.

Hit Run and press the font button in the editor. The picker control comes to life before your eyes!



How cool is that? ☺ Very! But even this is not good enough!

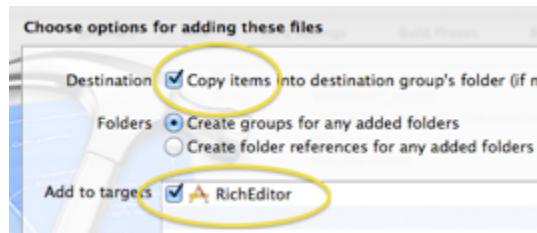
Integrating custom fonts

How about using your own custom fonts to render attributed strings?

To try this out, download the **Comfortaa** font from [fontsquirrel.com](http://www.fontsquirrel.com/fonts/Comfortaa), an awesome resource for free fonts: <http://www.fontsquirrel.com/fonts/Comfortaa>. Find and click the button circled below to download the font:



Extract the zip file and drag the **Comfortaa-Regular.ttf** file into your Xcode project file list (make sure you check the project target, so that Xcode includes the file with your app bundle):



Then select the Xcode project root in the project navigator, select the **Info** tab on the right and find the key called **Fonts provided by the application**. Expand it and enter **Comfortaa-Regular.ttf** in the first array item field:

| Summary | Info | Build Settings | Build Phases | Build Rules |
|---|---------|---|--------------|-------------|
| Custom iOS Target Properties | | | | |
| Key | Type | Value | | |
| Bundle versions string, short | String | 1.0 | | |
| Bundle identifier | String | <code>\$(PRODUCT_NAME)identifier</code> | | |
| InfoDictionary version | String | 6.0 | | |
| Main storyboard file base name | String | MainStoryboard | | |
| Application Category | String | | | |
| Bundle version | String | 1.0 | | |
| Executable file | String | <code>\$(EXECUTABLE_NAME)</code> | | |
| FONTs provided by application | | | | |
| Item 0 | String | Comfortaa-Regular.ttf | | |
| Application requires iPhone enviro | Boolean | YES | | |
| Supported interface orientations | | | | |
| | Array | (3 items) | | |

Now the Comfortaa font is included in your app. Let's add it to the font picker!

Open **FontsViewController.m** and find the line where you defined the list of available fonts. Just add the font name at the end of the list, like so:

```
@[@"Arial",@"AmericanTypewriter",
 @"Helvetica",@"Zapfino",@"Comfortaa"]
```

Run the app and open the font selector screen:



As you can see, you can use any custom font of your choosing with attributed strings if you add it to your app in this manner. Awesome!

Applying a font to the string

To complete the font selector, you need to send the current font selection to the **FontsViewController**, and then receive the new selection from it when the view is closed.

Open **FontsViewController.h** and replace it with this new code:

```
@protocol FontsViewControllerDelegate <NSObject>
-(void)selectedFontName:(NSString*)fontName
withSize:(NSNumber*)fontSize;
@end

@interface FontsViewController : UIViewController
```

```
@property (weak) UIViewController<FontsViewControllerDelegate>*
delegate;
@property (strong) UIFont* preselectedFont;
@end
```

Here you declare two new properties for the Fonts View Controller:

1. delegate – This is the class that will be notified when the user selects a new font. In this case, you will set this to the `viewController` class.
2. `preselectedFont` – This property will be set from `viewController` before the font selection screen pops up.

To be a good team player (or more precisely, “team programmer”), you declare the protocol for the class delegate – `FontsViewControllerDelegate`. When the user selects a new font, the delegate will receive the font name and size, so it can apply them to the text area.

Now open **FontsViewController.m** and find the `viewDidAppear:` stub. Add this code to it:

```
[super viewDidAppear:animated];

NSArray* fontSizes = fontsDataSource[0];

for (int i=0;i<fontSizes.count;i++) {
    NSNumber* size = fontSizes[i];
    if ([size floatValue]==self.preselectedFont.pointSize) {
        [fontPicker selectRow:i inComponent:0 animated:YES];
        break;
    }
}

NSArray* fontNames = fontsDataSource[1];

for (int i=0;i<fontNames.count;i++) {
    NSString* name = fontNames[i];
    if ([name compare:self.preselectedFont.fontName] ==
        NSOrderedSame) {
        [fontPicker selectRow:i inComponent:1 animated:YES];
        break;
    }
}
```

First you grab the list of available font sizes and store it in `fontSizes`. Then you loop over that array and compare each of them to the size of the `preselectedFont` property. (You will set that property from the `viewController` class). When a match is found, you select that option in the picker.

Then you do the same for the list of font names, but this time you select the row from the second column of the picker control.

OK, now let's implement the code in `ViewController` to pass the current font to the `FONTsViewController`.

Switch to **ViewController.m**, find the empty `prepareForSegue:sender:` method and add the following code to it:

```
if ([segue.identifier compare:@"fonts" == NSOrderedSame]) {  
    FontsViewController* screen = segue.destinationViewController;  
    screen.delegate = self;  
    screen.preselectedFont =  
        editor.typingAttributes[NSFontAttributeName];  
    return;  
}
```

You do several things in this piece of code when the segue to the fonts selection screen is invoked. First you set the `delegate` of the new screen to be your `ViewController` class. Then in one pass, you fetch the current font from the `UITextView` (via `typingAttributes`) and set it as the value of the `preselectedFont` property of `FONTsViewController`.

One final touch and you are ready. Scroll to the top of the file and add the protocol you just declared a few moments ago to the class extension. Your new `@interface` line should look like this:

```
@interface ViewController () <UITextViewDelegate,  
FontsViewControllerDelegate>
```

Run the app now and try out the font picker screen. This time, when you press the font button, the picker slides down and shows you the current font you're using in your text editor – by default, that is 14-pt Helvetica:



But it's not *sooo* impressive just yet, because when you change the font or the size, the new selection isn't applied back to the editor. To do this, you need to implement the action for the Apply button on the font selection screen.

The action method is already hooked up for you, so you need to just open **FONTsViewController.m**, find the `btnDoneTapped:` method, and paste inside:

```

//1
int selectedFontSizeIndex =
    [fontPicker selectedRowInComponent:0];
int selectedFontNameIndex =
    [fontPicker selectedRowInComponent:1];

//2
NSNumber* fontSize =
    ((NSArray*)fontsDataSource[0])[selectedFontSizeIndex];

NSString* fontName =
    ((NSArray*)fontsDataSource[1])[selectedFontNameIndex];

//3
[self.delegate selectedFontName: fontName withSize: fontSize];

//4
[self.navigationController popViewControllerAnimated:YES];

```

1. First you fetch the indexes for the two components of the picker control.
2. Then you grab the font size and font name from `fontsDataSource` using the indexes from the picker control.
3. You pass the font name and size to the delegate.
4. And you pop the current screen out of the navigation stack.

OK, nothing that'll break your head in here, so let's move on to the `selectedFontName:withSize:` delegate method in `ViewController`.

Open `ViewController.m` and add the method:

```

-(void)selectedFontName:(NSString*)fontName
withSize:(NSNumber*)fontSize
{
    NSDictionary* fontStyle = @{
        NSFontAttributeName :
            [UIFont fontWithName:fontName size: [fontSize floatValue]]
    };
    [self applyAttributesToTextArea:fontStyle];
}

```

This is something you've already done a few times. You create a new dictionary, and set the font style key to a new font with the selected name and size. You call your own `applyAttributesToTextArea:` method to apply the new font selection to the text.

Run again – now you can apply different font stylings to the text (don't forget to tap **Apply**)! Awesome!



The text editor is taking shape!

Applying the colors

Let's also do the color picker; colors will add even more pizzazz to note-writing!

Open **ColorViewController.h** and replace the code with this:

```
@protocol ColorViewControllerDelegate <NSObject>
-(void)selectedColor:(UIColor*)color;
@end

@interface ColorViewController : UITableViewController
@property (weak) UIViewController<ColorViewControllerDelegate>* delegate;
@end
```

As before, you set up a `delegate` property for the class, and a protocol to declare the method on the delegate class, which handles the color selection event. The class will work in a very similar manner to the font selector, so you can quickly implement all the functionality.

Open **ColorViewController.m** and replace the empty class extension at the top with this one:

```
@interface ColorViewController ()
{
    NSArray* colorsDataSource;
    NSArray* colorsNames;
}
@end
```

You define two arrays: the first will contain the list of colors, and the second will contain their names to show to the user.

Now find the `viewDidLoad` method and at the end of the method, paste the data for the two arrays:

```

colorsDataSource = @[
    [UIColor colorWithRed:0.106 green:0.49 blue:0.035 alpha:1],
    [UIColor colorWithRed:0.129 green:0.243 blue:0.725 alpha:1],
    [UIColor colorWithRed:0.725 green:0.129 blue:0.298 alpha:1],
    [UIColor colorWithRed:0.941 green:0.604 blue:0.02 alpha:1],
    [UIColor colorWithRed:0.941 green:0.02 blue:0.929 alpha:1],
    [UIColor colorWithRed:0.373 green:0.235 blue:0.035 alpha:1],
    [UIColor blackColor]
];

colorsNames = @[
    @"Green",
    @"Blue",
    @"Red",
    @"Orange",
    @"Pink",
    @"Brown",
    @"Black"
];

```

As you can see from the code, `colorsDataSource` contains the color objects, and `colorsNames` contains the human readable names of the colors in the list.

Now supply the correct data to the table view. Find `tableView:numberOfRowsInSection:` and replace its code with:

```
return colorsDataSource.count;
```

Now move on to the `tableView:cellForRowAtIndexPath:` method. Find the comment `// Configure the cell...`, and below that comment paste in the code to customize the table cell:

```

NSDictionary* attr = @{
    NSForegroundColorAttributeName:
        colorsDataSource[indexPath.row]
};

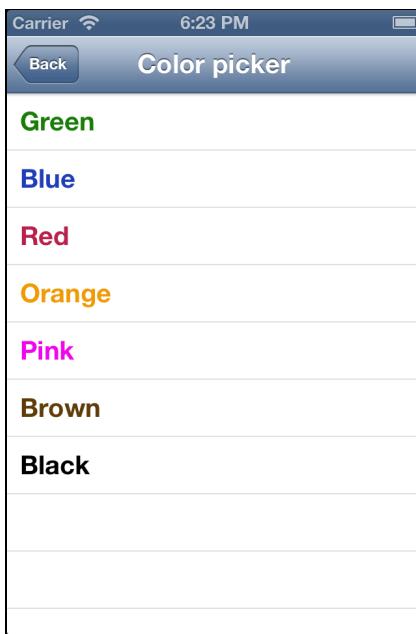
cell.textLabel.attributedText =
[[NSAttributedString alloc]
 initWithString:colorsNames[indexPath.row]
 attributes:attr];

```

Ah, that's an interesting piece of code! First you build a dictionary with one key – the foreground style key name. For the value, you fetch a color object from `colorsDataSource`.

Next you use the new to iOS 6 `attributedText` property of the label of the cell. Yes – the cell features a totally normal `UILabel`, so you can also use attributed text with it! You build an attributed string – for the text you use the color name, and you set the color of the text to the corresponding color.

Build and run the project right now to see the list of available colors. Open the editor and press the  button. Woot!



Now you can implement the functionality to apply the color selection back to the text editor.

Still in **ColorViewController.m**, find `tableView:didSelectRowAtIndexPath:` and paste inside:

```
[self.delegate selectedColor: colorsDataSource[indexPath.row]];
[self.navigationController popViewControllerAnimated:YES];
```

Just as with the font picker, you call the delegate method to pass the selected color, and then pop the screen out of the navigation stack.

Now switch to **ViewController.m** and at the end of `prepareForSegue:sender:` add:

```
if ([segue.identifier compare:@"colors"] == NSOrderedSame) {
    ColorViewController* screen = segue.destinationViewController;
    screen.delegate = self;
    return;
}
```

Since `ViewController` will now also be a delegate for the color picker, you need to add the appropriate protocol to the interface declaration. Scroll up and add

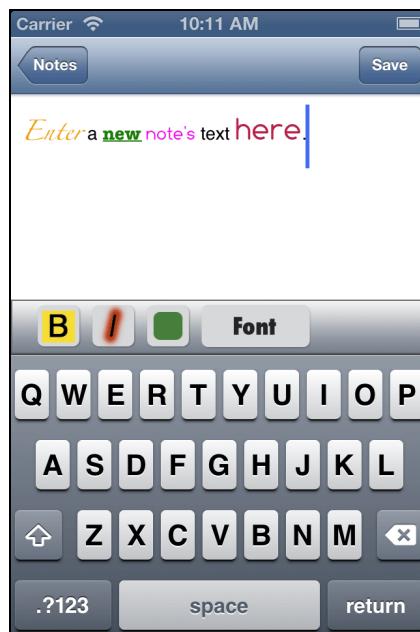
ColorViewControllerDelegate to the list of supported protocols to which the class complies. The final @interface line should look like so:

```
@interface ViewController () <UITextViewDelegate,  
FontsViewControllerDelegate, ColorViewControllerDelegate>
```

And finally, add the method to handle the user selecting a color:

```
- (void)selectedColor:(UIColor*)color  
{  
    NSDictionary* colorStyle = @{@"  
        NSForegroundColorAttributeName : color  
    };  
    [self applyAttributesToTextArea:colorStyle];  
}
```

You just apply the color to the text selection and you are ready to go! That was pretty fast, right? Hit Run and try out the color picker:



Whoa! Like it or not, that text has style! Your editor now features a variety of awesome text formatting.

At this point, you've learned 90% of what you'll need to know to use attributed strings on a regular basis, so if you want to take a break or move onto other chapters, now would be a good time to do so. ☺

But if you want to learn even more advanced things you can do with attributed strings like animating strings, saving strings, or applying additional formatting – read on!

Going beyond the UIKit limits with... attributed strings!

Using attributed strings is fun for a while, but when one wants to push the limits of good taste, then you need to get *really* creative. Let's see how to animate attributed strings, so you can achieve nice visual effects using only UIKit.

What you're going to do is apply an animated glowing effect to the Apply button in the font picker view controller. This is not particularly something that you are likely to use on a daily basis in your development, but I believe it will demonstrate that you can get far more creative with attributed strings than just coloring a word here and there in your app UI.

Open **FontsViewController.m** and add three more class variables inside the class extension:

```
NSTimer* timer;
float delta;
NSShadow* shadow;
```

You've already guessed it, I am sure – you are going to use a timer to continuously update the shadow of the Apply button, so that it looks like the text itself is glowing.

Let's do a bit of set up in `viewDidAppear:`. At the end of the method body, add:

```
shadow = [[NSShadow alloc] init];
shadow.shadowColor = [UIColor colorWithRed:0 green:0.42
    blue:0.039 alpha:1] /*#006b0a*/;
shadow.shadowBlurRadius = 0.0;
```

This code sets up the initial shadow for the button title – it's a healthy green glow.

Now you need to initialize the timer, which will continuously execute a method that updates the UI. Add these lines at the end of `viewDidAppear:` as well:

```
delta = 0.2;
timer = [NSTimer scheduledTimerWithTimeInterval:0.01
    target:self
    selector:@selector(glow:)
    userInfo:nil
    repeats:YES];
```

Let's not forget to also stop the timer in `viewDidAppear:`'s counterpart. Add the following to `viewDidDisappear:`:

```
[super viewDidDisappear:animated];
[timer invalidate];
timer = nil;
```

And now to actually update the button, add this new method, which will get called by the timer:

```
- (void)glow:(NSTimer*)t
{
    shadow.shadowBlurRadius += delta;

    if (shadow.shadowBlurRadius>6) delta = -0.2;
    if (shadow.shadowBlurRadius<0) delta = 0.2;

}
```

The task of this method is to update the shadow variable and increase (or decrease) the blur radius constantly by the value of the delta variable.

When the timer first starts to fire, delta has a positive value, 0.2, so the blur radius keeps growing for a while. Then when it reaches 6 pixels, the first `if` in the method body updates the `delta` variable with a negative value, so that as the timer keeps firing, the blur radius steadily decreases. When the radius reaches 0 pixels of blur, the second `if` statement kicks in and sets `delta` back to a positive value.

And thus the blur radius continuously grows and then shrinks, and then grows again.

What is left now is to add the code to the `glow:` method to update the UI. To make sure you are always running on the main thread, which can update the UI, you will dispatch a block of code to make the adjustments to the button.

At the end of the `glow:` method, add this code:

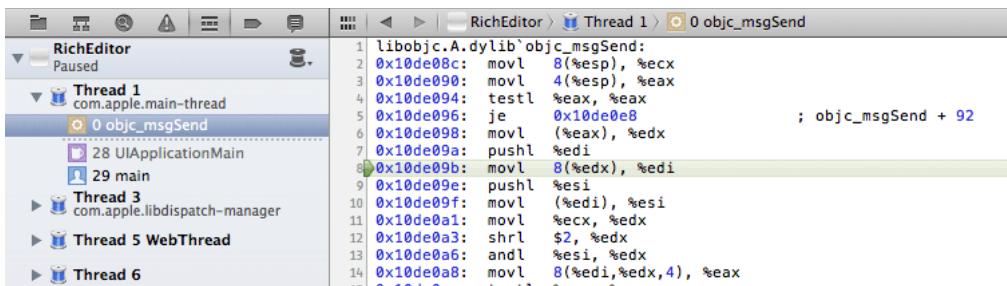
```
dispatch_async(dispatch_get_main_queue(), ^{
    NSAttributedString* title =
        [[NSAttributedString alloc]
            initWithString:@" Apply "
            attributes: @{@"NSShadowAttributeName":shadow}];

    [btnApply setAttributedTitle:title
                forState:UIControlStateNormal];
});
```

This doesn't look complicated, right? You just create an attributed string with the text " Apply " and you apply to it the current shadow object. In the end, you set it as the title of the button.

Note: Putting extra space around the word "Apply" in the button title leaves room for the shadow to render inside the label; otherwise, it will look cut off.

Hit Run and open the font picker screen to see the result:



Ack – it crashed! Hmm, it looks as if things are going wrong for some reason. Instead of using the shadow object directly, let's try making a copy of it on the main thread and applying that one to the button title.

Delete the last bit of code you added and replace it with this better one:

```
dispatch_async(dispatch_get_main_queue(), ^{
    NSShadow* mainThreadShadow = [shadow copy];
    NSAttributedString* title =
        [[NSAttributedString alloc] initWithString:@" Apply "
        attributes:@{NSShadowAttributeName:mainThreadShadow}];

    [btnApply setAttributedTitle:title
        forState:UIControlStateNormal];
});
```

Ah! Now the project runs and you can see some awesome eye-candy implemented using only UIKit and attributed strings!



If you want to create an even funkier effect, try changing the title color to white to see what happens.

Saving attributed strings for later

Now it's time to look into the last, but extremely interesting and significant, part of your Notes-on-Steroids app.

The whole purpose of the Notes app is to give the user the opportunity to save pieces of text, which can then be read at a later point. Your app lacks this feature at present, so buckle up and prepare to bring it in line!

You probably already know how to save a piece of text into a file on iOS. It's very easy to do, as `NSString` has a handy method called

`writeToFile:atomically:encoding:error:` that allows you to save a string's content to a text file on the disc.

Unfortunately, `NSAttributedString` (not being a sub-class of `NSString`) does not feature a method like the one above, so you will need to handle the saving task on your own.

I could take you through building an attributed string persistence class, written from scratch, but it will be much better for you to look into a certain library that will allow you to persist not only attributed strings, but all kinds of other classes.

Meet `NSObject+AutoCoding`! This is a handy `NSObject` category written by Nick Lockwood that takes care to automatically add serialization to your classes. But not just to *your* classes – since all classes inherit from `NSObject`, it adds automatic coding to every single one of them!

Woot!

Open the resources folder for this chapter and copy the **Extra Classes.zip** file, then extract it and copy **NSObject+AutoCoding.h** and **NSObject+AutoCoding.m** into your Xcode project.

Note: If you want to learn more about the author of `AutoCoding`, have a look at the latest source code, or are just curious, the project URL is:
<https://github.com/nicklockwood/AutoCoding>.

Right now you have only the (+) button working, which creates a new file. So let's implement the Save button. First you will make the app's initial view controller (`FileListViewController`) set the file name for the editor to open.

Open `FileListViewController.m` and find the method `prepareForSegue:sender:`. There is already a segue hooked up for the (+) button, so do the file name setup for the new note file in there. Add these two lines to the method:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(
    NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

This code fetches the documents directory, where you will save your note files.

Let's also add the code to generate the file name for the new note:

```
if ([segue.identifier compare:@"AddButton"] == NSOrderedSame) {
    ViewController* screen = segue.destinationViewController;

    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc]
        init];
    [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
```

```
[dateFormatter setTimeStyle:NSDateFormatterMediumStyle];

NSString *dateString = [dateFormatter stringFromDate:
    [NSDate date]];

NSString* fileName = [documentsDirectory
stringByAppendingPathComponent: [NSString
stringWithFormat:@"%@", dateString]];

//create a new note
screen.fileName = fileName;

return;
}
```

First you fetch the target view controller of the segue being invoked. Then you create a date formatter and create a new string `dateString`, which contains the current date and time – this will be the name of the new file. You add the `.plist` file extension and store the result in `fileName`. Finally, you assign the new file name to the `fileName` property of the target view controller.

Of course, you don't yet have the `fileName` property, which must mean it's time to implement it! Open **ViewController.h** and below the `@interface` line, add the new property declaration:

```
@property (strong, nonatomic) NSString* fileName;
```

Now that your editor class has a file name to work with, you can implement the method to save the file contents.

Open **ViewController.m** and at the top of the file, under all the other import lines, add one more:

```
#import "NSObject+AutoCoding.h"
```

The Save button is already hooked to an action called `btnSaveTapped:`, so find this method and add the following code to it:

```
[editor.attributedText writeToFile:self.fileName
    atomically:YES];
[[[UIAlertView alloc] initWithTitle:@"Success"
    message:@"Note saved"
    delegate:nil
    cancelButtonTitle:@"Close"
    otherButtonTitles: nil] show];
```

Wait... what? ☺

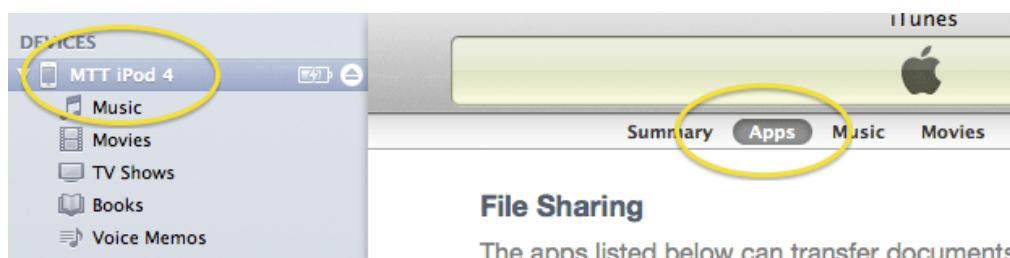
Yes, `NSObject+AutoCoding` is so smart, it just adds a `writeToFile:atomically:` to all objects. There are, of course, some limitations, but from my own tests so far, it works pretty well on a number of my own classes.

Cool! You've just implemented saving rich text notes. Thank you, `NSObject+AutoCoding`!

This time, run the app on your device (there's a reason for running on device, though you can also run on the Simulator, if you want), write a note and hit the Save button.



Now fire up iTunes and select your device in the left-hand sidebar, then select the Apps tab on the right-hand side:



Scroll down and you will see a list of apps on your device that have document sharing enabled. You should now see your saved Note file:



Note: If you are not familiar with document sharing in iTunes, open the **RichEditor-Info.plist** in Xcode and have a look at the list of properties. There's a Boolean property called "Application supports iTunes file sharing" and it is set to YES – that's all you need to enable to share your app's Documents folder to iTunes.

For more information, check out this tutorial:

<http://www.raywenderlich.com/1948/how-integrate-itunes-file-sharing-with-your-ios-app>

Now you need to implement the loading and editing of existing notes.

Open **FileListViewController.m**. You want to build a list of all the .plist files in the app's Documents directory, and show them in the table view on the initial screen.

Scroll up and replace the class extension with this one:

```
@interface FileListViewController ()  
{  
    NSArray* fileList;  
}  
@end
```

You need a new method to separate the functionality that loads the file list. Call it `loadFileList` and add it to the class:

```
-(void)loadFileList  
{  
    NSFileManager *fm = [NSFileManager defaultManager];  
    NSArray *paths = NSSearchPathForDirectoriesInDomains(  
        NSDocumentDirectory, NSUserDomainMask, YES);  
    NSString *documentsDirectory = [paths objectAtIndex:0];  
  
    fileList = [fm contentsOfDirectoryAtPath:documentsDirectory  
                           error:nil];  
}
```

You use the default file manager instance to get the list of document folders. You want the first result – that's your app's Documents directory. Then you call `contentsOfDirectoryAtPath:error:` to get the list of files in the given directory and store it in `fileList`.

Find `viewDidAppear:` and replace the placeholder code inside with this:

```
[super viewDidAppear:animated];  
[self loadFileList];  
self.title = [NSString stringWithFormat:@"Fancy Notes (%i)",  
            fileList.count];  
[self.tableView reloadData];
```

When the view appears on the screen, you call your new `loadFileList` method, update the title of the view controller, and finally refresh the table view.

Next you need to adjust the table view delegate methods to provide accurate data to the view. Find `tableView:numberOfRowsInSection:` and replace the placeholder code inside with:

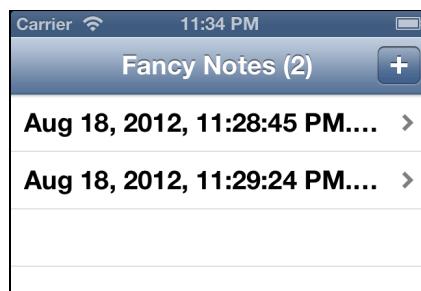
```
return fileList.count;
```

Now find the `tableView:cellForRowAtIndexPath:` method, and just below the `//configure the cell...` comment, add this line:

```
cell.textLabel.text = fileList[indexPath.row];
```

This piece of code simply sets the title of the table cell to the name of the note file.

That should be everything you need to do in order to show the list of saved notes. Run the project and give it a try:



The title shows the number of saved notes (just like in the default Notes app) and the list shows you the date and time each note was created.

The final step is to handle row selection in the table view and load the respective file contents. The cell is already hooked up for you in the project storyboard file. You just need to do some extra setup for when the segue fires.

Go to `prepareForSegue:sender:` in **FileListViewController.m** and add the following to the bottom of the method:

```
if ([segue.identifier compare:@"Edit"] == NSOrderedSame) {  
    //1  
    ViewController* screen = segue.destinationViewController;  
  
    //2  
    int selectedIndex = [self.tableView  
indexPathForSelectedRow].row;  
  
    //3  
    NSString* selectedFileName = fileList[selectedIndex];  
  
    //4  
    screen.fileName = [documentsDirectory  
stringByAppendingPathComponent:selectedFileName];  
  
    return;  
}
```

There isn't anything really complicated in this piece of code – if the segue name equals "Edit", then:

1. You get ahold of the target view controller.
2. You get the index of the selected table cell.
3. You get the file name from `fileList`, using the index of the selected cell.
4. Finally, you store the full path to the note file in the `fileName` property of `ViewController`.

OK, that's all you need to do in the file list screen. The last remaining piece of functionality to implement is the code in `ViewController` to load the file contents back into the text editor.

Open **ViewController.m** again, and at the end of `viewDidLoad` add the code to load the contents of the note:

```
//1
self.title = [[self.fileName lastPathComponent]
              stringByDeletingPathExtension];

@try {
//2
    editor.attributedText = [NSAttributedString
        objectWithContentsOfFile:self.fileName];
}
@catch (NSEException *exception) {
//3
    editor.attributedText = [[NSAttributedString alloc]
        initWithString:@""];
}
```

Here's what the above code does:

1. You call `lastPathComponent` on the full path to the note file in order to get only the file name. Then you use `stringByDeletingPathExtension` to remove the ".plist" part of the file name, and you set the result as the title of the view controller.
2. The `NSObject+AutoCoding` category adds a method called `objectWithContentsOfFile:` to all classes deriving from `NSObject`. This method is the counterpart to `writeToFile:atomically:` that you used earlier to save the note's contents to a file. The method will throw an exception if the file doesn't exist or the contents can't be read (thus, when the user is creating a new note, the method throws an exception and the `catch` block is executed).
3. In the `catch` block, you just initialize the text editor with an empty string.

OK, the loading of notes is also implemented! Run the project and try it out. You can create new notes, save your work, load existing notes and also edit existing notes and save the modified version of your text.



Advanced attributed text tricks

To end this chapter on new UIKit attributed strings functionality, you're going to take some advanced APIs for a spin.

More specifically, you'll have a look at:

- Modifying the text contents of an attributed string.
- Getting the rendering height of an attributed string by restricting it to a given width.
- Drawing an attributed string into an image context.

As it relates to your Notes-on-Steroids app, you're going to make the file list of your app show not the date and time when the note was created, but a preview of the note's contents. When you're done, the initial screen of the app will look something like this:



You can see that this is definitely an easier way to browse through notes, as compared with trying to remember what you wrote on a specific date and time. So let's go!

First, since you are going to be loading some rich text content from `FileListViewController`, you will have to include the `AutoCoding` category. Open

FileListViewController.m and scroll to the top. Under the last import line, add this one:

```
#import "NSObject+AutoCoding.h"
```

A few lines below, within the curly brackets of the `@interface` declaration, add a new class variable:

```
NSMutableDictionary* renderedStrings;
```

The plan is to cut a small piece of text out of the note content, then show it in a table cell. You can, of course, just assign the content to the `attributedString` property of the table cell, but (as you probably know if you have been developing for iOS for a while) rendering heavy in-cell components makes table scrolling somewhat jumpy.

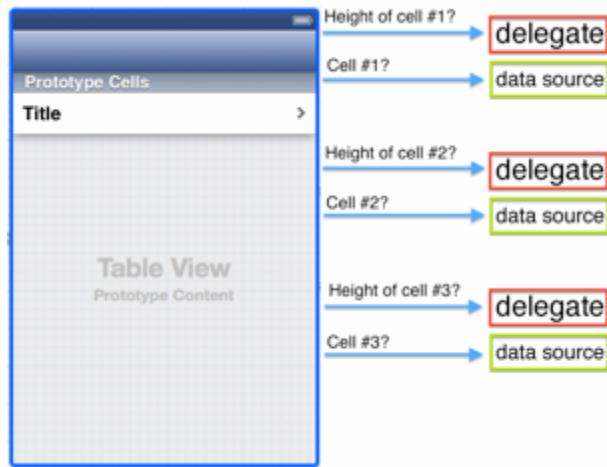
A common approach to optimize scrolling performance is to pre-render the content of the cells as images, and then use those images for the actual cell rendering. This is the approach you'll take to show your notes previews – this way, you'll learn about rendering attributed strings!

As you've probably figured out, you'll be storing the rendered strings in the `renderedStrings` dictionary that you just added to the class interface.

So far, you haven't had to do any set up for the class, but now you need your `renderedStrings` dictionary initialized. So add the following code:

```
-(void)viewDidLoad
{
    renderedStrings = [NSMutableDictionary
        dictionaryWithCapacity:5];
}
```

As you might already know, while table view contents are rendered, the table view continuously keeps asking its delegate and data source about the height and contents of each cell, like so:



Here's what you're going to do: in the delegate method that calculates the height of each cell in the table view, you'll fetch the contents of the corresponding note file, grab a preview of the text, render it to an image and return the height of the rendered image.

While doing that, you are also going to store the image in `renderedStrings`, so that when the table view asks the data source (which is again your `FileListViewController` class) to provide the cell object to show, you can just set the already-rendered image as the cell contents.

Let's proceed according to the plan. Add this initial version of the method that calculates the height of each table cell:

```
-(CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];

    NSString* filePath = [documentsDirectory
        stringByAppendingPathComponent: fileList[indexPath.row]];

    NSAttributedString* contents =
    [NSAttributedString objectWithContentsOfFile: filePath];
}
```

Let's consider what's going on in this new method so far. First, just as you did before, you fetch the Documents directory for your app in `documentsDirectory`. You append the file name for the provided `indexPath` (fetching it, of course, from `fileList`) and you store that path into `filePath`. Finally, you call

`objectWithContentsOfFile:` to load the file contents and store it in the attributed string `contents`.

Next, add this line to grab only the first few words of the text:

```
contents = [contents attributedSubstringFromRange:  
    NSMakeRange(0, MIN(30, contents.length))  
];
```

`attributedSubstringFromRange:` is the method to get substrings out of an attributed string. It takes in a range, and returns an attributed string. You grab the beginning of the note text, and by using the `MIN()` function, you ask for at most 30 characters.

OK, so far you've got the preview text stored in `contents`. Let's add a "..." at the end of the preview text to indicate that the rest has been cut off.

This is a seemingly simple task, but it turns out that it takes quite some typing. First you need a mutable copy of the `contents` attributed string, then you need to create another attributed string and initialize it with the plain text "...", and only then can you append them together. Add the code to do this:

```
NSMutableAttributedString* mContents =  
    [[NSMutableAttributedString alloc]  
    initWithAttributedString:contents];  
  
NSAttributedText* dots = [[NSAttributedText alloc]  
    initWithString:@"..."];  
  
[mContents appendAttributedString: dots];
```

You've got the final text to render, so it's time to detect how big of a canvas you need for the image. Fortunately, there's a handy method to provide you with that information, so go ahead and add this line:

```
CGRect bounds = [mContents  
    boundingRectWithSize:CGSizeMake(300, 10000)  
    options:NSStringDrawingUsesLineFragmentOrigin  
    context:nil];
```

As the first parameter, you supply a bounding size in which you would like to draw the string. You make the size out of the real width you want to use – 300 points – and a ridiculously big bounding height. You do that so as to bound the drawing algorithm to the width you want and actually check how much height it's going to take to draw the string.

To make the method respect the width of the bounding box, you supply `NSStringDrawingUsesLineFragmentOrigin` for the drawing options parameter (if you

don't, the attributed string will always be rendered on one line, and therefore the returned height won't be of any use to you for the current project).

Now `bounds` contains the height of the table cell! It is, however, too early to return the height. You still need to render the string as an image and store it for future use.

Drawing an attributed string in an image context is quite straightforward. Add this code:

```
UIGraphicsBeginImageContextWithOptions(bounds.size, NO, 0.0);

[mContents drawWithRect:bounds
    options:NSStringDrawingUsesLineFragmentOrigin
    context:nil];

UIImage *renderedText =
    UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();
```

First you create a new image context by calling

`UIGraphicsBeginImageContextWithOptions`. Its parameters are as follows:

- The size of the image context (you use the bounds you got for rendering the string).
- Is the image fully opaque? – `YES` or `NO`. `NSAttributedString` by default renders the text on a transparent background, so you need to provide `NO`.
- The final parameter is the scale factor to apply to the image context. If you provide `0.0`, the default scale factor for the device's screen is used.

The method that actually draws the string to the current image context is `drawWithRect:options:context:` (on the next line of code). You supply all the same arguments – the bounds for the drawing and the `NSStringDrawingUsesLineFragmentOrigin` option you used to calculate the bounding box of the string.

`UIGraphicsGetImageFromCurrentImageContext()` is what gives you back a `UIImage` instance out of the current image context, and at the end you call `UIGraphicsEndImageContext()` to finish drawing and end the image context.

The heavy lifting is done – you have the string rendered in a `UIImage`, so you only need to store it for future use, and you're finished.

Add these final lines to the method body:

```
renderedStrings[fileList[indexPath.row]] = renderedText;

return bounds.size.height;
```

You store the rendered `UIImage` in the `renderedStrings` dictionary and return the height of the drawing bounding-box as the cell height. *Finito!*

Phew! This method was quite long, but you covered all the APIs I promised you, right? You now know how to get the size of an attributed string, how to draw it, export it as a `UIImage`, and how to modify the string's contents. You covered quite a lot of material!

One final thing remains, though, before you wrap up the fancier file list for your Notes app.

Scroll to `tableView:cellForRowAtIndexPath:` and replace the following line:

```
cell.textLabel.text = fileList[indexPath.row];
```

With this code:

```
cell.textLabel.text = nil;
cell.imageView.image = renderedStrings[fileList[indexPath.row]];
```

The first line instructs the cell that you don't want any text rendered inside it. On the second line, you grab the rendered preview string from `renderedStrings` and you feed it to the cell's image view.

And that really is everything! Run the project and you should see the file list featuring previews of your saved notes. Your Notes-on-Steroids app is finished!



You are now a master of the new iOS 6 `NSAttributedString` class! There's certainly more to discover in the API documentation, but you have acquired a solid foundation of knowledge that includes most of what's available for use.

Where to go from here?

You are certainly rocking `NSAttributedString`'s world by now. You are a real pro at:

- Building attributed strings.
- Setting text and paragraph formatting.
- Using attributed strings with UIKit controls like `UILabel`, `UIPickerView`, `UIButton`, and `UITextView`.
- Using text formatting in Interface Builder.
- Modifying existing text attributes, processing text formatting runs, and more!
- Persisting attributed strings as .plist files.
- Getting a string's metrics, rendering, and exporting as a `UIImage` instance.

Just look at the list – wow! But hey, you can take things even further yourself! You could:

- Try all the formatting styles that didn't make it into this chapter. Seeing is believing, so as you keep trying out the different formatting options, you will get ideas of how to use the options in your own apps.
- Further develop the Notes-on-Steroids app – it still lacks features: deleting notes; giving notes custom titles; sharing notes via email, iCloud, Facebook, etc.; encrypting notes; and much more you could think up.
- Why not look into loading HTML formatting into an `NSAttributedString`? This question seems to always come up in discussions, and since attributed strings are now integrated directly into UIKit, using HTML to format the UI is even more interesting. Have a look at this project, which takes on parsing HTML and producing attributed strings out of it:
<https://github.com/Cocoanetics/DTCoreText>.

I wish you lots of fun using the new iOS 6 attributed strings in your apps, and I hope this chapter was a useful and rich resource!

Chapter 16: State Preservation and Restoration

By Matthijs Hollemans

A long time ago, in a galaxy far, far away, Apple released iOS 4 and introduced multitasking, and there was a new hope.

Before multitasking was available, tapping the Home button or answering an incoming phone call would always terminate the current app. Since iOS 4's "fast app switching" feature, apps no longer terminate outright but instead go to sleep and stay in the memory of the device in a suspended state. When the user switches back to an app, it simply wakes up and resumes exactly where it left off.

That's great for the user, because they don't have to wait for your app to load every time they want to use it, and they don't have to retrace their steps to get back to where they were inside the app. Fast app switching definitely improves the user experience.

But there's one problem - there is no guarantee that any of this will actually happen!

There is only so much memory in today's iPhones and iPads, and if the device runs out of memory, it has to remove suspended apps to make room. If the user switches to a game or another memory hog, then your app may still be terminated and unloaded from the device's memory. The iOS strikes back!

When this happens, it does not create a very good user experience if the app forgets all about what the user was doing. For example, if Jane User was typing a message when she was interrupted by a phone call, she expects to be able to resume typing her message when she switches back to your app.

Your apps need to be prepared to go back to the view controller that was active when the app was moved to the background, even if it was terminated, and to restore any data the user was working on.

Prior to iOS 6, you had to handle this yourself by saving application state to `NSUserDefaults` or some other type of temporary file. This worked, but was a major pain so a lot of developers didn't even bother, resulting in some poor user experiences.

Luckily, with iOS 6's new State Preservation and Restoration feature, this process is a lot easier! You have been granted new Jedi powers in the never-ending struggle between boundless human expectations and finite memory.



The goal is to always present to the user the illusion that the app never quit. The app should resume right where it left off, and bring the user back to exactly where they were and what they were doing.

Think of this as one of those indispensable invisible features – you only notice it when it isn't there, but it critically improves the user experience. And user experience is what, in the long run, sells apps!

(Re)introducing the Ratings app

To demonstrate how State Preservation and Restoration works, this chapter shows you how to build this feature into an existing app. The app is called "Ratings" and lets you keep track of how good your friends are at playing various games. If you have read *iOS 5 by Tutorials*, you may recognize this app from the Storyboards chapters.

You can find the source code for the initial version of the app among the resources for this chapter. It currently loads and saves the user's data (the list of players and their scores), but totally ignores any state information.

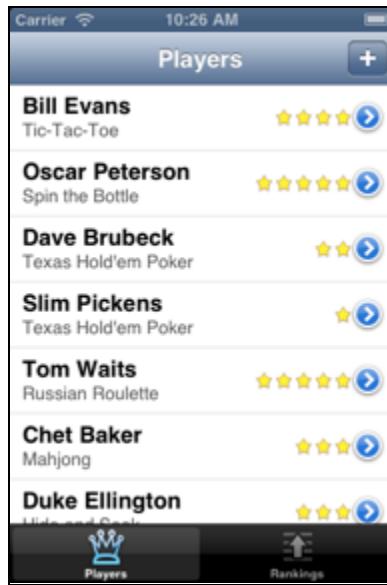
The app's storyboard looks like this:



Just so you get a high-level overview of how the app works, this section describes each of its screens and gives a quick summary of the view controllers. Feel free to follow along in the source code.

On the other hand, feel free to skim or skip this section if you have a good memory of the structure of the app.

The main screen, "Players," lists the names of the players and their scores:



This is a table view controller that goes by the name of `PlayersViewController`. Each table view cell is a prototype cell with a custom class, `PlayerCell`. The players and their scores come from an array of `Player` objects.

As you can see in the storyboard, this screen sits inside a navigation controller that in turn sits inside a tab bar controller.

You can do three things from this screen: the + button at the top opens the Add Player screen for adding new players; the blue detail disclosure button in each row (the > icon) lets you edit that player; and tapping a row brings up the Rate Player screen:



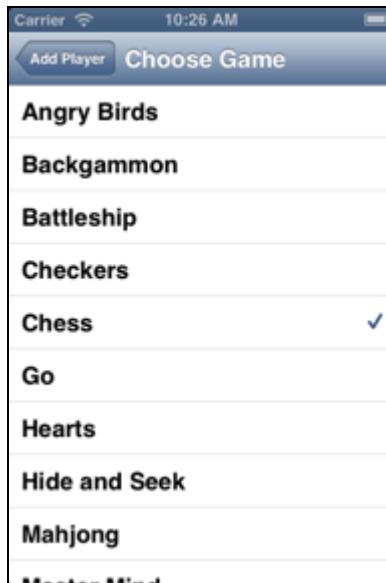
The Rate Player screen is represented by the `RatePlayerViewController`. This view controller gets pushed on the navigation stack, so it has a back button that lets you return to the main screen. To change the rating for a player, you tap one of the big buttons with stars on them.

The Add Player screen looks like this:



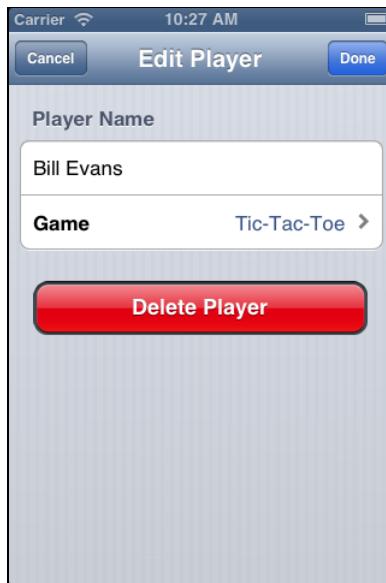
There is an entry field for the player's name, and a row that lets you pick the game for this player. This screen is implemented by the `PlayerDetailsViewController` and always gets presented modally. It's simply a table view controller, with static cells, that sits in its own navigation controller (refer to the storyboard diagram).

Tapping the game row brings up the Choose Game screen:



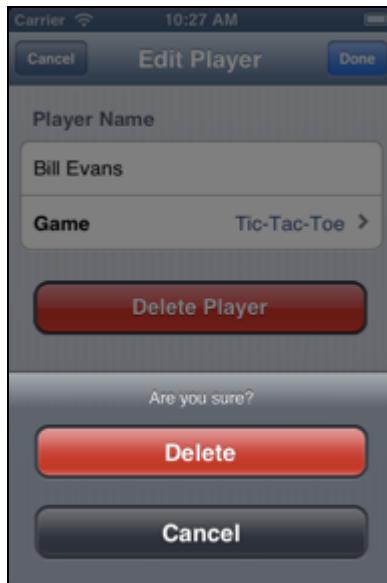
This screen, `GamePickerController`, is pushed on the navigation stack. It's just a table view controller with a list of games. You pick a game by tapping on a row, after which the screen pops back.

The Edit Player screen is very similar to Add Player, which is not so strange, because it is the exact same class, `PlayerDetailsViewController`:

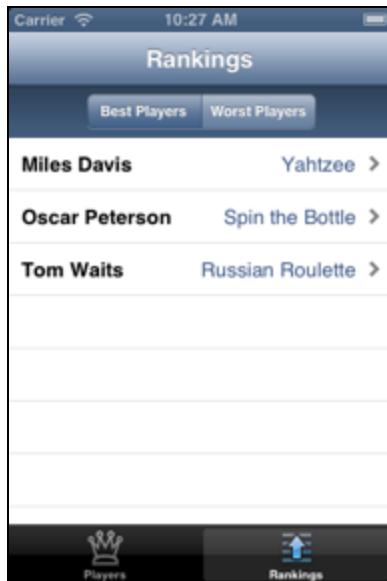


`PlayerDetailsViewController` has a property called `playerToEdit`. If that property is `nil`, the class acts as the Add Player screen. But if `playerToEdit` contains a valid `Player` object, the view controller morphs into the Edit Player screen, fills in the fields with the player's data, and adds a big red delete button.

Tapping the delete button brings up an action sheet for confirmation:



Let's hop back to the main screen. Recall that it has two tabs, "Players" (which you've already seen) and "Rankings." The Rankings screen looks like this:



This screen also lists the players and their games (again, `Player` objects), but ranked either by best (players with 5 stars) or worst (players with 1 star). The segmented control on top lets you choose which. This is the `RankingsViewController`. It just presents a different view of the data.

Run the app and play around with it for a few minutes. Also take a look at the source code. If you've ever programmed with storyboards and table views before, you will find it very straightforward.

Note: The `Player` class conforms to the `NSCoding` protocol so that the array of players can be saved to a PLIST file. The first time you run the app, it puts a bunch of default player objects into the array, just so you have something to work with, but from then on the App Delegate will save the `Player` objects to a file when the app goes to the background.

Going to the background

To set the stage, let's see what happens when you put the app into the background and then bring it back up again.

Run the app and tap the + button to add a new player. Type a name and pick a game:



Now tap the Home button on the Simulator. The app disappears and the Simulator's springboard comes to the front. Tap the icon for the app to open it again:



The app should return to the screen right where you left it. The text field contains the exact same text and the game row shows the name of the game you picked. If the keyboard was showing before, it is still showing now.

Press Xcode's Stop button to terminate the app:



When you tapped the Home button, the application became suspended, but it stayed in the iPhone's memory. It wasn't really gone, just sleeping. When you woke it up again, it simply picked up from where it left off. But with the Stop button, the application really does stop running; it is not kept in a suspended state.

This is similar to what a user might do by double-tapping the Home button and removing the app from the list of recently-used applications:



Now when you click Xcode's Run button again, the application cannot resume from where it was – it has to launch fresh from the beginning. To the app, this might as well be the very first time it was ever run. Instead of going back to the Add Player

screen, you end up on the main screen again. Essentially, the app has thrown away some of your data: the name of the new player and the game you picked.

Why is this important? After all, the user had to exit the app manually to make this happen, right?

Well, not always. Apps can only stay suspended in memory as long as there is enough memory to keep them there. If the user switches to another app that needs a lot of memory to operate – such as a graphics-intensive game or an app that uses the photo library – then chances are that any suspended apps will get kicked out of memory.

You as the developer have no control over this, so you should assume that your app could get terminated at any point. Users will have a better experience if the app always goes back to the state it was in when they switched apps, even if the app got killed in the meantime.

Note: For the purposes of this chapter, you will be killing your apps a lot (don't worry, it won't hurt them). It works best if you do this using Xcode's Stop button, or by pressing **Cmd-**. (the period key). Xcode will get very confused if you launch an app from Xcode but terminate it by removing its icon from the recently-used apps list.

State Preservation and Restoration

Hopefully I have convinced you that it is a good idea to always put your app back into the state that the user left it in, even if the last time they used the app was many moons ago – although you'd hope that your apps get used every day!

But let's be honest, you always skipped writing the code for this because it's a ton of work and you have more important things on your to-do list, right? I know I should have been more diligent about it in my own apps, that's for sure.

Well, there is good news for slackers like you and me: iOS 6 makes state restoration a lot easier, and in many cases, almost automatic.

State Preservation and Restoration is an opt-in feature. You need to specifically enable it in your apps. While UIKit is pretty smart, it isn't smart enough to understand the purpose of your app. You still need to tell it what is important to preserve and restore and what is not.

Enabling this feature is pretty easy. Add the following two methods to the application delegate in **AppDelegate.m**:

```
- (BOOL)application:(UIApplication *)application  
shouldSaveApplicationState:(NSCoder *)coder
```

```
{  
    return YES;  
}  
  
- (BOOL)application:(UIApplication *)application  
    shouldRestoreApplicationState:(NSCoder *)coder  
{  
    return YES;  
}
```

Preservation and restoration are two different steps. When your app goes into the background, the *preservation* step happens and UIKit will ask your view controllers and views to encode any information that they want to persist. At some later point, when your app comes back to life, UIKit performs the *restoration* step and passes that saved information back to the view controllers and views.

However, there may be specific cases where you don't want restoration to happen – for example, after your app has been updated to a new version that is significantly different from the old version the user had been using – and in that case you can return `NO` from `application:shouldRestoreApplicationState:`.

This code by itself is not enough to enable state restoration, but let's run the app anyway. Switch to the Rankings tab, then press the Home button to send the app to the background.

Note: This step is important! If you don't send the app to the background before you terminate it, UIKit will not perform the state preservation pass!

Once the app is in the background, kill the app from within Xcode using the Stop button. After the app has terminated, click the Run button to start it up again. The app will now activate the Players tab, not the Rankings tab. So it did not remember your previous selection.

However, you may have seen a brief flash where the second tab was active when the app started. What was that?

Whenever your app goes to the background, iOS makes a temporary screenshot of the contents of the screen. As the app becomes active again, it shows that screenshot to give the user the illusion the app is loading faster than it really is – similar to how the Default.png image is shown during startup. If you have state preservation enabled, iOS displays this screenshot when your app starts up because you are saying that you support getting the user back to their previous state – and that is what you're seeing here.

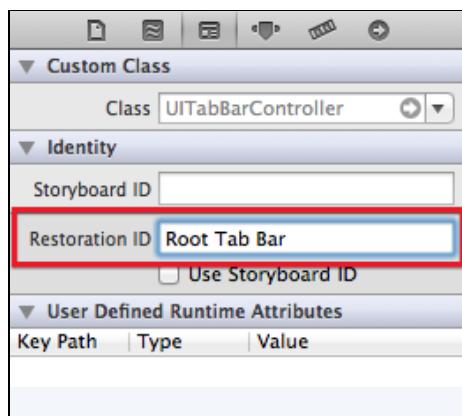
But the effect is misleading (and a bit jarring) because the app doesn't currently stay on the second tab. If you see such a "flashing image" after launching your app, you know that your state restoration isn't working properly yet.

Restoration identifiers

UIKit doesn't automatically save everything there is to know about the state of your app. This is an opt-in feature, and you have to tell UIKit which of your view controllers should partake. You do this by assigning those view controllers a *Restoration Identifier*. Anything that has a Restoration ID will get preserved and restored.

You can either set these IDs via code or directly in the storyboard editor. The latter is the simplest option, so that is what you will do in this chapter. It is important that you also set Restoration IDs on the navigation controllers and tab bar controllers, not just on your own view controllers!

It's easy to set your Restoration IDs – let's try it out. Open **MainStoryboard.storyboard** and select the Tab Bar Controller. In the Identity inspector, type "Root Tab Bar" into the Restoration ID field:



Also set Restoration IDs on the two navigation controllers that are attached to the Tab Bar Controller. Name the first "Players Navigation" and the second "Rankings Navigation." It doesn't really matter what you call them, as long as these names are unique.

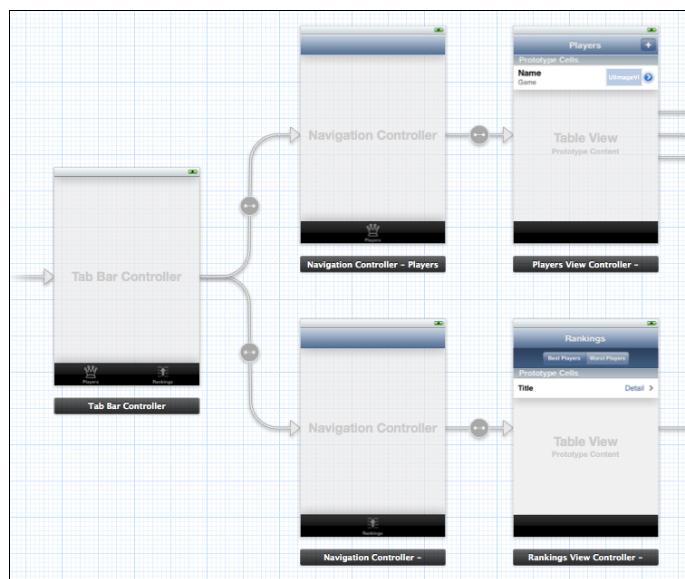
Finally, set a Restoration ID on the Players View Controller. Since the ID can be anything you fancy, I find it just as easy to re-use the name of the class, in this case "PlayersViewController":



Note: Make sure you're setting the Restoration ID on the actual view controller, not on its main view. Sometimes when you click on a scene in the storyboard editor, it doesn't select the view controller but the top-level view. Views can also have a Restoration ID, but that is not what you want to do here. So make sure you have the right element selected before you set the ID.

Also give the Rankings View Controller (the one that is connected to the second tab) a Restoration ID. Again, simply use the name of the class, "RankingsViewController."

You should now have set Restoration IDs on these five view controllers, but not on any of the others:



Press the Stop button to terminate the app, and then Run it again. Inside the app, switch to the Rankings tab. Press Home to move the app to the background and click the Stop button to kill it once more.

Now after you click Run, the app will launch anew and immediately switch back to the Rankings tab. The app pretends that you never left this tab. Congrats, it's working!

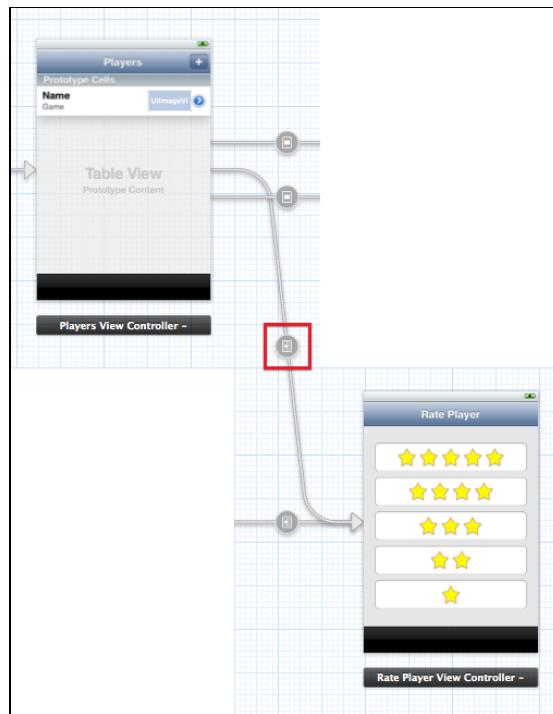
Of course, you could have coded all of this yourself, by storing in `NSUserDefaults` the index of the tab the user was on. As the application launches, you would look at this value and tell the `UITabBarController` to switch to that tab.

But that is a lot more work than just setting a few Restoration IDs! With iOS 6, you get all that for free.

The Rate Player screen

Let's move on to the next screen, Rate Player. In the storyboard, give this scene its own Restoration ID, "RatePlayerViewController."

The Rate Player View Controller gets pushed onto the navigation stack by a segue. Here is this segue in the storyboard:



Stop the app and then Run it again. Tap on any one of the rows from the list of players to make the Rate Player screen appear:



Press the Home button, kill the app, and re-launch it. The same screen reappears – thanks to setting the Restoration ID – so that's good, but it does look a little different:



The title no longer has the name of the player, but says just “Rate Player.” Tapping the buttons with the stars no longer has any effect – try it out – although the Back button still works.

What happened here?

Remember that the application got launched from scratch. Normally when an app comes back from being cryogenically frozen in the background, all the objects are still the same objects because they are still in the phone’s memory. But when an app is re-launched, it creates all new objects instead. It has to, because all the old ones have long been erased from memory.

That’s exactly what happened here. In order to restore the Rate Player screen, UIKit made a new instance of `RatePlayerViewController` and pushed it on top of the navigation stack. But that’s where UIKit stopped; it didn’t restore any of that view controller’s contents.

Let’s look at **RatePlayerViewController.h**:

```
@interface RatePlayerViewController : UIViewController

@property (nonatomic, weak) id
    <RatePlayerViewControllerDelegate> delegate;

@property (nonatomic, strong) Player *player;

@end
```

This view controller has two public properties: `delegate` and `player`. When you tap on a row in the Players screen, it performs a segue to the Rate Player screen. Those two properties are set in the `prepareForSegue:` method that takes place for that segue.

You can see this method in **PlayersViewController.m**:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                  sender:(id)sender
{
    . . .

    else if ([segue.identifier isEqualToString:@"RatePlayer"])
    {
        RatePlayerViewController *ratePlayerViewController =
            segue.destinationViewController;
        ratePlayerViewController.delegate = self;

        NSIndexPath *indexPath = [self.tableView
                                  indexPathForCell:sender];
        Player *player = (self.players)[indexPath.row];
        ratePlayerViewController.player = player;
    }
}
```

When the “RatePlayer” segue happens, the `RatePlayerViewController`’s `delegate` property is set to be the `PlayersViewController`, and the `player` property is given the `Player` object for the selected row. So far, so good.

However, during the state restoration process, UIKit knows nothing about this segue. It just creates a new instance of `RatePlayerViewController` and pushes it on the navigation stack. That’s all. As a result, the `delegate` and `player` properties are never given any values and remain `nil`.

That’s why the player’s name no longer appears in the title – it comes from the `player` property. And it’s why the buttons no longer work – they attempt to send a message to the `delegate`, and that’s now `nil`.

In order to fix this, `RatePlayerViewController` needs to tell the state preservation system that it wants to save the values of these two properties, so they can be restored later.

To do that, you need to implement two methods: `encodeRestorableStateWithCoder` and `decodeRestorableStateWithCoder`. Implement these methods in **RatePlayerViewController.m** as follows:

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
```

```
[super encodeRestorableStateWithCoder:coder];

[coder encodeObject:self.delegate forKey:@"Delegate"];
[coder encodeObject:self.player forKey:@"Player"];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    self.delegate = [coder decodeObjectForKey:@"Delegate"];
    self.player = [coder decodeObjectForKey:@"Player"];
}
```

In `encodeRestorableStateWithCoder:` you tell UIKit what values you want it to remember for this view controller. This method is called the very moment the app goes into the background. It uses the `NSCoder` system. You tell the `NSCoder` object to store the value of `self.delegate` under the key “Delegate” and the `player` object under the key “Player.”

The `decodeRestorableStateWithCoder:` method works in reverse. It is called when the app comes alive again and UIKit performs the state restoration step. You ask the coder for the object stored under the “Delegate” key and put it back in `self.delegate`. You do likewise for the `player` object.

These two methods are the meat and bones of the State Preservation and Restoration system. You set Restoration IDs on the view controllers that you want UIKit to preserve, and implement these two methods to save and restore any additional data that UIKit may not know about.

Note: You should always call `super` in these two methods. If you don’t, then your view controllers may not restore properly.

With the app still running, go back to the Players screen and tap on a new row to open the Rate Player screen again. Press the Home button, kill the app, and restart it.

After the app starts up, you should immediately be brought back to the Rate Player screen. For some reason, the title bar still says “Rate Player,” but try tapping one of the buttons. They should work, and the app will return to the main screen.

That means the `delegate` property got properly restored, so the encoding and decoding methods must be working – but why didn’t the title change?

Let’s take a closer look at **RatePlayerViewController.m**. The title is set in the `viewDidLoad` method:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.title = self.player.name;
}
```

Previously, this worked fine. When the segue takes place, the `player` property gets set in `prepareForSegue:`, which happens before the destination view controller's view is loaded. Afterwards, `viewDidLoad` is called and puts the player's name into `self.title`. No problems there.

During state restoration, however, the sequence of events is slightly different. The `decodeRestorableStateWithCoder:` method is called *after* `viewDidLoad`. It does restore the `Player` object and places it into the `self.player` property, but this is too late and the view controller title never gets set.

The solution is to change the `viewDidLoad` method into `viewWillAppear`:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    self.title = self.player.name;
}
```

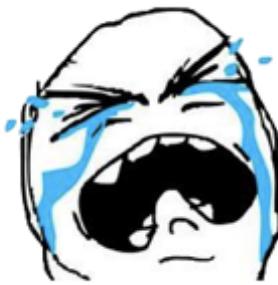
This method gets called after decoding, so by the time this happens, `self.player` does have a proper `Player` object in it.

Try it out. Press Home to go to the background, kill the app, and restart it. Now you should end up back on the Rate Player screen and the name of the player is once again in the title bar. Cool!

When a player is not a player

Now try this: tap on a player with a 5-star rating. Put the app in the background, kill it, and restart it. The app should show the Rate Player screen for that player again. Tap the 1-star button. The Rate Player screen pops off the navigation stack as it should and you end up back on the main screen.

But check this out: did the rating for the player you selected actually change from 5 stars to 1? Nope, it didn't.



Why didn't it update? Let's take a look at what's going on step by step.

Earlier you implemented the `encodeRestorableStateWithCoder:` method to put the player object into the `NSScoder`:

```
[coder encodeObject:self.player forKey:@"Player"];
```

But what actually happens when you do this? First of all, the object that you're encoding – `self.player` in this case – must implement the `NSCoding` protocol.

Let's check the `Player` class to see if it implements that. It does that indeed, as you can see in **Player.h**:

```
@interface Player : NSObject <NSCoding>
```

And **Player.m** implements those `NSCoding` methods:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super init]))
    {
        self.name = [aDecoder decodeObjectForKey:@"Name"];
        self.game = [aDecoder decodeObjectForKey:@"Game"];
        self.rating = [aDecoder decodeIntForKey:@"Rating"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
    [aCoder encodeObject:self.name forKey:@"Name"];
    [aCoder encodeObject:self.game forKey:@"Game"];
    [aCoder encodeInt:self.rating forKey:@"Rating"];
}
```

The `encodeWithCoder:` method puts the `Player` object's properties – name, game and rating – into the `NSCoder`, and `initWithCoder:` does the reverse by reading from the `NSCoder` and setting the properties.

Note: Wondering why the `Player` class implements `NSCoding` in the first place? That is how the app writes these objects to a "Players.plist" file when the app goes to the background, and how it reads them back in again when the app launches. It all happens in **AppDelegate.m** in the `loadPlayers` and `savePlayers` methods. You can look these up in the source code if you're curious.

So because `Player` already conforms to the `NSCoding` protocol, you were able to write the `player` object to the `NSCoder` in `encodeRestorableStateWithCoder:` and read it back in again with `decodeRestorableStateWithCoder:`.

But here's the important part: In `decodeRestorableStateWithCoder:`, when you do `[coder decodeObjectForKey:@"Player"]`, a *new* `Player` object gets allocated and its `initWithCoder:` method is called. That new `Player` object gets the same name, game and rating values as the `Player` that was encoded – but it is a completely different instance.

After the app is brought back to life and the state restoration process is done, the Rate Player screen has a `Player` object with the same property values as the one you originally picked, but it is actually a different object.

When you tap a star button, the delegate gets told that this new `Player` has a new score, but the delegate – in other words, the `PlayersViewController` – doesn't have this `Player` object in its data model, and as a result, nothing happens.

Note: In summary, the State Preservation and Restoration mechanism is not meant for storing and retrieving model data. You still do that inside your data model. In the Ratings app, the App Delegate is the owner of the data model, and therefore loading and saving the list of players happens in `AppDelegate.m`.

The State Preservation and Restoration mechanism is only intended for storing what the user was doing at the time s/he switched to some other app. You use it to keep track of the state of your view controllers; for example, which items are selected, what the user typed into the text field, and so on. These are not things you would normally keep in your data model, but want to preserve anyway.

The solution is to not encode and decode the entire `Player` object, but only a reference to that object, so that a pointer to the correct player can be obtained from the data model after the view controller has been restored.

Note: This does NOT mean you should store the object's actual pointer. When the app is restarted, all the objects will be new instances, so any pointers from the previous run of the app will no longer be valid. Instead, store a piece of data that can uniquely identify the object that you're trying to save.

Instead of storing the entire `Player` object, you give each player a unique ID and store just that ID.

Add the following property declaration to `Player.h`:

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

Augment the `initWithCoder:` and `encodeWithCoder:` methods in `Player.m`:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super init]))
    {
        . .
        self.playerID = [aDecoder decodeObjectForKey:@"ID"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
    . .
    [aCoder encodeObject:self.playerID forKey:@"ID"];
}
```

That's easy enough; the ID will simply be stored along with the other fields. Note that these two methods are just for loading and saving the `Player` objects to the `Players.plist` file that stores the data model – these methods are not used by the State Preservation and Restoration mechanism.

So where do you get this ID from? There are many ways to give your objects unique IDs. For example, if you're using Core Data, you could use `NSManagedObject`'s `objectID` property.

For this app, you'll use a UUID, a Universally Unique Identifier (also known as a GUID), which you can think of as a mostly random number that is practically guaranteed to be unique. iOS 6 added a handy new class, `NSUUID`, for making such identifiers, so you'll use that here.

Add a new `init` method to `Player.m`:

```
- (id)init
{
    if ((self = [super init]))
    {
        self.playerID = [[NSUUID UUID] UUIDString];
    }
    return self;
}
```

This assigns a new, unique ID to the `playerID` property when the `Player` object is created. (But not when the data model is loaded from the `Players.plist` file, in which case `initWithCoder:` is used instead.)

You're doing all of this so you can encode just this `playerID` property, rather than the entire `Player` object, in `RatePlayerViewController`.

Change the `encodeRestorableStateWithCoder:` method in **RatePlayerViewController.m** to:

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super encodeRestorableStateWithCoder:coder];

    [coder encodeObject:self.delegate forKey:@"Delegate"];
    [coder encodeObject:self.player.playerID forKey:@"PlayerID"];
}
```

That's all well and good, but how should you decode this? You could add the following line to `decodeRestorableStateWithCoder`, replacing the existing line that restores the full `Player` object:

```
NSString *playerID = [coder decodeObjectForKey:@"PlayerID"];
```

But that just gives you the player's ID, not the actual `Player` object. The big question is: how do you obtain the `Player` object using just that ID?

Unfortunately, the `RatePlayerViewController` class does not have access to the complete array of `Player` objects, or it could just step through that to look for a player with a matching ID. The only classes that do have the list of players are `PlayersViewController` and the `AppDelegate`. You're going to have to write some additional code to make this work.

For now, remove the line for decoding the player from `decodeRestorableStateWithCoder:` so that the method just reads:

```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];
```

```
    self.delegate = [coder decodeObjectForKey:@"Delegate"];
}
```

The birth of a view controller

Let's backtrack a little. How did UIKit know what to do in order to instantiate the Rate Player View Controller in the first place? This was the last screen that was active before the app went into the background, and it has a Restoration ID, so UIKit knows that it somehow needs to restore the screen with that ID. But how and where did UIKit create that `RatePlayerViewController` object?

When an app launches, it only loads the minimum required number of view controllers from the storyboard. The Ratings app first loads the Tab Bar Controller and the two navigation controllers for the tabs. A navigation controller by itself doesn't do anything, so the app also loads their root view controllers, the `PlayersViewController` and the `RankingsViewController`, respectively.

But there it stops. None of the other view controllers from the storyboard are needed at that point, so it's a waste of time and resources to load them.

With the state restoration mechanism, however, UIKit knows that the view controller with Restoration ID "RatePlayerViewController" was the last one active. It hasn't been loaded yet, so UIKit looks through the storyboard to see if any of the view controllers has a matching ID, then instantiates that view controller and pushes it on the navigation stack.

Therefore, in the state restoration phase, UIKit will search the storyboard to load the view controller objects that it needs to restore. This happens completely automatically – you don't have to do anything.

However, you can choose to take charge of this loading process, which gives you more control over how your view controllers are instantiated. There are several ways you can do this, but in this example you will add some code to the App Delegate.

Add an import in `AppDelegate.m`:

```
#import "RatePlayerViewController.h"
```

Add the following method:

```
- (Player *)playerWithID:(NSString *)playerID
{
    for (Player *player in _players)
    {
        if ([player.playerID isEqualToString:playerID])
            return player;
    }
}
```

```
    }
    return nil;
}
```

This simply loops through the `_players` array to look for a `Player` object with the specified ID. That's how you will get the `Player` object if you just have its ID.

The real magic happens in the following method – add it to **AppDelegate.m**:

```
- (UIViewController *)application:(UIApplication *)application
    viewControllerWithRestorationIdentifierPath:
        (NSArray *)identifierComponents
    coder:(NSCoder *)coder
{
    NSLog(@"viewControllerWithRestorationIdentifierPath %@", identifierComponents);

    UIViewController *viewController = nil;
    NSString *identifier = [identifierComponents lastObject];

    if ([identifier isEqualToString:@"RatePlayerViewController"])
    {
        UIStoryboard *storyboard = [coder decodeObjectForKey:
            UIStateRestorationViewControllerStoryboardKey];
        if (storyboard != nil)
        {
            viewController = [storyboard
                instantiateViewControllerWithIdentifier:identifier];

            if (viewController != nil)
            {
                RatePlayerViewController *ratePlayerViewController =
                    (RatePlayerViewController *)viewController;

                NSString *playerID = [coder
                    decodeObjectForKey:@"PlayerID"];
                if (playerID != nil)
                {
                    ratePlayerViewController.player = [self
                        playerWithID:playerID];
                }
            }
        }
    }

    return viewController;
}
```

```
}
```

This method is called by the state restoration mechanism for every single view controller that has its Restoration ID set. The `identifierComponents` parameter is an array that contains the so-called *restoration path*, which is the sequence of restoration identifiers from the root view controller – in our case the Tab Bar Controller – all the way up to the view controller to restore.

Let's see how this works. There is an `NSLog()` statement in the method that will dump the restoration path to the Xcode output pane.

Open the app and tap on a player to open the Rate Player screen. Click Home to move the app into the background, and then stop it from within Xcode. Run the app again and keep an eye on the output pane. It should say something like this:

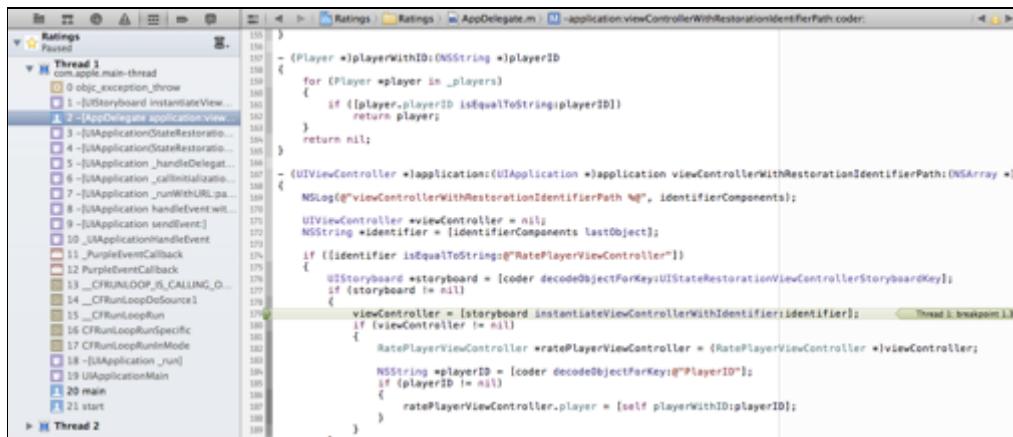
```
2012-08-01 14:33:05.099 Ratings[67088:11303]
viewControllerWithRestorationIdentifierPath (
    "Root Tab Bar"
)
2012-08-01 14:33:05.101 Ratings[67088:11303]
viewControllerWithRestorationIdentifierPath (
    "Root Tab Bar",
    "Players Navigation"
)
2012-08-01 14:33:05.103 Ratings[67088:11303]
viewControllerWithRestorationIdentifierPath (
    "Root Tab Bar",
    "Rankings Navigation"
)
2012-08-01 14:33:05.104 Ratings[67088:11303]
viewControllerWithRestorationIdentifierPath (
    "Root Tab Bar",
    "Players Navigation",
    PlayersViewController
)
2012-08-01 14:33:05.107 Ratings[67088:11303]
viewControllerWithRestorationIdentifierPath (
    "Root Tab Bar",
    "Players Navigation",
    RatePlayerViewController
)
```

The `application:viewControllerWithRestorationIdentifierPath:coder:` method gets called five times. The first time it is for the Tab Bar Controller, which has Restoration ID "Root Tab Bar." You don't have to do anything for this view controller because it gets loaded from the storyboard automatically.

The second time the method gets called it is for the navigation controller from the Players tab, and the third time it is for navigation controller from the Rankings tab. The fourth call is for restoring the `PlayersViewController`, and the fifth for restoring the `RatePlayerViewController`.

You can see that for each view controller, the restoration path tells you where it lives inside any other view controllers; in other words, it lists all the parent controllers. That is useful for when you use the same view controller class more than once in your storyboard and you need to figure out which one it is that you are asked to restore.

You may also have noticed that the app has crashed at this point in the method:



To understand why, I should explain what this method is doing. Here is the relevant code again:

```

NSString *identifier = [identifierComponents lastObject]; // 1

if ([identifier isEqualToString:
    @"RatePlayerViewController"])
{
    UIStoryboard *storyboard = [coder decodeObjectForKey:
        UIStateRestorationViewControllerStoryboardKey]; // 3

    if (storyboard != nil)
    {
        viewController = [storyboard
            instantiateViewControllerWithIdentifier:identifier];
    }

    if (viewController != nil)
    {
        RatePlayerViewController *ratePlayerViewController =
            (RatePlayerViewController *)viewController;
    }
}

NSString *playerID = [coder

```

```
        decodeObjectForKey:@"PlayerID"]; // 5
    if (playerID != nil)
    {
        ratePlayerViewController.player = [self
                                           playerWithID:playerID]; // 6
    }
}
}
```

Let's take this step-by-step:

1. As you have seen, the `identifierComponents` array contains the Restoration ID for the view controller that should be restored, as well as the IDs for its parent view controllers. Usually, you're only interested in the ID for the view controller to restore, so you can use `lastObject` to obtain that.
2. If that Restoration ID is "RatePlayerViewController," then you're good to go.
3. The purpose of this whole method is to create the view controller object. Instead of letting UIKit search the storyboard for it, you will do this yourself so you can set a property on the view controller. As a convenience, the `NSCoder` object contains a reference to the storyboard that you can obtain by asking for the special `UIStateRestorationViewControllerStoryboardKey`.
4. Ask the storyboard to give you the view controller with the specified identifier.
5. Look into the `NSCoder` object for the "PlayerID" key. This is the ID that was set in the new version of `RatePlayerViewController`'s `encodeRestorableStateWithCoder:`.
6. Once you have the player ID, look up the `Player` object using the `playerWithID:` method that you just added, and then set the `RatePlayerViewController` object's `player` property.

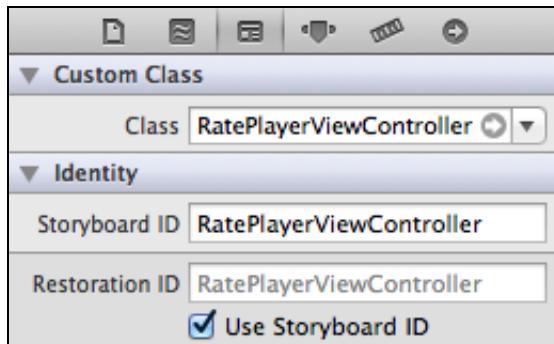
To summarize, instead of letting UIKit look up and instantiate the Rate Player View Controller automatically, you do it yourself. It's a bit more work, but it gives you the opportunity to set that view controller's `player` property, because at this point in the app you do have the full data model at your disposal.

This all looks good in theory, but why does the app crash? It bombs out at step 4, where it does `[storyboard instantiateViewControllerWithIdentifier:identifier]`.

The problem is that "identifier" here does not mean the Restoration ID, but the Storyboard ID. Each view controller can have multiple identifiers: the Restoration ID is used by the state restoration mechanism, but the Storyboard ID is for manually instantiating view controllers from a storyboard. They're getting mixed up here.

Go to the storyboard, select the Rate Player View Controller and open the Identity inspector. Fill in the Storyboard ID field and check the "Use Storyboard ID" box.

This gives the view controller identical values for the Restoration ID and Storyboard ID attributes:



You're not required to make these two IDs identical, but you might as well.

Note: At this point, it's a good idea to remove the Players.plist file before you run the app again. Because Player.m now also stores the `playerID` string in the PLIST file, the format of the data has changed. If you don't delete the Players.plist file, all existing `Player` objects will have a `nil` ID because they have been saved using the old format, and this may cause weird behavior in the app.

To delete Players.plist, you can completely remove the app from the phone. If you have been testing on the Simulator, you can also reset the Simulator (with iOS Simulator\Reset Content and Settings), or remove the PLIST file from the app's sandbox, which is located in your Library folder under Application Support/iPhone Simulator/6.0/Applications/<id>/Documents/.

Run the app again and tap a row to open the Rate Player screen. Put the app into the background and kill the app from Xcode.

Run it again and... still no dice. In fact, the app seems to have regressed: in the title bar says "Rate Player" again, instead of the player's name.

Let's see what's going on. Go back to **AppDelegate.m** and put a breakpoint on the line:

```
ratePlayerViewController.player = [self playerWithID:playerID];
```

Run the app again. The debugger will now hit the breakpoint as UIKit tries to restore the Rate Player screen:

The screenshot shows the Xcode debugger interface during the execution of the `-[AppDelegate application:viewControllerWithRestorationIdentifierPath:coder:]` method. The code being executed is:

```

179     viewController = [storyboard instantiateViewControllerWithIdentifier:identifier];
180     if (viewController != nil)
181     {
182         RatePlayerViewController *ratePlayerViewController = (RatePlayerViewController *)viewController;
183
184         NSString *playerID = [coder decodeObjectForKey:@"PlayerID"];
185         if (playerID != nil)
186         {
187             ratePlayerViewController.player = [self playerWithID:playerID]; // Thread 1: breakpoint 2.1
188         }
189     }
190 }
191
192 return viewController;
193
194
195 @end
196
197

```

The Local variable inspector shows the following state:

- `_cmd = (SEL) application:viewControllerWithRestorationIdentifierPath:coder: (lldb)`
- `application = (UIApplication *) 0x0887e480`
- `identifierComponents = (_NSArrayM *) 0...`
- `coder = (UIStateRestorationKeyedUnarchiv...`
- `identifier = (_NSCFString *) 0x0759dd40...`
- `playerID = (_NSCFString *) 0x0759eeb0...`
- `ratePlayerViewController = (RatePlayerVi...`
- `storyboard = (UIStoryboard *) 0x0759bd00`
- `viewController = (RatePlayerViewControll...`

The All Output pane shows the restoration identifier path:

```

viewControllerWithRestorationIdentifierPath (
    "Root Tab Bar",
    "Players Navigation",
    PlayersViewController
)
2012-08-01 15:05:16.330 Ratings[67410:11303]
viewControllerWithRestorationIdentifierPath (
    "Root Tab Bar",
    "Players Navigation",
    RatePlayerViewController
)
(lldb)

```

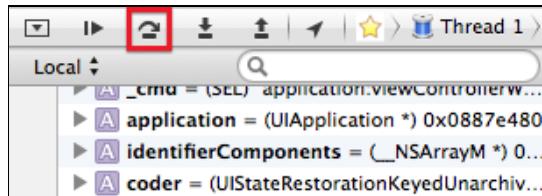
Everything went fine up to this point. The app created the new view controller instance and obtained the `playerID` from the coder.

Type the command “`po playerID`” behind the `(lldb)` prompt in the debugger window:

```
(lldb) po playerID
(NSString *) $1 = 0x0759eeb0 2AAFBB37-830F-46AE-A8C5-02712341D790
```

That's good news. The `playerID` has a valid UUID string with the value `2AAFBB37-830F-46AE-A8C5-02712341D790`. Of course, for you the ID will be different.

Click the Step Over button from the debugger toolbar to execute the next line of code:



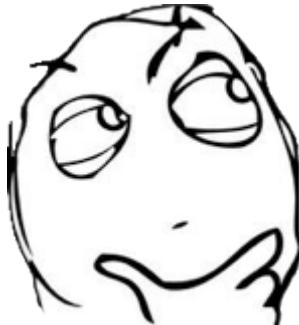
Now type:

```
(lldb) po ratePlayerViewController.player
(Player *) $2 = 0x00000000 <nil>
```

Well, that's no good. The `player` property is `nil`, which means that the `playerWithID:` method returned `nil`. Either there is no `Player` object with that ID or – hey, wait a minute, there are no `Player` objects at all:

```
(lldb) po _players
(NSMutableArray *) $3 = 0x00000000 <nil>
```

Yikes! Why is the `_players` array empty at this point?



If you dig through the code in **AppDelegate.m**, you'll find that the `_players` instance variable is initialized in the `loadPlayers` method, which in turn is called from `application:didFinishLaunchingWithOptions:`. Here's the rub: the "did Finish Launching" method gets called *after* the state restoration process is complete. Obviously, you need to create the `_players` array earlier on in the startup sequence.

To solve this problem, iOS 6 adds a new method to `UIApplicationDelegate`, called `willFinishLaunching`. Change the code in **AppDelegate.m** to:

```
- (BOOL)application:(UIApplication *)application
willFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [self loadPlayers];

    self.playersViewController.players = _players;
    self.rankingsViewController.players = _players;

    [self.window makeKeyAndVisible];
    return YES;
}

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    return YES;
}
```

You have moved all the initialization code into `willFinishLaunching`, which gets called *before* the state restoration process happens. The `didFinishLaunching` method now just returns `YES`.

Note: Apple recommends that you make the app's window visible before the state restoration phase takes place. You need to do this in your app delegate's `willFinishLaunching` method by calling `[self.window makeKeyAndVisible]`.

In a storyboard-based app, it never used to be necessary to make the window explicitly visible, but with State Preservation and Restoration enabled it is best not to skip this step, otherwise certain views may not restore properly.

Remove the breakpoint and run the app again. Now the whole caboodle should work. If you change the rating, the correct `Player` object should update. Phew!

Keys and consts

There is still a small improvement that can be made to the coding style. Until now, you've hardcoded the `NSCoder` key names using `NSStrings` such as `@"PlayerID"`. As a general rule, it is better to use `#defines` or `consts` for this.

Add this line to **RatePlayerViewController.h**, outside the `@interface` section:

```
extern NSString * const PlayerIDKey;
```

And these lines to **RatePlayerViewController.m**, above the `@implementation` section:

```
NSString * const PlayerIDKey = @"PlayerID";
static NSString * const DelegateKey = @"Delegate";
```

Note that the second declaration is `static` because it is limited to this file only. The first one also needs to be used from `AppDelegate`, so that's why it is declared as `extern` in the header file.

Replace the hardcoded keys in `encodeRestorableStateWithCoder:` and `decodeRestorableStateWithCoder:` with these `consts`:

```
[coder encodeObject:self.delegate forKey:DelegateKey];
[coder encodeObject:self.player.playerID
               forKey:PlayerIDKey];

. . .

self.delegate = [coder decodeObjectForKey:DelegateKey];
```

And in **AppDelegate.m**:

```
NSString *playerID = [coder decodeObjectForKey:PlayerIDKey];
```

Run the app again to verify that it works.

Note: If you think this approach of creating the `RatePlayerViewController` from within the App Delegate is ugly, then as an alternative approach, you can change the view controller to always use player IDs instead of real `Player` objects.

Instead of saying, "I changed this Player's rating," the delegate method would say "I changed the rating for the player with this ID." The delegate is responsible for changing the actual `Player` object. That way App Delegate doesn't have to know anything about `RatePlayerViewController`.

Later on in the chapter you will see another approach, "Restoration Classes."

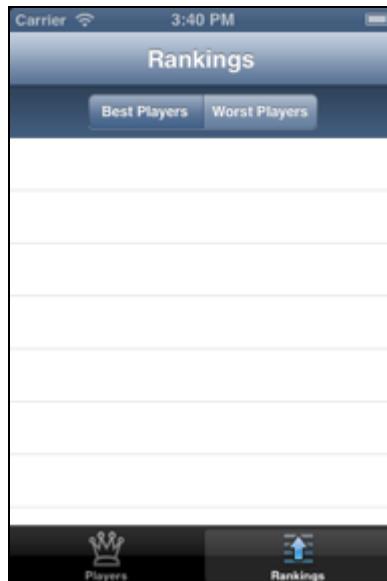
The Rankings screen

You can also open the Rate Player screen from the Rankings tab. Run the app, switch to the Rankings tab and tap on a name from the list.



Just like before, this adds a new `RatePlayerViewController` instance on the navigation stack, except this time it's the navigation stack that belongs to the Rankings tab (you can see that in the text on the back button).

Tap the Home button, kill the app from Xcode, and restart it. The Rate Player screen should restore OK, but when you tap a star button or the back button, the Rankings screen appears empty:



The table view doesn't reload until you tap on Best Players or Worst Players.

As it turns out, this is a subtle bug in the code. The `RankingsViewController` sets a boolean flag, `_needsUpdate`, when the table view needs to reload completely. When that flag is YES, `viewWillAppear:` calls the `updateRankedPlayers` method, and that fills up the `_rankedPlayers` array and the table view. Here is the relevant code:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    if (_needsUpdate)
        [self updateRankedPlayers];

    _needsUpdate = NO;
}
```

However, the `_needsUpdate` flag is set to `NO` when the Rate Player View Controller's delegate method is invoked to indicate to the delegate that a player rating was changed. In that case, you don't want to reload the entire table, but only insert or delete a single row with a nice animation. Calling `reloadData` on the table view would be overkill, and `viewWillAppear:` now skips the call to `updateRankedPlayers`. Take a peek at `RankingsViewController.m` to see how this logic works (it's not as hard as it sounds).

Now that you have added state restoration to the app, the sequence of events is slightly different. Because UIKit doesn't display the Rankings screen before it goes to the Rate Player screen, `viewWillAppear:` is never called with `_needsUpdate` set to `YES`, the `_rankedPlayers` array remains `nil`, and the table view never gets filled up. Still with me? Good. ☺

To fix this, change `RankingsViewController`'s `viewWillAppear:` to:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    if (_needsUpdate || _rankedPlayers == nil)
        [self updateRankedPlayers];

    _needsUpdate = YES;
}
```

Now it should work. The `updateRankedPlayers` method also gets called if the list of ranked players is still empty.

The reason I included this example is to illustrate that you may need to tweak your refreshing logic when switching to State Preservation and Restoration! If the normal order of events is that view controller A gets loaded and displayed before view controller B, with state restoration it can happen that B is displayed before A. Your code should be able to deal with both situations!

Note: What happens when you have the Rate Player screen open in both tabs at the same time? Try it out: go to the Players screen and tap a row, then go to the Rankings screen and also tap a row there. Both tabs now should have the Rate Player screen open.

Put the app in the background, and kill and restart it. Both of the Rate Player screens should restore without any problems. These are two different instances of `RatePlayerViewController`; you can see this in Xcode's output pane. And if you selected the same player both times, then both these view controllers refer to the very same `Player` object.

The segmented control

The Rankings screen has a segmented control that lets you flip between 5-star players and 1-star players:



However, UIKit doesn't automatically remember the state of this control. Try it out. The screen always switches back to "Best Players."

Certain UIKit controls do remember at least some of their state information, but `UISegmentedControl` isn't one of them. Giving this control its own Restoration ID accomplishes nothing.

Note: The standard UIKit views that do save state information are:

`UICollectionView`, `UIImageView`, `UIScrollView`, `UITableView`, `UITextField`, `UITextView`, and `UIWebView`. Refer to their Class Reference documentation to see exactly what sort of state data they store.

So if UIKit doesn't automatically do this, you'll just have to do it yourself. Add the following line to `RankingsViewController.m`, above the `@implementation` section:

```
static NSString * const RequiredRatingKey = @"RequiredRating";
```

And add the methods for encoding and decoding:

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super encodeRestorableStateWithCoder:coder];

    [coder encodeInt:_requiredRating forKey:RequiredRatingKey];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    _requiredRating = [coder decodeIntForKey:RequiredRatingKey];
    self.segmentedControl.selectedSegmentIndex =
        (_requiredRating == 5) ? 0 : 1;
}
```

Encoding is pretty simple: you just store the value of the `_requiredRating` instance variable into the `NSCoder`.

If you look at the `segmentedControlChanged:` action method, you'll see that it sets `_requiredRating` to 5 if the first segment ("Best Players") is selected, and to 1 if the second segment ("Worst Players") is selected. This number, of course, refers to the number of stars that the players have. The `updateRankedPlayers` method uses this number to filter the list of players.

Conversely, the decoding method reads the value of `_requiredRating` from the `NSCoder` and uses it to select the correct segment in the segmented control.

If the app is still open at this point, switch to the Rankings screen and select the Worst Players segment. Put the app in the background, kill it, and restart. Whoops, the Rankings screen is empty again. What gives?

Here's what happened: when you put the app into the background, you hadn't compiled the new encoding method yet. But when you ran the app again, it got recompiled first, and `decodeRestorableStateWithCoder:` tried to look up the value under the `RequiredRatingKey`. There was no such value because the app never encoded it.

If `NSCoder` cannot find the key given in `decodeIntForKey:` it returns the value 0, and as a result `_requiredRating` gets set to 0. There are no players with zero stars, and the list is empty.

This situation is a little contrived and wouldn't (or shouldn't!) happen in practice, although it can occur during development if you're not paying attention. It's good to protect against such situations – and unnecessary confusion – by adding in some defensive programming. Replace the decoding method with:

```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    _requiredRating = [coder decodeIntForKey:RequiredRatingKey];
    if (_requiredRating < 1 || _requiredRating > 5)
        _requiredRating = 5;

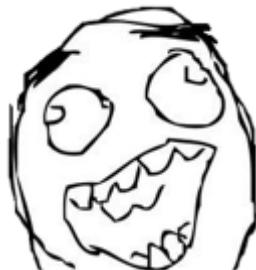
    self.segmentedControl.selectedIndex =
        (_requiredRating == 5) ? 0 : 1;
}
```

This way you'll never set `_requiredRating` to an invalid value. You don't have to be too paranoid with this sort of thing, but building in basic protections such as these can save you – and your users – from headaches down the road.

Without putting the app in the background, kill the app and run it again. Now the Ranking screen should switch back to the "Best Players" segment. The `NSCoder` still

doesn't have the "RequiredRating" key in it (because you killed the app without giving the state preservation system a chance to do its thing), but at least the app falls back to something that works – it no longer shows the empty list.

Just to make sure everything works as it should, select "Worst Players," put the app in the background, kill it, and restart it. Does it restore to the list of Worst Players? Awesome!



IT WORKS!

The Add Player screen

The Add Player screen is a bit different from the ones you've seen so far because it does not get pushed on the navigation stack – instead, it gets presented modally on top of everything else.

To the State Preservation and Restoration mechanism, however, that makes no difference. It will automatically figure out that this view controller was modally presented and restore it that way. At least, if you give it a Restoration ID.

Open the storyboard and give the Add Player screen a Restoration ID. As before, just name it after the view controller, "PlayerDetailsViewController." Don't forget to also give its Navigation Controller a Restoration ID. I suggest you name it "Player Details Navigation."



Run the app, and click the + button inside the Players tab to open the Add Player screen:



You know the drill: tap the Home button to close the app, kill the app from Xcode, and run it again.

The app should indeed restore the Add Player screen, but the Cancel and Done buttons no longer work. That should bring about a case of déjà vu, because something similar happened on the Rate Player screen.

Indeed, the delegate has gone missing. That's an easy fix because you already know how to do it. Add the following line at the top of **PlayerDetailsViewController.m**:

```
static NSString * const DelegateKey = @"Delegate";
```

And add the following methods:

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super encodeRestorableStateWithCoder:coder];

    [coder encodeObject:self.delegate forKey:DelegateKey];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    self.delegate = [coder decodeObjectForKey:DelegateKey];
}
```

Note that you're using the same key here, @"Delegate", that you also used to encode the delegate in the Rate Player View Controller. That's OK. Each view controller gets its own `NSCoder` object, so there is no chance of naming conflicts. You don't need prefixes to disambiguate between different namespaces.

Note: Do you recall what I pointed out earlier – that you should never store the pointer to an object inside the `NSCoder`? After all, you cannot persist pointers across application invocations. But `self.delegate` obviously contains a pointer to another view controller, so how would that ever work?

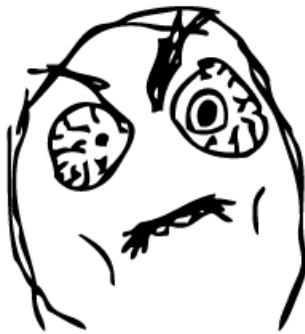
When you encode views or view controllers, UIKit does something special behind the scenes. It doesn't store a direct pointer to the object, but the object's Restoration ID. In the decoding stage, it looks up the object by that Restoration ID and returns the new pointer.

So it's safe to put references to any other view controllers inside the `NSCoder`, provided they have their Restoration IDs filled in.

Kill the app and re-launch it. Bummer dude, it still doesn't work. The reason should be obvious: the app hasn't yet had a chance to properly encode the delegate.

But even putting the app in the background first before you killed it wouldn't have solved anything – the particular instance of the Add Player screen that you're looking at will always have a delegate that is `nil`, no matter how many times you restart the app.

Now what? The app is stuck on a malfunctioning Add Player screen and there is no way to get out of there. The only thing you can do is throw away the saved restoration data. Or, delete the app and install again, so that fresh restoration data is saved.



First kill the app. Then locate the folder where the app stores the restoration data and remove it. You can find this inside your Library folder. If you're on Lion or Mountain Lion, then your Library folder may be hidden from Finder, but you can still access it through the Terminal, or unhide it (Google for instructions).

On my machine, the folder is:

```
~matthijs/Library/Application Support/iPhone Simulator/6.0/Applications/  
761269F2-0DF6-4735-88B0-DB1B696246FE/Library/Saved Application  
State/com.yourname.Ratings.savedState/
```

That folder contains a file named “data.data.” This is actually a binary PLIST file. If you’re curious, open the file in an editor that can read binary PLIST files (such as TextWrangler) and take a peek inside, although it’s not really meant to be human-readable.

Remove this “data.data” file and restart the app. Now there is nothing for it to restore (you can see that there is nothing being printed in the Xcode output pane now), and the app opens up as if it were launched for the very first time.

Note: If you’re testing on the device and you get stuck in a similar situation, it’s simplest to just delete the app.

Alternatively, make the app crash. When the app crashes during startup, UIKit will remove the restoration data files, so that during the next launch cycle your app will not try to restore anything. If UIKit didn’t do that, users could get into an infinite loop and never get your app working again. That won’t get you many 5-star ratings on the App Store!

Repeat the Add Player test: press the + button, close app, kill app, run app. Now it works perfectly because the Add Player screen remembers who its delegate is.

Picking the game

Well, maybe not perfectly... The Add Player screen no longer gets stuck, but it doesn’t remember what you typed into the Player Name text field, or which game you chose. Try it out: type something and pick a game, and see if it restores any of that. Nope, it doesn’t.

Add a new key definition to **PlayerDetailsViewController.m**:

```
static NSString * const GameKey = @ "Game" ;
```

Add the following line to `encodeRestorableStateWithCoder`:

```
[coder encodeObject:_game forKey:GameKey];
```

And this line to `decodeRestorableStateWithCoder`:

```
_game = [coder decodeObjectForKey:GameKey];
```

It looks like this would be enough to remember the chosen game, but it isn’t. Not only does the view controller need to remember the name of the game, it also needs to display this text in the cell’s detail label. The logic for that currently

resides in `viewDidLoad`, but that method is called *before* `decodeRestorableStateWithCoder:` happens.

The fix to this Catch-22 is as follows:

```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    self.delegate = [coder decodeObjectForKey:DelegateKey];

    _game = [coder decodeObjectForKey:GameKey];
    if (_game == nil || [_game length] == 0)
        _game = @"Chess";

    self.detailLabel.text = _game;
}
```

You fetch the game string from the `NSCoder`, make sure it has a valid value, and then update the table view cell's label. That's perfectly OK to do from this method, because the view and everything in it has already been loaded (but is not visible yet).

Try it out. Add a new player, pick Hearts as the game, and put the app to sleep. Kill and restart it. The game label should still say "Hearts," and when you press the Done button to add the player, the new Player object should truly have "Hearts" set as well.

It would also have worked fine to use the previous solution of moving the code and updating the detail label from `viewDidLoad` to `viewWillAppear:`. But it's always good to have more than one tool in your toolbox, right? ☺

The Player Name text field

Earlier I mentioned that `UITextField` is one of a handful of `UIKit` classes that stores state information if you give it a Restoration ID. You would think that it stores the actual text that the user has entered, but that is not the case. It only remembers the current selection, not the text itself.

The idea is that views should not store data that they can easily get from the view controller. Instead, the view controller either preserves that data or gets it from the data model. The view would only store data that is intrinsic to that view, such as the selection state of the text in a `UITextField`, or the scroll position in a `UIScrollView`. The view controller typically doesn't know or care about such things, but you want to store them anyway.

So you're on your own for preserving and restoring the text field's contents.

Add a new constant to `PlayerDetailsViewController.m`:

```
static NSString * const PlayerNameKey = @"PlayerName";
```

Add the following line to the encoding method:

```
[coder encodeObject:self.nameTextField.text
            forKey:PlayerNameKey];
```

And add this snippet to the decoding method:

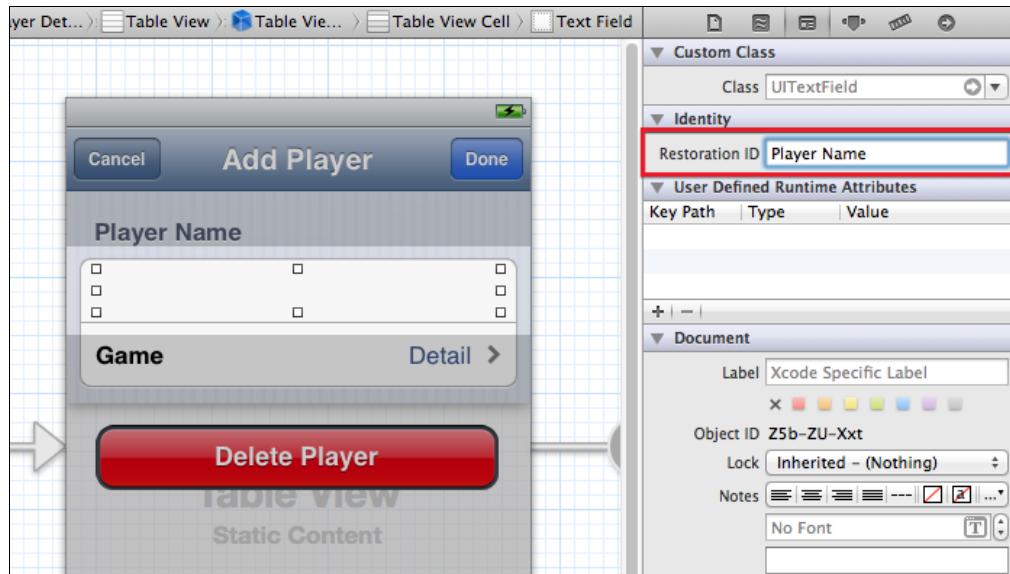
```
NSString *playerName = [coder
                        decodeObjectForKey:PlayerNameKey];

if (playerName != nil)
    self.nameTextField.text = playerName;
```

Try it out. That's a lot better already – the user no longer loses what they typed in.

However, if the keyboard was active, it is now gone. Not a big deal, but it would be nice if the app would remember that, too. You can program the solution yourself, but it turns out this is part of the state information that `UITextField` does remember. Oh happy day!

Open the storyboard, go to the Add Player scene, and select the text field in the Player Name cell. In the Identity inspector, fill in the Restoration ID field. It doesn't really matter what you set it to – I used "Player Name" – as long as all the views inside the same view controller have unique IDs.



In theory that's all you should have to do. The text field now ought to remember whether the keyboard was visible or not, as well as any selection you have made:



Unfortunately, it doesn't work. Run the app and try it out. The keyboard doesn't reappear and the selection is forgotten. This had me stumped because it works fine if the `UITextField` is not inside a table view cell. Luckily, the helpful engineers at Apple found a solution. Add the following line at the bottom of the `decode` method:

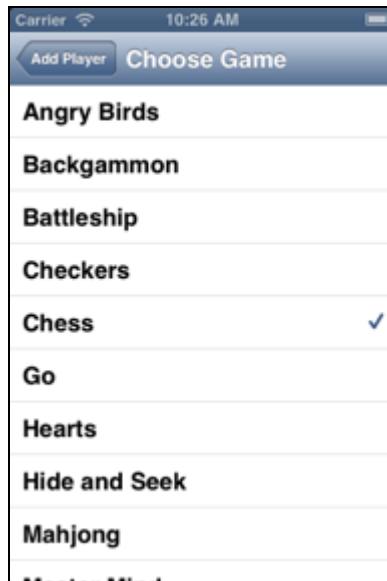
```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    ...
    [self.view layoutIfNeeded];
}
```

This forces the view to re-do its layout, which puts the cells into the view hierarchy, so that state restoration can actually find the “first responder” – the text field – and make the keyboard active again. Run the app and now it should work!

The Choose Game screen

You may have noticed we're just going through all the screens in the app to see if they do what they should when state restoration is enabled. The next one on the list is the Choose Game screen.

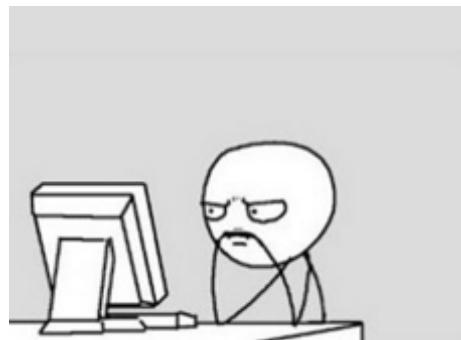
As always, first set its Restoration ID in the storyboard (“GamePickerController”).



Run the app, add a new player, click the Game row to open the Choose Game screen.

Close the app, kill it good, and restart it. You'll notice that the checkmark disappears, and tapping a row no longer returns you to the previous screen. That's an obvious case of a delegate gone missing.

See if you can fix these problems by yourself – I'll wait right here.



Done? OK, here's what I came up with:

Add these constants to **GamePickerController.m**:

```
static NSString * const DelegateKey = @"Delegate";
static NSString * const SelectedIndexKey = @"SelectedIndex";
```

And add encoding and decoding methods:

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super encodeRestorableStateWithCoder:coder];
```

```
[coder encodeObject:self.delegate forKey:DelegateKey];
[coder encodeInteger:_selectedIndex
                     forKey:SelectedIndexKey];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    self.delegate = [coder decodeObjectForKey:DelegateKey];
    _selectedIndex = [coder
                     decodeIntegerForKey:SelectedIndexKey];
}
```

Notice that this time we use `encodeInteger:` rather than `encodeInt:`. The type of `_selectedIndex` is `NSInteger`, not `int`. It's a small but subtle difference.

This should be enough effort on our part, because the table view data source methods already look at the value of `_selectedIndex` to set the checkmark, so you don't have to do that anywhere else.

Run the app and try it out.

There's an improvement you can make here, though. The list of games is pretty long. What if the user had scrolled down to pick Yahtzee? When they come back to the app, they will have to scroll all the way down again.

It would be better if the table view remembered its scroll position. In fact, that's exactly what it does when you give it a Restoration ID.

Open the storyboard, select the table view inside the Choose Game screen and give it a Restoration ID. I chose "Game List," but anything will do.

Try it out. Run the app and scroll the game list all the way to the bottom. Close, kill, restart, and... no such luck. It turns out there is a bug in iOS 6.0 that prevents this from working in a storyboard, although it works fine in a nib-based app.

Fortunately, there is a workaround. Change the decoding method to:

```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    self.delegate = [coder decodeObjectForKey:DelegateKey];
    _selectedIndex = [coder
                     decodeIntegerForKey:SelectedIndexKey];
```

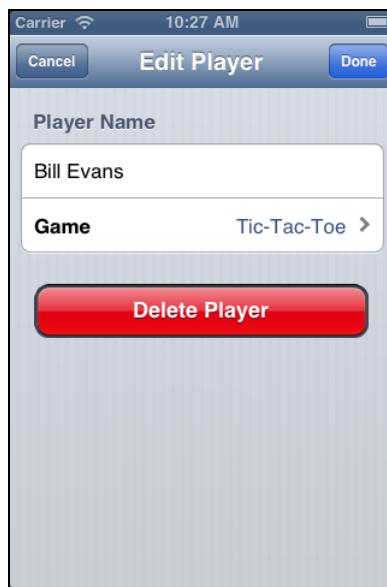
```
[self.tableView reloadData];  
}
```

Reloading the table view ensures that its content size is known, and that it can scroll to the proper location. Try it again, and now the game list should be exactly how and where you left it.

The Edit Player screen

Adding a player works fine, but what about editing a player? Both screens use the same view controller class, `PlayerDetailsViewController`, so you'd expect it to restore properly, especially since you've already made a bunch of changes to it. The only difference is that in edit mode, the view controller's `playerToEdit` property is set.

Run the app and click the blue detail disclosure button to edit a player:



Put the app into the background, kill it from Xcode, and re-launch it. Oddly enough, the screen reappears as the Add Player version. This view controller currently only preserves the text from the text field and the selected game, but nothing more. It does not remember that it is supposed to edit an existing `Player` object.

You have dealt with this problem before when restoring the Rate Player screen. It was no good to put a copy of the `Player` object into the `NSCoder`, because that gave you back a different `Player` object when the app was restored. Instead, you stored the player's ID. That's exactly what you'll do here.

Add to `PlayerDetailsViewController.m`:

```
static NSString * const PlayerIDKey = @"PlayerID";
```

And in the encoding method:

```
[coder encodeObject:self.playerToEdit.playerID  
    forKey:PlayerIDKey];
```

Now you face the same issue as before: having the player's ID is great, but at some point you will have to fetch the actual `Player` object from somewhere. This view controller does not have the complete list of player objects, so what do you do?

It's possible to move the loading code for this view controller into the App Delegate, just like you did for the Rate Player screen, but here I want to show you another solution: using a *restoration class*.

Remember how in App Delegate the method `application:viewControllerWithRestorationIdentifierPath:coder:` was invoked for each view controller that needed to be restored? With a restoration class, the process is similar, except the message gets sent to that restoration class instead of the app delegate.

Usually it is easiest to make the view controller its own restoration class, and that is what you will do here. First, you need to make the view controller conform to the `UIViewControllerAnimatedRestoration` protocol.

In `PlayerDetailsViewController.h`, change the `@interface` line to:

```
@interface PlayerDetailsViewController : UITableViewController  
    <UIViewControllerAnimatedRestoration>
```

You also need to set the `self.restorationClass` property on the view controller. When the object is preserved (as the app goes into the background), its restoration class is saved along with it. Upon restoration, the restoration class is asked to create a new instance of the view controller.

Unfortunately, there is no way to set the restoration class from within Interface Builder, so you have to do that in code. A good place to set this property is in `awakeFromNib`:

```
- (void)awakeFromNib  
{  
    [super awakeFromNib];  
    self.restorationClass = [self class];  
}
```

The `UIViewControllerAnimatedRestoration` protocol only has a single method in it. It is very similar to the one from App Delegate, except that this one is + because it is a class method.

Add it to `PlayerDetailsViewController.m`:

```
+ (UIViewController *)
```

```

viewControllerWithRestorationIdentifierPath:
    (NSArray *)identifierComponents
coder:(NSCoder *)coder
{
    PlayerDetailsViewController *viewController = nil;
    NSString *identifier = [identifierComponents lastObject];

    UIStoryboard *storyboard = [coder decodeObjectForKey:
        UIStateRestorationViewControllerStoryboardKey];

    if (storyboard != nil)
    {
        viewController = [storyboard
            instantiateViewControllerWithIdentifier:identifier];

        if (viewController != nil)
        {
            NSString *playerID = [coder
                decodeObjectForKey:PlayerIDKey];
            if (playerID != nil)
            {
                viewController.playerToEdit = ???;
            }
        }
    }

    return viewController;
}

```

Most of this you have seen before. You ask the storyboard to instantiate the view controller and you get the player ID from the NSCoder.

Note that here you don't have to check whether the Restoration ID is really "PlayerDetailsViewController" because this restoration class is only used in combination with this particular view controller, so that will never go wrong.

However, what do you do here?

```
viewController.playerToEdit = ???;
```

This piece of code needs to get a `Player` object from somewhere with the value of the `playerID` variable, but the problem is that here you are inside a class method (+) with no references to any of the other objects.

One approach is the following. The App Delegate is the owner of the data model (the array of `Player` objects), and it already has a `playerWithID:` method, except

that this method is currently private. But if you put its method signature in App Delegate's public @interface, you can call it from anywhere in your app.

Change **AppDelegate.h** to:

```
@class Player;

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

- (Player *)playerWithID:(NSString *)playerID;

@end
```

Now you can turn the missing bit for `viewControllerWithRestorationIdentifierPath:coder:` in **PlayerDetailsViewController.m** into:

```
AppDelegate *appDelegate = (AppDelegate *)[[UIApplication
                                         sharedApplication] delegate];
viewController.playerToEdit = [appDelegate
                               playerWithID:playerID];
```

To make this work, you also need to add an import:

```
#import "AppDelegate.h"
```

Before you run the app, don't forget to also set the Storyboard ID for the "Add Player" view controller, because you're now instantiating it from the storyboard. Without that Storyboard ID set, the app will crash when it tries to instantiate the view controller programmatically.

Try it out. The Edit Player screen should make an uneventful comeback. It still looks the same and even the Delete button is there.

Note that showing the Delete button is done inside `viewDidLoad`, which happens before decoding, but obviously after the restoration class has done its work. Because you set the `playerToEdit` property *before* `viewDidLoad` happened, no changes need to be made to the code for this to work.

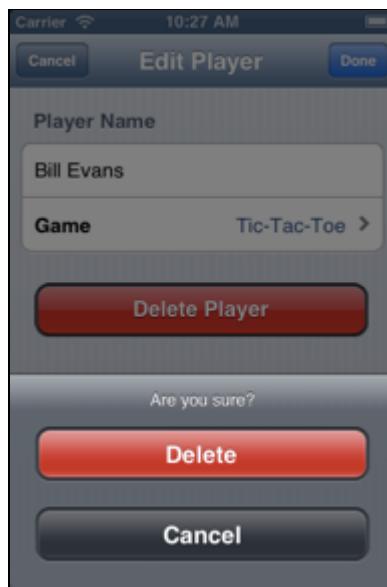
If you want to be thorough, also test this with the Choose Game screen open on top of the Edit Player screen.

Note: If you use a restoration class, there are two places where you can initialize your view controllers: a) in the `viewControllerWithRestorationIdentifierPath:coder:` class method, which is called *before* the controller (and therefore its view) is loaded; and b) in `decodeRestorableStateWithCoder:`,

which is called *after* the controller's view is loaded. It is recommended that you do as much as possible in `decodeRestorableStateWithCoder:`, and only the absolutely necessary stuff in the restoration class method.

The action sheet

The big red Delete Player button pops up an action sheet that asks for confirmation:



What happens when you try to restore this? See for yourself. Tap the Delete Player button and put the app in the background while the action sheet is showing. Kill and restart.

The action sheet is now gone. To be honest, that's a good thing! The HIG (Apple's Human Interface Guidelines document) recommends that you remove action sheets and alert views as soon as the user goes to the background. The user was in the middle of an action, but if s/he returns hours later, it can be confusing to see that action sheet or alert view still there.



Note: It is not required that you always restore the app to the exact state that it was in. It's good to remind the user where they were and not to lose any data, but the process of returning to your app also shouldn't create any confusion. Leaving action sheets and alert views on the screen often does exactly that.

There are other situations where you may not want to return to certain view controllers, such as a login dialog or any other transitory screens. Exactly what it makes sense to restore depends on the sort of app you're writing.

Remember, only view controllers that have their Restoration ID filled in will be restored, so if you don't want to restore a certain view controller, simply leave its Restoration ID blank.

In order to make the app behave identically when going to the background and being restored, you're always going to cancel the action sheet as soon as the app moves to the background.

Add a new `_actionSheet` instance variable to `PlayerDetailsViewController.m`:

```
@implementation PlayerDetailsViewController
{
    NSString* _game;
    UIActionSheet *_actionSheet;
}
```

Change the `deletePressed:` method to use the new instance variable:

```
- (IBAction)deletePressed:(id)sender
{
```

```
_actionSheet = [[UIActionSheet alloc]
    initWithTitle:@"Are you sure?"
    delegate:self
    cancelButtonTitle:@"Cancel"
    destructiveButtonTitle:@"Delete"
    otherButtonTitles:nil];

[_actionSheet showFromRect:self.deleteButton.frame
    inView:self.view animated:YES];
}
```

Add the following line to the `UIActionSheetDelegate` method at the bottom of the source file:

```
- (void)actionSheet:(UIActionSheet *)actionSheet
    didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    ...
    _actionSheet = nil;
}
```

This keeps track of the currently active `UIActionSheet` object in the `_actionSheet` instance variable.

Add these three methods to the class:

```
- (void)registerBackgroundNotifications
{
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(applicationDidEnterBackground:)
        name:UIApplicationDidEnterBackgroundNotification
        object:nil];
}

- (void)unregisterBackgroundNotifications
{
    [[NSNotificationCenter defaultCenter]
        removeObserver:self
        name:UIApplicationDidEnterBackgroundNotification
        object:nil];
}

- (void)applicationDidEnterBackground:(NSNotification *)
    notification
{
```

```
[_actionSheet dismissWithClickedButtonIndex:  
    _actionSheet.cancelButtonIndex animated:NO];  
}
```

This registers the view controller with `NSNotificationCenter` for the “did enter background” notification message. When this event occurs, it dismisses the action sheet by pretending to click the Cancel button. Because the `animated` parameter is `NO`, the action sheet disappears from the screen immediately.

Call the `registerBackgroundNotifications` method inside `awakeFromNib`:

```
- (void)awakeFromNib  
{  
    [super awakeFromNib];  
    self.restorationClass = [self class];  
    [self registerBackgroundNotifications];  
}
```

And unregister from notifications in the object’s `dealloc`:

```
- (void)dealloc  
{  
    [self unregisterBackgroundNotifications];  
}
```

That will do it. Run the app, and tap the Delete Player button so that the action sheet is visible. Put the app in the background and immediately open it again. The action sheet should be gone now. (It will also be gone from the screenshot that iOS takes when the app is suspended.)

Smarter table views

You already made the table view in the Choose Game screen remember its scroll position. But that’s not the only table view in our app; the Players screen also has one. It’s a good idea to give all your table views Restoration IDs.

Let’s do that for the table view on the Players screen. I named it “Players List.” Just giving it a Restoration ID should be enough to make the table view remember its position. Try it out.

Tip: If a table view in your own app doesn’t remember its scroll position, remember to make your window visible in app delegate’s `willFinishLaunching` method, like this:

```
- (BOOL)application:(UIApplication *)application
```

```
willFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // other initialization here

    [self.window makeKeyAndVisible];
    return YES;
}
```

That is usually not required in Storyboard apps, but when you use State Restoration and Preservation it is.

Also, remember that at the time of writing this chapter there is a bug in iOS 6.0 that prevents this from working in a storyboard, although it works fine in a nib-based app. As mentioned earlier, the workaround is to reload the table view at the end of `decodeRestorableStateWithCoder:`.

However, it is not really the table view that remembers which item is visible. As you may know, `UITableView` extends `UIScrollView`, and it's really the scroll view that remembers the pixel offset of where it was scrolled.

That's not necessarily a bad thing, but you can do better. By giving the table view a better idea of what it is displaying, the table view can restore its scroll position based on the actual content, not just the previous pixel offset. To make this happen, your table view data source should implement the `UIDataSourceModelAssociation` protocol, which is another addition in iOS 6.

Note: Doing this is especially important if the data can change while the app is in limbo.

For example, in the Ratings app, if the player data comes from a web API or is stored on iCloud, then when you return to the app, the list of players may actually be in a different order, depending on how you decided to restore that list.

The table view cannot simply go back to the previous index-path, because that index-path may no longer point to a valid row, or it might be for a completely different row.

The `UIDataSourceModelAssociation` protocol lets you map the table's index-paths to a logical identifier that you can use to find the same item later – if that item still exists.

Rather than storing index-paths, you say: "The user was last looking at, or had last selected, item X."

When the app comes back to life, UIKit will say: "Do you still have item X, and if so, at what index-path is it now?"

For this app, you can simply use the `playerIDs` to perform this mapping.

When your data source implements the `UIDataSourceModelAssociation` protocol, the table view will also remember which item is selected. That's not really a feature you're using in this app, but just to demonstrate how this works, you will temporarily disable the segues leading off the Players screen. That way the table view cells will stay selected when you tap them.

Add this method to **PlayersViewController.m**:

```
- (BOOL)shouldPerformSegueWithIdentifier:(NSString *)identifier
                                    sender:(id)sender
{
    return NO;
}
```

This simply disables all outgoing segues. Run the app and tap a row in the list of players; it will stay selected with a blue background. Close the app, kill it, and restore. The table view no longer has the selection, as expected.

Let's add the `UIDataSourceModelAssociation` protocol to make this work. Add it to the class extension in **PlayersViewController.m**:

```
@interface PlayersViewController : UIViewController
<PlayerDetailsViewControllerDelegate,
 RatePlayerViewControllerDelegate,
 UIDataSourceModelAssociation>
```

This protocol has two required methods:

```
#pragma mark - UIDataSourceModelAssociation

- (NSString *)modelIdentifierForElementAtIndexPath:
    (NSIndexPath *)indexPath inView:(UIView *)view
{
    Player *player = (self.players)[indexPath.row];
    return player.playerID;
}

- (NSIndexPath *)indexPathForElementWithModelIdentifier:
    (NSString *)identifier inView:(UIView *)view
{
    NSUInteger index = [self indexOfPlayerWithID:identifier];
    if (index != NSNotFound)
        return [NSIndexPath indexPathForRow:index inSection:0];
```

```
    else
        return nil;
}
```

The first one returns the player's ID for a given index-path, and the second returns an index-path for the given identifier, or `nil` if the `Player` object for that ID no longer exists. (That won't happen in this app, but if the list of `Players` was to be downloaded from somewhere, it could have changed since the last run.)

Also add this new method:

```
- (NSUInteger)indexOfPlayerWithID:(NSString *)playerID
{
    __block NSUInteger foundIndex = NSNotFound;

    [_players enumerateObjectsUsingBlock:^(Player *player,
                                           NSUInteger idx, BOOL *stop)
    {
        if ([player.playerID isEqualToString:playerID])
        {
            foundIndex = idx;
            *stop = YES;
        }
    }];
}

return foundIndex;
}
```

Give it a shot. The table view now restores the selected item.

Nib-based apps

So far, you've seen how State Preservation and Restoration works in a storyboard-based app. Unless you specify a restoration class for a view controller, UIKit will automatically look up the view controller in the storyboard and instantiate it.

But what if you don't have a storyboard and use a bunch of separate nibs in your app? Will UIKit still be able to automatically load your view controllers from the nibs? The answer is: no!

For a nib-based app, you will have to write your own restoration classes. What follows in this section are some quick tips on how to pull this off. You don't have to type this out – it's just for your reference if you ever have to do this.

To begin with, move the initialization of the initial view controller into `application:willFinishLaunchingWithOptions:` (not *did*, but *will*):

```

- (BOOL)application:(UIApplication *)application
    willFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
                                                mainScreen] bounds]];
    self.viewController = [[ViewController alloc]
                           initWithNibName:@"ViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

```

Restoring this view controller is easy because you have already instantiated it, so UIKit will be able to find it.

The nib-based Interface Builder does not have an option for setting the Restoration ID, so you'll have to set it in code. The `initWithNibName:bundle:` method is a good place for that:

```

- (id)initWithNibName:(NSString *)NibNameOrNil
                  bundle:(NSBundle *)nibBundleOrNilOrNil
{
    if ((self = [super initWithNibName:nibNameOrNilOrNil
                                bundle:nibBundleOrNilOrNil]))
    {
        self.restorationIdentifier = @"My ViewController";
    }
    return self;
}

```

For any other view controllers that you load from nibs, you need to give them a restoration class. It's easiest to make the view controller its own restoration class:

```

@interface MyTableViewController : UITableViewController
    <UIViewControllerRestoration>

```

In the `initWithNibName:bundle:` method for such a view controller, you also set the `restorationClass` property:

```

- (id)initWithNibName:(NSString *)NibNameOrNil
                  bundle:(NSBundle *)nibBundleOrNilOrNil
{
    if ((self = [super initWithNibName:nibNameOrNilOrNil
                                bundle:nibBundleOrNilOrNil]))
    {

```

```
        self.restorationIdentifier = @"My ViewController";
        self.restorationClass = [self class];
    }
    return self;
}
```

And finally, you need to provide the implementation of the restoration method that creates the view controller:

```
+ (UIViewController *)viewControllerWithRestorationIdentifierPath:
                           (NSArray *)identifierComponents
                           coder:(NSCoder *)coder
{
    NSString *identifier = [identifierComponents lastObject];
    UIViewController *viewController = [[self alloc]
                                         initWithNibName:identifier bundle:nil];

    // Configure view controller if necessary

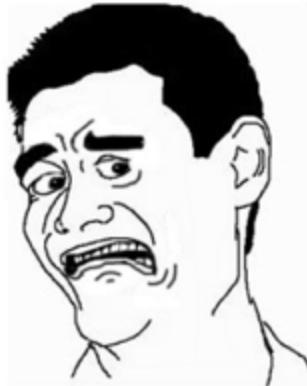
    return viewController;
}
```

That should be enough to get you going.

Restoring across different versions

Suppose that one day your user puts your app in the background and forgets about it for awhile. In the meantime, you release one or more updates to the app and the user downloads these updates, perhaps unknowingly through iTunes or over-the-air updates.

A little while later, the user needs to use your app again and launches it. If your UI has changed significantly, you may be in a world of trouble when UIKit tries to restore your old view controllers.



You can detect this situation and handle it accordingly. You have seen that the `NSCoder` contains a special key named

`UIStateRestorationViewControllerStoryboardKey` that you can use to obtain a reference to the storyboard. The coder has several other keys, one of which is `UIApplicationStateRestorationBundleVersionKey` that tells you the current version number of the app.

You can check for this in `application:shouldRestoreApplicationState:` in your App Delegate, as follows:

```
- (BOOL)application:(UIApplication *)application
    shouldRestoreApplicationState:(NSCoder *)coder
{
    NSString *bundleVersion = [coder decodeObjectForKey:
        UIApplicationStateRestorationBundleVersionKey];
    NSLog(@"Bundle version = %@", bundleVersion);

    return YES;
}
```

What you can do is save a version number from another App Delegate method, `application:willEncodeRestorableStateWithCoder:`. For example:

```
static NSString * const VersionKey = @"Version";

- (void)application:(UIApplication *)application
    willEncodeRestorableStateWithCoder:(NSCoder *)coder
{
    [coder encodeObject:@"1.0" forKey:VersionKey];
}
```

In `shouldRestoreApplicationState:`, you can now compare that version number with the bundle version and decide whether the app should restore the state, or simply launch with a clean slate:

```
- (BOOL)application:(UIApplication *)application  
    shouldRestoreApplicationState:(NSCoder *)coder  
{  
    NSString *bundleVersion = [coder decodeObjectForKey:  
        UIApplicationStateRestorationBundleVersionKey];  
    NSLog(@"Bundle version = %@", bundleVersion);  
  
    NSString *myVersion = [coder decodeObjectForKey:VersionKey];  
    NSLog(@"My version = %@", myVersion);  
  
    return [bundleVersion isEqualToString:myVersion];  
}
```

This only returns YES if the two version numbers match. You can tune this to match your own requirements.

The idea is that if it's too complicated to restore the state from the previous version (there have been too many changes in the UI), then you simply return NO from this method, and UIKit will not attempt to restore anything.

Where to go from here?

Congratulations, you have learned how to put State Preservation and Restoration to work in your apps!

It's pretty straightforward: you opt-in by implementing two methods in App Delegate, and you give a Restoration ID to every view controller and every view that you wish to restore. If you want to save more state information than what gets restored by default, you also need to encode and decode this data.

State Preservation and Restoration is an iOS 6-only feature, but definitely worth adding to your apps. If you have apps that still need to run on iOS 5, then you can set the Restoration IDs in code, but wrap them inside a call to `respondsToSelector:`. That way, the state restoration mechanism is only activated for people on iOS 6, and users of older iOS versions can still run your app.

To learn more about this cool new feature, see the "State Preservation and Restoration" chapter of the *iOS App Programming Guide* on the Developer Portal.

May your new iOS 6 Jedi Powers preserve your apps from the clutches of the dark side for all eternity! ☺

Chapter 17: What's New with Core Image

By Jacob Gundersen

Core Image is a hardware-accelerated image processing framework that can add awesome effects to still images and live video and create cool transitions between them. Core Image is built on top of OpenGL, and under the hood uses shaders to do image processing. This means it's designed to take advantage of the full power of the GPU and is very fast!

Core Image originally came out in iOS 5, but iOS 6 has extended its capabilities dramatically. In iOS 5, there were 48 filters, all focused on image manipulation. In iOS 6, there are now 93 filters! Some of the most important additions include Gaussian Blur, transition effects, and new generator filters, and we will cover some of these in this chapter.

Another addition to Core Image in iOS 6 is the ability to read and write more kinds of image data into and out of Core Image. For example, you can now import, manipulate, and export OpenGL textures. This means you can use Core Image to easily add effects to games or other OpenGL-based applications. That is very cool!

This chapter assumes you know the basics of using Core Image already. If you're new to Core Image, you should check out the Core Image chapters in *iOS 5 By Tutorials*, where we show you how you can apply a Core Image Filter (like adjusting brightness or contrast) to still images.

In this chapter, you're going to do something similar, but you'll do it with live video! This will open the door to a whole new suite of apps you can develop, and will demonstrate some of the cool new features of Core Image in iOS 6.

Core Image Review

Before we get started, let's review the most important classes in the Core Image framework:

- **CIContext.** The class that handles drawing in Core Image. It is required whenever a Core Image filter is rendered, similar to a `CGContext` in Core Graphics.

When you instantiate a `CIContext`, you tell it whether to use a GPU- or CPU-based context. A GPU-based context will be much faster, as it takes advantage of the parallel nature of the GPU hardware on the device.

However, a GPU context is limited to the hardware texture size of the GPU (1024, 2048, or 4096 pixels square depending on which device you are using). Also, a GPU-based context cannot continue to run if your app is pushed to the background.

The CPU context, on the other hand, can handle any size and can continue to run in the background. But it runs significantly more slowly. Ah, the tradeoffs!

- **CIImage**. The primary class that contains an image object. It can be created in a number of different ways and by using a number of different data representations. These include raw pixel data (`NSData`, `CVPixelBufferRef`), image data classes (`UIImage`, `CGImageRef`, etc.), and OpenGL textures.
- **CIFilter**. This class does all the pixel manipulations. A filter has some built-in methods that you can use to find out what kinds of inputs the filter takes. These inputs can be the incoming image, and a variety of parameters in dictionary form that tell the filter what to do.

`CIFilters` can also be daisy-chained together in varying combinations to make an infinite number of effects. Core Image does optimizations under the hood when filters are combined so that combinations of operations can take less time than the cumulative time it would take if those operations were performed sequentially.

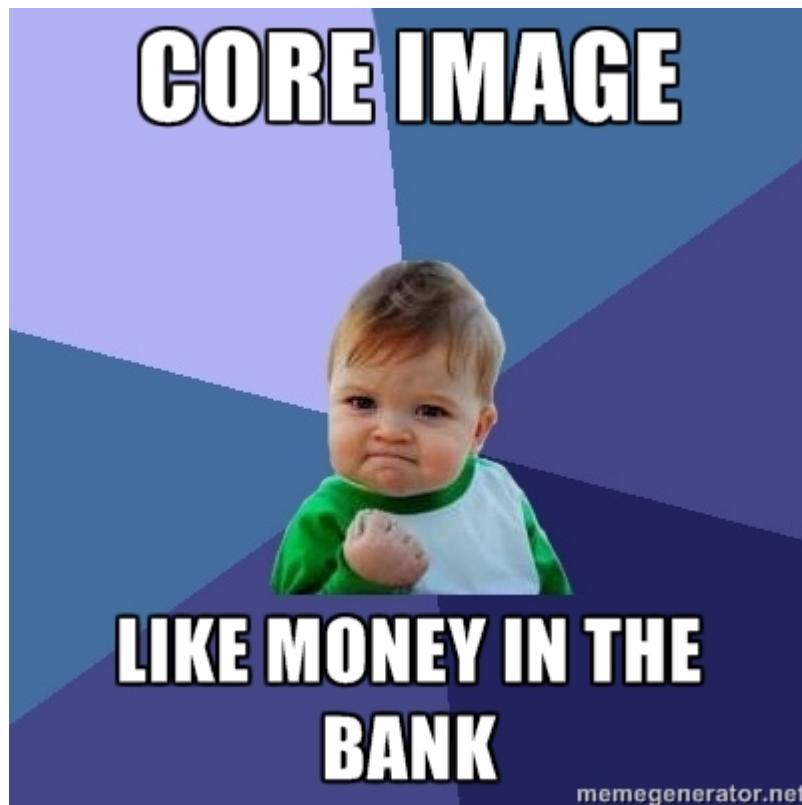
You'll be using these classes right off the bat in the first sample project.

Steps to use Core Image

You've got the classes, but how do you use them? In a typical example, you'll take the following steps in order:

1. **Import an Image**. For example, pick a `UIImage` from your camera roll and import that into a `CIImage` object using the appropriate initialization methods.
2. **Build a CIFilter Chain**. Instantiate one or more `CIFilter` objects and set their parameters using an `NSDictionary`. These parameters typically include the amount of filter effect to apply, the image to apply them to, the position of the effect, etc.
3. **Render the CIImage with a CIContext**. Call `drawImage` on a `CIContext` to draw the image into a `UIImage` or other image representation. If you want to display it on screen, you can use that `UIImage` as the image property for a `UIImageView`.

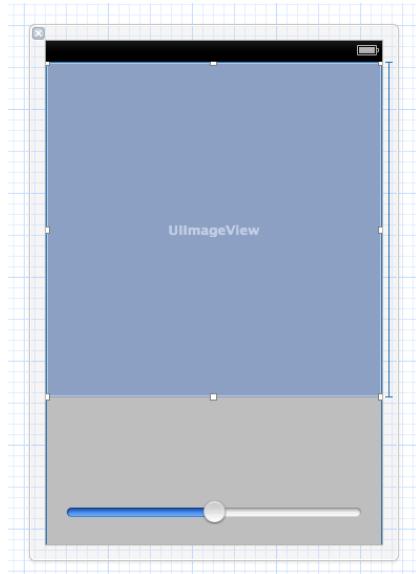
Sounds easy, right? Don't worry if not, we're about to go through a very simple example that should make these concepts crystal-clear. Soon you'll have a seasoned and no-fear attitude and will share this kid's view:



The simplest example of all time

To get you comfortable with the framework, I've prepared an elementary Core Image project that builds the simplest possible use case.

You can find this project in the resources for this chapter, named **SingleUIImageProjectStarter**. Open it in Xcode, and open **ViewController.xib**. You'll see that I've created a simple view controller with an image view and a slider:



In addition, I've already added the Core Image framework and a sample image we can display in the image view (`hubble.png`). However, that's about all the project does so far – the rest is up to you!

Let's start by getting the image view to display something. You'll put the image into a `CIIImage` first, which isn't strictly necessary at this point but will things up so it's easy for you to apply a filter to the image later.

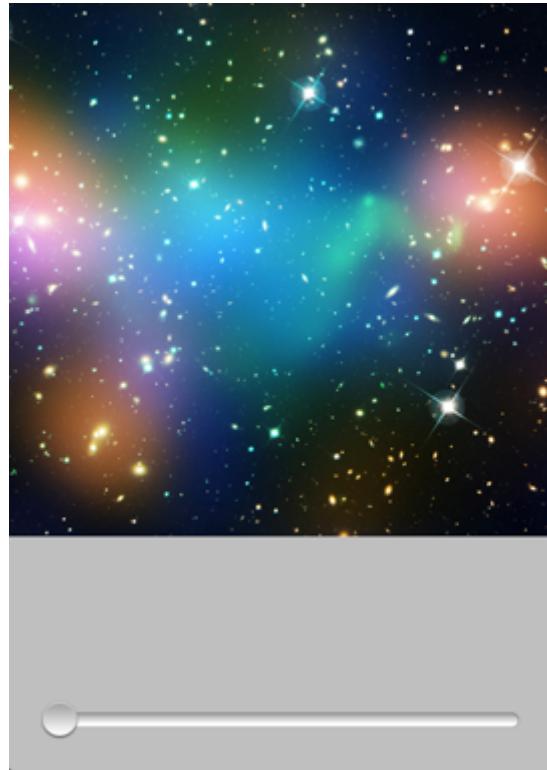
So open **ViewController.m** and add the following code to the end of `viewDidLoad`:

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"hubble"
ofType:@"png"];
NSURL *fileURL = [NSURL fileURLWithPath:path];
CIIImage *image = [CIIImage imageWithContentsOfURL:fileURL];

UIImage *uiImage = [UIImage imageWithCIIImage:image];
imgView.image = uiImage;
```

This loads an image named **hubble.png**, which is already included in your project, into a `CIIImage` (by creating an `NSURL` object with a file path) and then uses `UIImage`'s `imageWithCIIImage` method to get a `UIImage` from the `CIIImage`. Finally, it loads the `UIImage` into `imgView`, which as mentioned before, is a `UIImageView` on your main view.

Build and run, and you'll have an image proudly displayed on your device (or the Simulator).



So far, this seems pretty pointless. As mentioned earlier, you didn't need to load the `CIImage` to display the file. But, you've done it this way for good reason! Core Image filters can only operate directly on `CIImage` objects. You can load many types of image data into a `CIImage`, but you must have a `CIImage` object to use a `CIFilter`.

Time for the GOOD STUFF!

Create a `CIFilter` by replacing the `UIImage *uiImage` declaration line in `viewDidLoad` with this code:

```
CIFilter *filter = [CIFilter filterWithName:@"CISepiaTone"];
[filter setValue:image forKey:kCIInputImageKey];
[filter setValue:@1.0 forKey:@"inputIntensity"];

UIImage *uiImage = [UIImage imageWithCIImage:filter.outputImage];
```

The first line creates the workhorse of the Core Image framework, the `CIFilter` object. Next, you add a couple of parameters, the first being the most important: the input image.

Most, but not all, Core Image filters need an input image. You can use the `kCIInputImageKey` constant instead of the `@"inputImage"` string literal for convenience, but either will work as the key. Throughout this chapter, you'll use `kCIInputImageKey`.

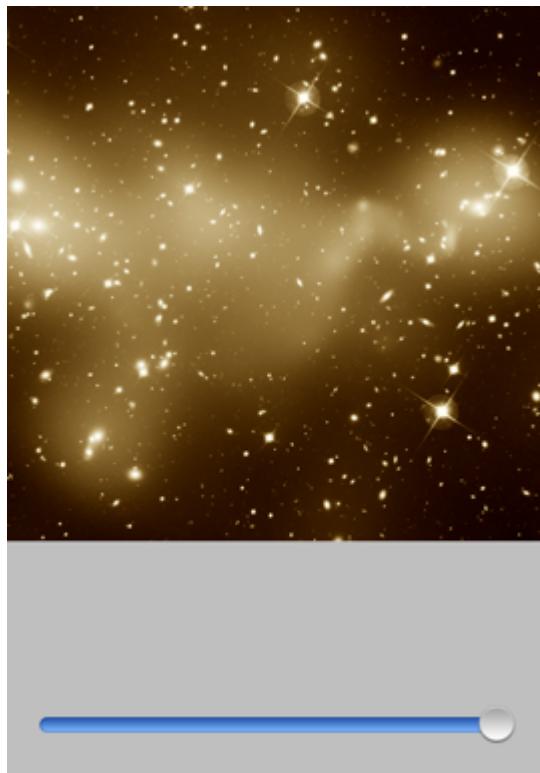
The `CISepiaTone` filter takes a second parameter, `@"inputIntensity"`, which varies from 0.0 to 1.0, and controls how much sepia to apply to the image. You're setting it to the max, 1.0.

Note: You'll be using the new literal syntax in this chapter to set `NSDictionary`, `NSArray`, and `NSNumber` values. It makes your code much easier to read. `@1.0` is the equivalent to `[NSNumber numberWithFloat:1.0]`. For more information, check out Chapter 2, "Programming in Modern Objective C."

Every filter has an `outputImage` property that represents the resulting `CIImage` after running the filter. This makes chaining filters together much easier. You access this property to pass the resulting `CIImage` to the `UIImage` initialization method `imageWithCIImage`.

This is by far the easiest way to use Core Image in your project. You can apply filters or filter chains to images very quickly and with a minimal amount of code.

Build and run now, and you'll have a sepia version of your image!



Applying filters in this manner is so easy, it's like taking candy from a baby. ☺

Sadly, it's also the most inefficient way of working with Core Image. Using `imageWithCIImage` creates a new `CIContext` every time it's invoked (and tears it down when it's done), and there's significant overhead in doing so. But if you only

need to create an image from time to time, and you wish to do so without much trouble, this is a quick and easy way to use the capabilities of Core Image.

So let's call this is the "occasional-use way." Use it when you just have a few moments free to create the image, and when you don't need real-time feedback on the adjustments (and never for live video).

This way is far too slow if you want to make live edits to an image or process live video. To see what I mean, add the following code to `changeSlider`:

```
UISlider *slide = (UISlider *)sender;

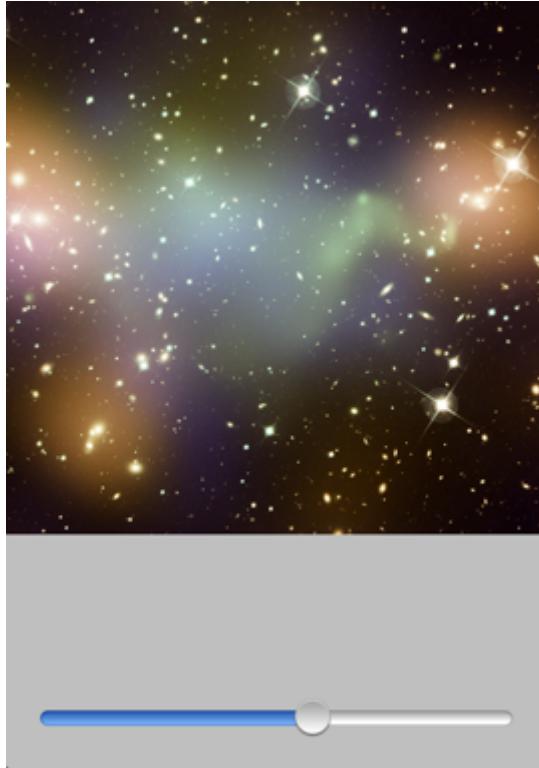
NSString *path = [[NSBundle mainBundle] pathForResource:@"hubble"
ofType:@"png"];
NSURL *fileURL = [NSURL fileURLWithPath:path];
CIIImage *image = [CIIImage imageWithContentsOfURL:fileURL];

CIFilter *filter = [CIFilter filterWithName:@"CISepiaTone"];
[filter setValue:image forKey:kCIInputImageKey];

[filter setValue:[NSNumber numberWithFloat:slide.value]
forKey:@"inputIntensity"];

UIImage *uiImage = [UIImage imageWithCIIImage:filter.outputImage];
imgView.image = uiImage;
```

You can see that you're repeating all the steps from `viewDidLoad` each time the slider value updates. This is creating and destroying a `CIContext` object every frame. Build and run now, and move the slider around.



Incredibly annoying, right?! Wayyy toooo slowwwwww.

Reusing the same Core Image objects

In order to fix this, you need to create a single instance of `CIContext` and use it to render the `CIImage`. When you don't have to rebuild and destroy the Core Image context every time, things run much faster.

Note: In order to create a Core Image context, you also need an `EAGLContext`, which means you need the OpenGL ES framework. I've already added it to this project, but you should know that you'll be using it and remember to add it to your own projects.

You don't have to import any specific headers to use the classes that you need, just make sure the framework is linked in the project.

Add the following code to the class extension (the `@interface` declaration) in **ViewController.m** (but remember to add this within curly braces since you're adding instance variables):

```
CIImage *_image;
EAGLContext *_context;
CIContext *_coreImageContext;
```

```
CIFilter *_sepiaFilter;
```

You're creating the instance variables for each object in the Core Image pipeline, in order to be more efficient. These objects only need to be set up once and can then be reused. Now replace `viewDidLoad` with the following:

```
-(void)viewDidLoad {
    [super viewDidLoad];
    // 1
    _context = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES2];
    if (!_context) {
        NSLog(@"No EAGL Context created");
    }
    // 2
    _coreImageContext = [CIContext
contextWithEAGLContext:_context];
    // 3
    NSString *path = [[NSBundle mainBundle]
pathForResource:@"hubble" ofType:@"png"];
    NSURL *fileURL = [NSURL fileURLWithPath:path];
    _image = [CIIImage imageWithContentsOfURL:fileURL];
    _sepiaFilter = [CIFilter filterWithName:@"CISepiaTone"];
    [_sepiaFilter setValue:_image forKey:kCIIInputImageKey];
    [_sepiaFilter setValue:@1.0 forKey:@"inputIntensity"];
    // 4
    CGImageRef cgImg = [_coreImageContext
createCGImage:_sepiaFilter.outputImage
fromRect:[_sepiaFilter.outputImage extent]];
    // 5
    UIImage *uiImage = [UIImage imageWithCGImage:cgImg];
    imgView.image = uiImage;
    // 6
    CGImageRelease(cgImg);
}
```

Let's run through the above code step-by-step:

1. Set up the EAGL Context. An EAGL Context is the object that manages the state. It also communicates OpenGL commands to the GPU. Any GPU rendering you might do (from 3D games to image processing) requires this object in order to communicate with the graphics hardware.
2. Set up your Core Image context. You have to pass in the EAGL Context you just created, so Core Image can communicate with the OpenGL state of the GPU.

Note: There's another initialization method for the Core Image context that allows you to pass in a dictionary of options named `contextWithOptions`. In this chapter, the default options are what we desire, so you aren't using that method here.

Some of the options have to do with managing the color space. One option controls whether the Core Image context is a CPU- or GPU-based context. This chapter is going to focus on live video image processing. A CPU context would not be fast enough for this application, so I won't be covering it here.

A couple examples of situations where you'd want to use the CPU would be:

- 1) You are processing a still image that is too large for the GPU limitations.
- 2) Your app will continue to process and save the image in the background after the user has left the app.

GPU-based processing cannot run in the background. So, any GPU processing is interrupted when the app is put in the background.

3. This section ought to look familiar, since it's identical to the previous content in `viewDidLoad`.
4. You use a new method to get the image out of the Core Image pipeline. This method is called on the context and generates a reference to a `CGImage`. You aren't using the method you used earlier, `[UIImage imageWithCIImage:]`, because this call creates a new `CIContext` every time it is used. By calling `createCGImage:fromRect:` you are using the `CIContext` object that already exists. This is much more efficient.
5. There isn't a method that generates a `UIImage` directly, so here you first create a `CGImage`, then you use that `CGImage` to create a `UIImage`.
6. You clean up the `CGImageRef`. Because this is a Core Foundation object, ARC isn't going to clean it up for you. You need to release it.

One final thing – replace `changeSlider` with the following:

```
-(IBAction)changeSlider:(id)sender {
    UISlider *slide = (UISlider *)sender;

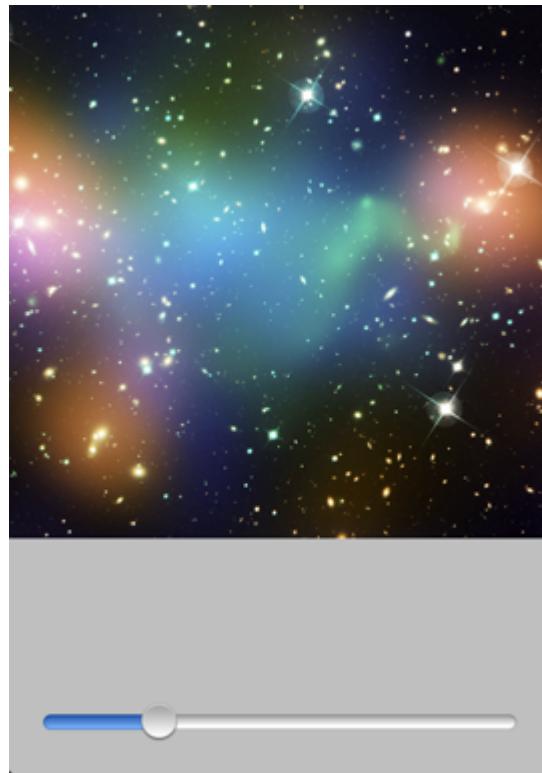
    [_sepiaFilter setValue:[NSNumber numberWithFloat:slide.value]
    forKey:@"inputIntensity"];

    CGImageRef cgImg = [_coreImageContext
    createCGImage:_sepiaFilter.outputImage
    fromRect:[_sepiaFilter.outputImage extent]];
    UIImage *uiImage = [UIImage imageWithCGImage:cgImg];
```

```
    imgView.image = uiImage;
    CGImageRelease(cgImg);
}
```

Because the `CIImage`, `CIFilter`, and `CIContext` objects are now instance variables, you don't need to create them each time this method is fired. The only things you need to do are change the value of `inputIntensity`, render the `CIImage` to a `CGImage`, and convert the resulting `CGImage` to a `UIImage`.

Build and run this now. You can run this in the Simulator, but it actually runs faster on the device (because it's efficiently using the GPU). You'll see that the slider is now much more responsive. There are ways to further optimize this code to run even faster, but this will probably be sufficient for most interactive still image scenarios.



Ahh, beautiful speed - that's more like it! 😊

Filters available in iOS 6

Before we leave this simple project behind, let's explore the Core Image query methods. The Core Image framework has a few built-in methods that you can use to find out which filters are available on the platform (it's different on OS X and iOS). You can also use these methods to find out the parameters for each filter.

Dumping this information out is a useful way to get a reference to what's available. Let's try - add the following code to `viewDidLoad` immediately after the `[super viewDidLoad]` line:

```
NSArray *ciFilters = [CIFilter  
filterNamesInCategory:kCICategoryBuiltIn];  
for (NSString *filter in ciFilters) {  
    NSLog(@"filter name %@", filter);  
    NSLog(@"filter %@", [[CIFilter filterWithName:filter]  
attributes]);  
}
```

The `filterNamesInCategory` method returns an array of all the names of the filters in that category. The constant `kCICategoryBuiltIn` returns a list of all the filters available on the platform that you're using (iOS in this case, but you knew that).

You use the name of the filter to create an instance of that filter. Then you can call the `attributes` method, which returns an `NSDictionary` containing the attributes of that filter, including the filter's name, the input parameters it accepts, and information about each input parameter (acceptable range, data type, `UISlider` values, etc.).

Build and run now, and you'll get a list in the console, like this:

```
    CIAttributeType = CIAttributeTypeImage;  
};  
inputRadius = {  
    CIAtributeClass = NSNumber;  
    CIAtributeDefault = 300;  
    CIAtributeIdentity = 0;  
    CIAtributeMin = 0;  
    CIAtributeSliderMax = 800;  
    CIAtributeSliderMin = 0;  
    CIAtributeType = CIAtributeTypeDistance;  
};  
}  
2012-07-10 23:39:54.813 SingleUIImageProject[287:907] filter name CIWhitePointAdjust  
2012-07-10 23:39:54.815 SingleUIImageProject[287:907] filter {  
    CIAtributeFilterCategories = {  
        CICategoryColorAdjustment,  
        CICategoryVideo,  
        CICategoryStillImage,  
        CICategoryInterlaced,  
        CICategoryNonSquarePixels,  
        CICategoryBuiltIn  
    };  
    CIAtributeFilterDisplayName = "White Point Adjust";  
    CIAtributeFilterName = CIWhitePointAdjust;  
    inputColor = {  
        CIAtributeClass = CIColor;  
        CIAtributeDefault = "(1 1 1 1)";  
        CIAtributeIdentity = "(1 1 1 1)";  
        CIAtributeType = CIAtributeTypeColor;  
    };  
    inputImage = {  
        CIAtributeClass = CIImage;  
        CIAtributeType = CIAtributeTypeImage;  
    };  
}
```

You asked for information and you got it! If you look through the list, you'll see there's a massive amount of filters and options available – we'll explore some of these in the rest of these chapters.

Note: One thing worth noting about the input parameters – the keys returned by the `attributes` method (`inputColor`, `inputImage`, `inputRadius`, etc.) are the names of the keys you'll use to set the values. Alternatively, there are sometimes (but not in every case) constants that can be used for convenience, including `kCIInputImageKey` and `kCIInputBackgroundImageKey`.

Moving data in and out of Core Image

Before we move on to the project that will occupy you for the rest of this chapter, let's take a look at the many number of ways to move image data into and out of Core Image.

First, here's your toolbox for moving data in:

1. Image file. `+[CIIImage imageWithContentsOfFile:]`

This method is for loading an image from a file, such as a PNG, JPG, etc.

2. CGImage. `+[CIIImage imageWithCGImage:]`

This method is used when you have a `CGImage`. You may have a `CGImage` from a Core Graphics context that you've drawn, or potentially from an unsupported file format.

3. CVPixelBuffer. `+[CIIImage imageWithCVPixelBuffer:]`

This method will probably be used most often in conjunction with AVFoundation. In this chapter, you'll use this method to get data from the camera, but you could also use it from a video file output.

4. NSData. `+[CIIImage imageWithData,
imageWithBitmapData:bytesPerRow:size:format:colorSpace:]`

If you are programmatically creating a texture, or have a file format that is unsupported, you may need to make use of this method.

5. OpenGL Texture. `+[CIIImage imageWithTexture:size:flipped:colorSpace:]`

This method is used to interface with an OpenGL texture. A texture could come from a file or could be the result of an OpenGL render to texture process. It opens a lot of extremely powerful options for using Core Image in games or other graphically intensive applications.

And here are the methods you can use to get the result of a filter chain *out* of Core Image:

1. Generate a UIImage by calling `[UIImage imageWithCIIImage]`.

This is the easiest, but most inefficient way, to get a `UIImage`. You could display a `UIImage` in an `UIImageView` or save it to disk.

2. Generate a CGImage by calling `[CIContext createCGImage:fromRect:]`.

This is a much better-performing method. A `CGImage` can be used in a number of ways: in a Core Graphics drawing scenario, imported into OpenGL, saved to disk, or converted to a `UIImage`.

3. Generate a **CVPixelBuffer** by calling `[CIContext render:toCVPixelBuffer:]`.

A `CVPixelBuffer` can be converted into a `CGImage`, or it could be used to write video frames into a video file. You'll be doing this later.

4. Render to a **frame buffer** or **render buffer** with `[CIContext drawImage:inRect:fromRect:]`.

If this call draws to a render buffer, it will be displayed onscreen. This is the most efficient method of displaying the contents of the Core Image chain to the screen. If the Core Image writes to a render buffer, that could be used for further graphics work in an OpenGL environment.

Feeling confident, equipped, and ready to code something epic? Read on!

Introducing FilterMe

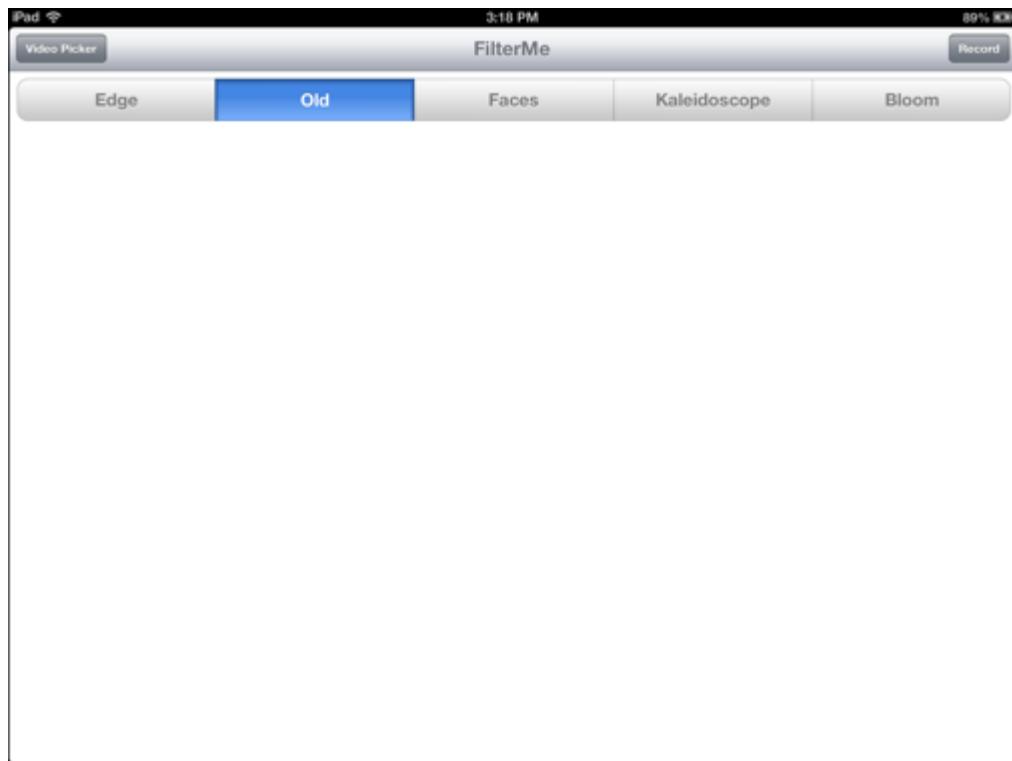
In the rest of this chapter, you are going to build a live video effects app called FilterMe. This will involve creating five filter chains to show off the various capabilities of Core Image. In the process, you will learn how to capture data from the device's camera and write the processed video frames to a video file, complete with audio. I don't mean to toot my own horn, but this could be the coolest tutorial project ever! ☺

I've prepared a starter project that already has a number of things in place. You can find it in the resources for this chapter called **FilterMeStarter.zip**. This project has the following frameworks already added:

- **ImageIO**
- **QuartzCore**
- **AVFoundation**
- **CoreImage**
- **CoreMedia**
- **CoreVideo**
- **MobileCoreService**
- **OpenGL**
- **GLKit**
- **AssetsLibrary**

Wow, that's a lot of frameworks!

It also has a couple of buttons and a `GLKView` for rendering the resulting view. Find the project in the resources for this chapter. If you build and run it, you'll get a view with a segmented control, a record button, a video picker button, and no video.



We will remedy that *tout-de-suite* – that means right away, *en français!*



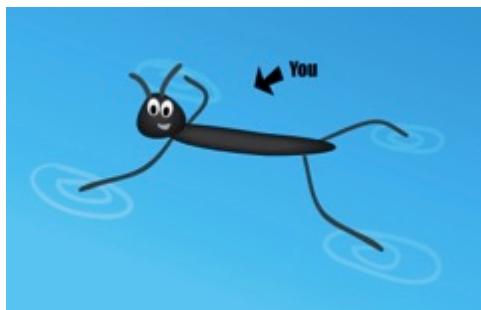
AVFoundation basics

AVFoundation is the primary framework on the iOS platform for capturing, recording, playing, and manipulating audio and video assets. This chapter will walk you through two main functions of the framework: capturing and recording video and audio data.

AVFoundation contains classes for capturing both audio input (from a mic, headphones, or Bluetooth) and video input (from the front and back cameras). You can use it to read and write audio files, still images, and video files, and to create compositions by mixing these file types together.

As this is a chapter about Core Image and not AVFoundation, I won't cover AVFoundation exhaustively. However, I will try to give you a complete understanding of the classes you'll be using.

We're going to skim across a deep topic (AVFoundation) like a water skeeter skims across the top of a pond. You'll be able to see the depth, but you won't be diving in so that we can keep the focus on Core Image.



You'll use classes that capture from the camera and the microphone, and output raw data. You'll feed the video data through the Core Image pipeline. Once Core Image has manipulated the video data, that data and the audio data will be fed into classes designed to write audio and video data to a video file.

Let's start with the **AVCapture** classes. Here are the objects you'll use:

- **AVCaptureSession**. This is the manager class that controls the flow of the capture experience. You'll configure and add multiple inputs and outputs to this class, which is responsible for starting and stopping the session.
- **AVCaptureDevice**. This class is responsible for the state of the input device. In the case of a camera, the flash, exposure, white balance, and focus can be manipulated through this object interface. The device object will be used to create an `AVCaptureDeviceInput`.
- **AVCaptureDeviceInput**. This class is added to the session object and is created using the `AVCaptureDevice`.
- **AVCaptureConnection**. This object describes a connection between a capture input and capture output. You won't be deliberately creating an instance of this object in this project, but one will be automatically created when you add inputs and outputs to the session.
- **AVCaptureOutput**. This is an abstract class that represents an output from a capture session. It has a bunch of different concrete subclasses, including `AVCaptureAudioDataOutput`, `AVCaptureVideoDataOutput`, `AVCaptureStillImageOutput`, `AVCaptureMovieFileOutput` and

`AVCaptureAudioFileOutput`. It should be clear from the names of these objects what they do. In this project, you'll use the data output classes.

- **AVCaptureVideoDataOutputSampleBufferDelegate** and **AVCaptureAudioDataOutputSampleBufferDelegate**. These are the protocol methods for the `AVCaptureVideoDataOutput` and `AVCaptureAudioDataOutput` classes. These protocols have one method for obtaining the data from the inputs (camera or mic). Both data types are sent to the same method. Each time there is new data available, `captureOutput:didOutputSampleBuffer:fromConnection:` will be called with a new sample buffer with the audio or video data. If a frame or audio segment is dropped, there's a method to handle that scenario.
- Don't worry if some of this seems abstract or confusing at the moment – you'll try it out next, and you can come back to this for reference. Get your devices ready... it's time to build!

Lights, camera... action!

First add a capture session instance variable. In **ViewController.m**, change the beginning of the file to the following (replace the current `@interface` section):

```
#import <AVFoundation/AVFoundation.h>
#import "ViewController.h"
#import <GLKit/GLKit.h>

@interface ViewController : UIViewController
<AVCaptureVideoDataOutputSampleBufferDelegate> {
    AVCaptureSession *_session;
}

@end
```

In this code, you first import the necessary frameworks. You declare that this class implements the `AVCaptureVideoDataOutputSampleBufferDelegate`. You need this delegate to receive the video data buffers from the capture session. Then you add an instance variable for the capture session.

Next, replace the contents of `viewDidLoad` with the following:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 1
    _session = [[AVCaptureSession alloc] init];
    // 2
    [_session beginConfiguration];
    // 3
    if(!_session
```

```
canSetSessionPreset:AVCaptureSessionPreset640x480]) {
    [_session
        setSessionPreset:AVCaptureSessionPreset640x480];
}
// 4
NSArray *devices = [AVCaptureDevice devices];
// 5
AVCaptureDevice *videoDevice = [AVCaptureDevice
    defaultDeviceWithMediaType:AVMediaTypeVideo];
// 6
for (AVCaptureDevice *d in devices) {
    if (d.position == AVCaptureDevicePositionFront &&
        [d hasMediaType:AVMediaTypeVideo]) {
        videoDevice = d;
        break;
    }
}
// 7
NSError *err;
AVCaptureDeviceInput *videoInput = [AVCaptureDeviceInput
    deviceInputWithDevice:videoDevice error:&err];
[_session addInput:videoInput];
// 8
AVCaptureVideoDataOutput *videoOutput =
    [[AVCaptureVideoDataOutput alloc] init];
// 9
[videoOutput setAlwaysDiscardsLateVideoFrames:YES];
// 10
[videoOutput
    setVideoSettings:@{(id)kCVPixelBufferPixelFormatTypeKey :
    @(kCVPixelFormatType_32BGRA)}];
// 11
[videoOutput setSampleBufferDelegate:self
    queue:dispatch_get_main_queue()];
// 12
[_session addOutput:videoOutput];
[_session commitConfiguration];
[_session startRunning];
}
```

All of this code is required to set up a capture session that takes video from the camera and outputs the pixel data to the delegate method. Here's what the code does:

1. Creates the `AVCaptureSession` object.

2. Calls `beginConfiguration` on the session. A `beginConfiguration` call is paired with a `commitConfiguration` call. When using these methods, you'll batch all the configuration changes to be done atomically when the `commitConfiguration` is called. This is more critical when changing configurations while the session is running (like when you switch from front to back cameras), but for the sake of good coding habits, you're using them here.
3. Sets the incoming resolution of the video capture session. You first call `canSetSessionPreset` in order to make sure that the preset is supported on the device. All the cameras can output 640x480, so this should always work (assuming that you have a camera).

Note: There are other presets available. Some presets are fixed resolution, like the one you're using here. These resolutions may not be available on all devices. Other presets may be different on each device: for example, `AVCaptureSessionPresetHigh`, which will use the highest resolution available for each device.

Here is a list of available session presets:

```
AVCaptureSessionPresetPhoto  
AVCaptureSessionPresetHigh  
AVCaptureSessionPresetMedium  
AVCaptureSessionPresetLow  
AVCaptureSessionPreset320x240  
AVCaptureSessionPreset352x288  
AVCaptureSessionPreset640x480  
AVCaptureSessionPreset960x540  
AVCaptureSessionPreset1280x720
```

4. Gets an array of all the input devices on the current hardware. You're using the front camera, if the device has one. However, if the device doesn't have a front camera, you'll use the back camera.
5. Gets the default camera. The next line gets the default device for the `AVMediaTypeVideo`, which will always be the back camera.
6. Iterates through the devices looking for those that have the front position and the correct media type (this array also contains the mic). If you find a device that has front as its position, you set your input device to that camera. If one isn't found, then you proceed with the back camera.

7. Creates the `AVCaptureDeviceInput` object using the capture device just determined. This is necessary to add the device to your session.
8. Creates the `AVCaptureVideoDataOutput` object. This is one of many possible output types. In this project, you'll only be using this output type, which supplies the raw pixel data for each frame in a method callback. Other available output types include still images, video files, and preview layer (to screen).
9. Sets the `setAlwaysDiscardsLateVideoFrames` parameter. This setting is recommended for real-time video processing.
10. Sets the pixel format to BGRA, which is recommended for Core Image processing.
11. Sets the delegate object, which will receive the pixel buffers through the callback method. That will be the `ViewController`. This method also sets the queue that's responsible for handling the processing of the video data. You're just using the main thread. You could optimize it further by using a separate queue, but such things are beyond the scope of this chapter.
12. Finally, this code adds the output to the session, commits the configuration, and starts the capture session.

You're almost ready to capture video! You need just one more thing in order to get it working – to implement the callback method. Add this method to the class:

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput  
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer  
fromConnection:(AVCaptureConnection *)connection {  
    CVPixelBufferRef pixelBuffer = (CVPixelBufferRef)  
        CMSampleBufferGetImageBuffer(sampleBuffer);  
  
    int width = CVPixelBufferGetWidth(pixelBuffer);  
    int height = CVPixelBufferGetHeight(pixelBuffer);  
  
    NSLog(@"got sample buffer, width %d, height %d",  
          width, height);  
}
```

You're going to use this method to process the pixel data. This method gets both audio (if you added an audio input) and video data. The first thing you do is get the `CMPixelBufferRef` from the `CMSampleBufferRef`. Then you use the `CVPixelBufferGet` methods to obtain the width and height of the pixel buffer.

Finally, you simply log the size of the buffer for the time being, just so you can see it working.

Build and run, and you should see output similar to this in the console (but only if you are on device – on the Simulator, you won't see anything at all, since there is no camera!):

```
All Output • Capture OpenGL ES frame
2012-09-05 15:23:27.399 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.433 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.467 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.500 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.533 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.567 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.600 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.634 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.667 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.700 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.733 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.767 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.800 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.834 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.867 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.900 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.933 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:27.967 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:28.001 FilterMe[9376:907] got sample buffer, width 640, height 480
2012-09-05 15:23:28.034 FilterMe[9376:907] got sample buffer, width 640, height 480
(lldb)
```

Passing raw pixel data into Core Image

Now that you've got camera capture data coming in, it's time to do something with it. You're going to set up a Core Image environment like you did in the simple project you created earlier. However, instead of pushing a single image to a `UIImageView`, you'll render the frames to a `GLKView`.

Core Image is a wrapper around OpenGL shaders. When any drawing is done in Core Image, it's done first into an OpenGL buffer. If you draw directly to the `GLKView` (a wrapper around OpenGL) this operation is very fast. If you create an image object (either a `CGImageRef` or a `UIImage`, it first draws into an OpenGL buffer and then reads the data from that buffer. So, in those cases where you are writing to a `UIImage` and setting a `UIImageView`, it's doing more work to get that data to the screen.

You need the Core Image context, an `EAGLContext`, and an `IBOutlet` to the `GLKView` in the storyboard. Add the following instance variables to the `@interface` section at the top of `ViewController.m`:

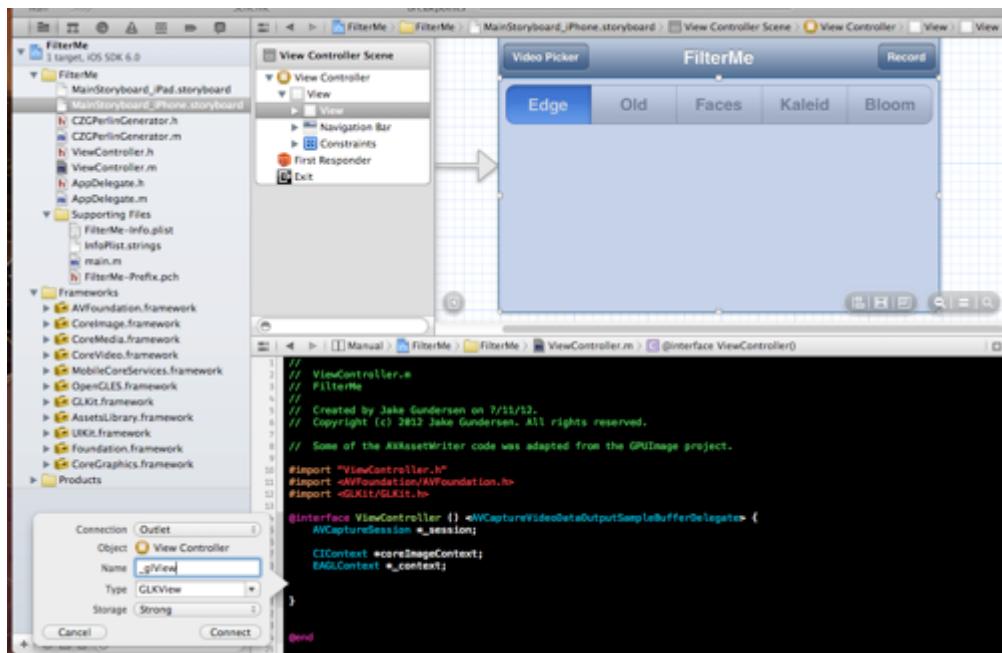
```
CIContext *_coreImageContext;
EAGLContext *_context;
```

Also, create an instance variable for the `GLKView` in the storyboard. To do this, bring up the storyboard file (whichever you are using, either the iPhone or iPad version). Expand the `ViewController` in the "View Controller Scene" container. Expand the root view. You should see a child view along with the controls.

Activate the Assistant Editor by either clicking on its button, or, if you're not sure which button it is, by pressing Option-Command-Comma (,),.

The Assistant Editor will probably bring up the `ViewController.h` file. Switch to the `ViewController.m` file, by clicking on the .h file and selecting the .m from the drop-down menu.

Command-click and drag from the `GLKView` container into the `@interface` section in **ViewController.m**. Create a `_glView` `IBOutlet` to the `GLKView` object in the storyboard, as in the screenshot below:



If you want to test on both iPad and iPhone, you'll need to connect the other storyboard `GLKView` object to this outlet as well. That's done in a similar fashion (Command-click and drag from the container to the `IBOutlet`).

All right, you're ready to set up the `GLKview`, the Core Image context, and the `EAGLContext`. The latter two objects are constructed exactly as they were in your **SimpleUIImage** project.

Add this code to the `viewDidLoad` after the `[super viewDidLoad];` line:

```

_context = [[EAGLContext alloc]
            initWithAPI:kEAGLRenderingAPIOpenGLES2];

if (!_context) {
    NSLog(@"Failed to create ES context");
}

_glView.context = _context;
float screenWidth =
    [[UIScreen mainScreen] bounds].size.height;
_glView.contentScaleFactor = 640.0 / screenWidth;
_coreImageContext = [CIContext
    contextWithEAGLContext:_context];

```

The code that sets up the EAGL and Core Image contexts should look familiar.

In the middle section, you first set the `GLKView`'s context to the `EAGLContext`. A `GLKView` needs an `EAGLContext` because it is an OpenGL view. As I mentioned earlier, in order to work with OpenGL, one needs an `EAGLContext`.

The next two lines set up the content scale factor of `_glview`. The `contentScaleFactor` is a value that tells the view how many screen pixels to render per UIKit point. You're using a value of 640 divided by the screen width. 640 is the width of the video coming in from the camera. By dividing by the screen width, you'll get that video filling the entire screen. This doesn't change the video data, it is still processed and recorded at 640 x 480.

Note: Core Image performance scales linearly, depending on the number of output pixels. Depending on how many Core Image filters are chained together, you probably won't be able to input HD video and render to the full iPad resolution. The recommended solution is to use Core Animation to scale up.

Depending on what you're trying to do with your filters, you'll want to try to work with a larger resolution than this. Try to start with as high a resolution as you can (full retina resolution), and scale this down as needed in order to obtain the required frames per second.

Start with a resolution such as full HD, 1920 x 1080 or 1280 x 720 by changing the camera preset to one of the higher options. This will mean also changing the `contentScaleFactor`. As you build the Core Image filter chain, you will probably find that full HD is too many pixels to render at full video speed. As you need to, lower the preset and change the `contentScaleFactor` accordingly. Try to do as high resolution video as you can, but still retain full frame rates (24 to 30fps).

Next you'll add a way to display the incoming frames to the screen.

Replace `captureOutput:didOutputSampleBuffer:fromConnection:` with the following:

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput  
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer  
fromConnection:(AVCaptureConnection *)connection {  
    CVPixelBufferRef pixelBuffer = (CVPixelBufferRef)  
        CMSampleBufferGetImageBuffer(sampleBuffer);  
  
    CIImage *image = [CIImage  
        imageWithCVPixelBuffer:pixelBuffer];  
  
    [_coreImageContext drawImage:image inRect:[image extent]  
        fromRect:[image extent]];  
    [_context presentRenderbuffer:GL_RENDERBUFFER];
```

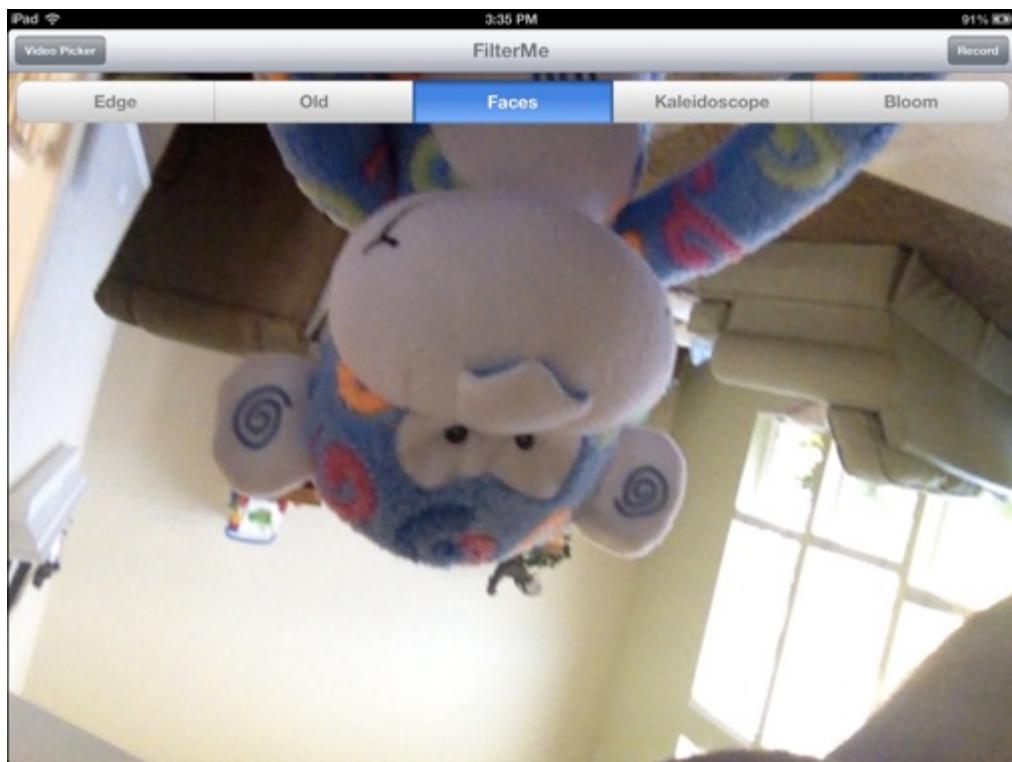
```
}
```

As in the previous scenario, the first step is to get the `pixelBuffer` from the `sampleBuffer`. The next step is to create a `CIImage` from the pixel buffer. You use the appropriate method to initialize a `CIImage`.

The next step is to use the `drawImage:inRect:fromRect:` method to render the `CIImage` to the render buffer. Because you're using a `GLKView`, all the necessary OpenGL setup is taken care of by this class – all you have to do is call the `cicontext` method to render to the OpenGL render buffer.

The final step is to call `presentRenderBuffer` on the EAGL context. This call displays the contents of the render buffer in your `GLKView`.

Build and run now. You should have live video coming in from your camera. Depending on the device orientation, the video could be upside-down (or even stretched/skewed). You'll fix that next.



Handling orientation

Your app will support both right and left landscape orientations – but to do this you need to know which orientation the device is in. If it's in the `UIInterfaceOrientationLandscapeRight` orientation, you'll need to flip the incoming video 180 degrees.

First, set up `supportedInterfaceOrientations`, the new autorotation method for your view controller introduced in iOS 6:

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

The new method takes one of the new enums. You're passing the `UIInterfaceOrientationMaskLandscape`, which means rotate to any landscape orientation.

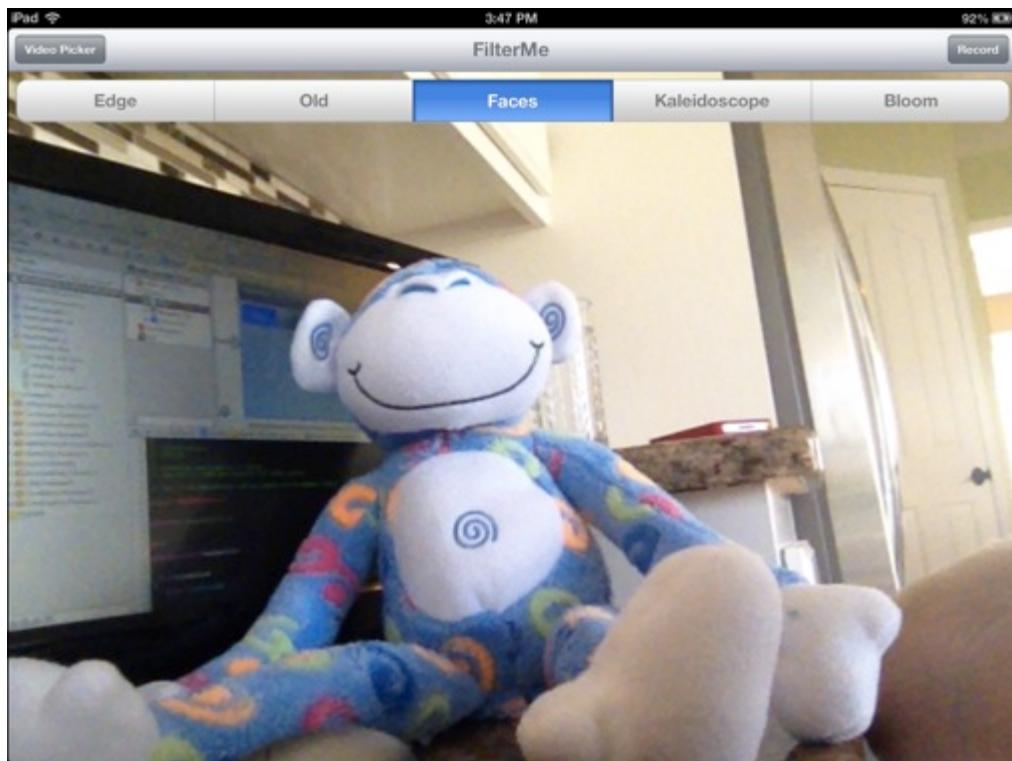
Now you need to handle the pixel buffer. As we said, if the device is in Landscape Right, you need to rotate the image 180 degrees.

Add the following code to `captureOutput:didOutputSampleBuffer:fromConnection:`, right after the line that creates `image`:

```
if (self.interfaceOrientation ==
    UIInterfaceOrientationLandscapeRight) {
    image = [image imageByApplyingTransform:
        CGAffineTransformMakeRotation(M_PI)];
    image = [image imageByApplyingTransform:
        CGAffineTransformMakeTranslation(640.0, 480.0)];
}
```

At the time of writing this chapter, a `CGAffineTransformMakeRotation` will rotate the image around the bottom-left origin point. The resulting image is below and to the left of the screen. That means that you also need to translate the image left and up.

Now if you build and run, your video should be right-side up.



That's more like it!

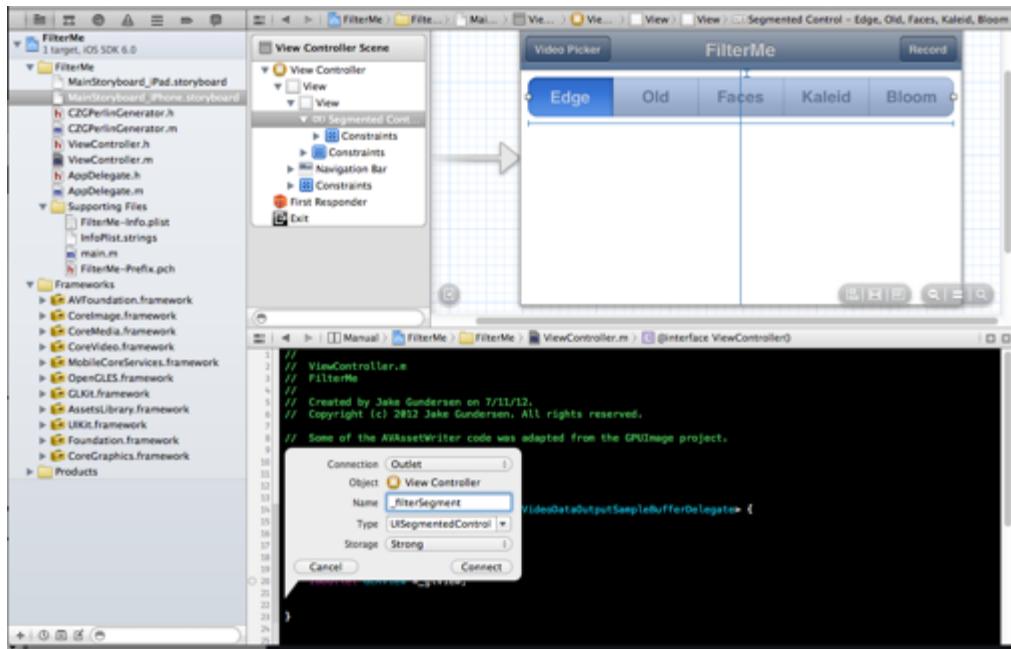
Core Image chains

In this section, you're going to create five Core Image filter chains. These filter chains will serve to introduce you to the capabilities of the framework, as well as the performance costs of some of the filters.

The first step is to connect the segmented control to your class so that when you choose a segment, the filter chain will change. All this will happen in real-time and can be recorded to a movie file (once we get to that point).

You will see that it is awesome!

Open up the storyboard file and the Assistant Editor, as you did before. Select **ViewController.m** in the bottom window. Control-click and drag from the segmented control to the @interface section of the implementation. Name the outlet `_filterSegment`. It should look like this:



The bloom filter

The first filter chain you'll create is the simplest. It will consist of a single filter, the new `CIBloom` filter introduced in iOS 6. A bloom is often used in video games and other video effects to add a "halo" to a light source.

Each filter you create will have its own method. Each method will take a `CIImage` as an argument and return a `CIImage`. Create the bloom filter now by adding this method to `ViewController.m`:

```
-(CIImage *)bloom:(CIImage *)inputImage {
    CIFilter *bloom = [CIFilter filterWithName:@"CIBloom"];
    [bloom setValue:inputImage forKey:kCIInputImageKey];
    [bloom setValue:@100 forKey:@"inputRadius"];
    [bloom setValue:@1.0f forKey:@"inputIntensity"];
    return bloom.outputImage;
}
```

The first line initializes the filter by name. The second adds the `inputImage` as the `kCIInputImageKey` parameter.

The next two lines set the parameter values for the input radius and the input intensity. These values can be altered to change the look of the effect. The `inputRadius` has a range from 1 to 100 and the `inputIntensity` ranges from 0 and 1. You've set both to max here.

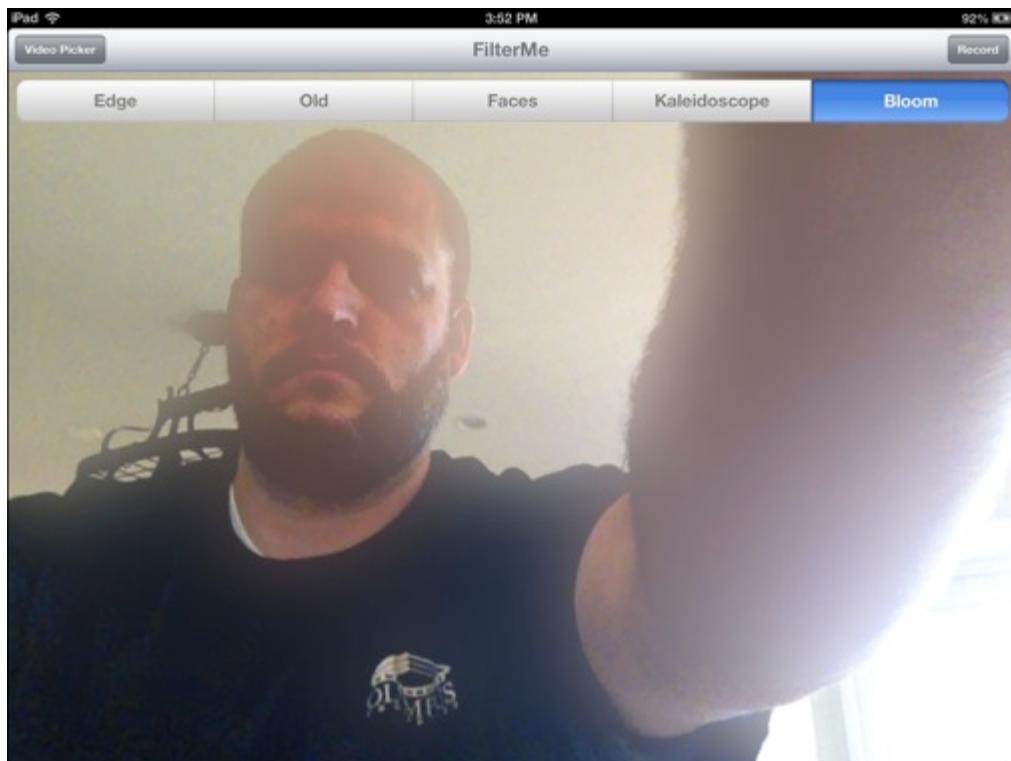
Now you'll alter the callback for the `didOutputSampleBuffer` to use one of the appropriate image filter chains. Eventually, you'll have five filters to choose from.

Add the following code to `captureOutput:didOutputSampleBuffer:fromConnection:` before the `[_coreImageContext drawImage:...]` line:

```
switch (_filterSegment.selectedSegmentIndex) {  
    case 0:  
        //  
        break;  
    case 1:  
        //  
        break;  
    case 2:  
        //  
        break;  
    case 3:  
        //  
        break;  
    case 4:  
        image = [self bloom:image];  
        break;  
    default:  
        break;  
}
```

This switch statement will use one of the five methods (that you'll write) to alter the image object in one of – you guessed it – five ways.

So far, you only have the bloom set up and it's set to the last button in the segment. Build and run now. Selecting the bloom button will give you the following effect:



It's just a coincidence that I'm wearing a shirt that says "Olympus" on it – I may be a god among coders, but despite the halo and beard, I'm not a Greek deity. ☺

The kaleidoscope filter

Next, you're going to use a rotated tile filter. There are a number of tile filters newly available in iOS 6. This filter takes a piece of the input image and repeats it over and over. In this case, you'll use the `CISixfoldRotatedTile` filter. This will create an effect much like looking through a kaleidoscope.

Add the following method somewhere in the file:

```
- (CIImage *)kaleidoscope:(CIImage *)inputImage {
    CIFilter *kaleid = [CIFilter
        filterWithName:@"CISixfoldRotatedTile"];
    [kaleid setValue:inputImage forKey:kCIInputImageKey];
    [kaleid setValue:[CIVector vectorWithCGPoint:
        CGPointMake(320, 240)] forKey:@"inputCenter"];
    [kaleid setValue:@200 forKey:@"inputWidth"];

    CFAbsoluteTime timeNow = CFAbsoluteTimeGetCurrent();
    timeNow = (fmodl(timeNow, 4 * 6.28) / 4.0) - 3.14;

    [kaleid setValue:@(timeNow) forKey:@"inputAngle"];
```

```
CIFilter *crop = [CIFilter filterWithName:@"CICrop"];
[crop setValue:kaleid.outputImage forKey:kCIInputImageKey];
[crop setValue:[CIVector vectorWithCGRect:
[inputImage extent]] forKey:@"inputRectangle"];

return crop.outputImage;
}
```

The first two lines should be very familiar by now. The third line sets the center of the rendered image. The reflections start at this point.

Note: This code uses a Core Image class that I haven't mentioned yet. The **CIVector** class is an input for many of the parameters in Core Image. **CIVector** can be used to construct a number of inputs including, **CGPoint**, **CGRect**, **CGAffineTransform**, and a number of others. Querying the **attributes** method of a given filter will let you know if you should use a **CIVector** class to input the parameters.

The fourth line sets the width. This is the size of the input image tile.

The next two lines get the current time on the clock in seconds. From this value, you generate a number between -3.14 and 3.14 that cycles every 25 seconds or so. This is the input angle for the tile and it will slowly rotate around. You set it in the next line.

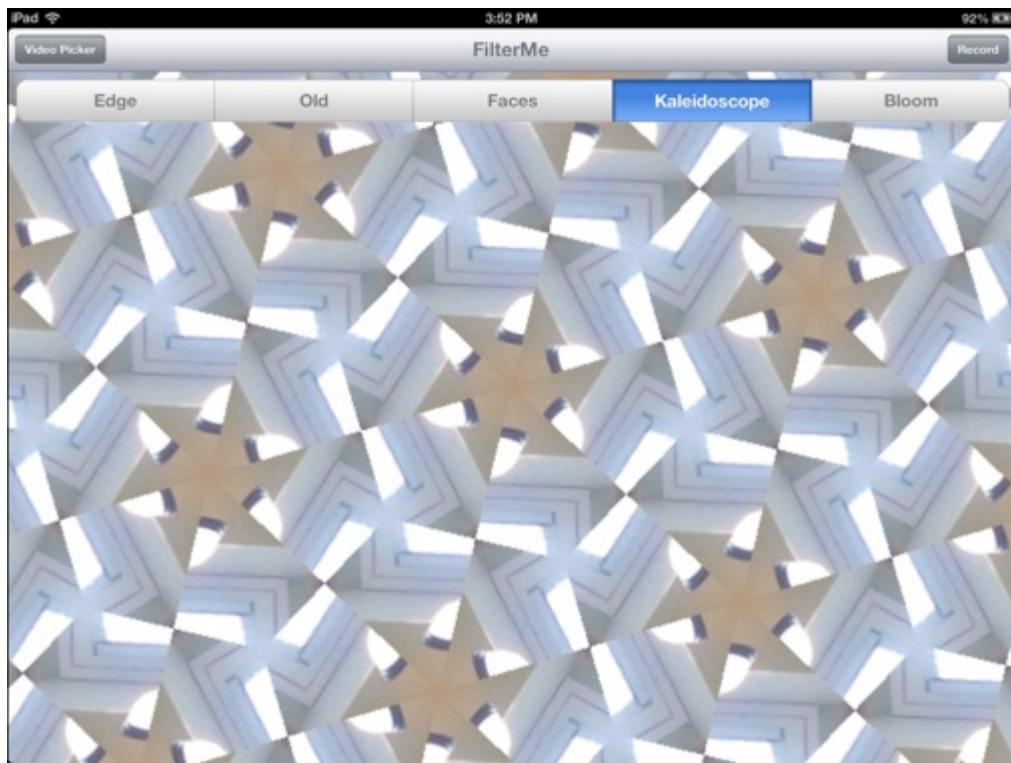
Some kinds of filters generate an infinite image. **CIColorGenerator** is an example of this – the color goes on infinitely. **CISixfoldRotatedTile** (and all repeating tile filters) is another example.

If you try to render an infinite image, you'll get an error. So, before you can render this image, you must crop it. The next three lines set up a crop filter. It crops your infinite image to your standard size (640 x 480).

Now you just call this new method to `case 3:` under `captureOutput:didOutputSampleBuffer:fromConnection:` (before the `break`):

```
image = [self kaleidoscope:image];
```

Build and run. If you select the fourth segment (it's called Kaleidoscope on the iPad, but Circle on the iPhone due to space constraints ☺), you'll have something like this:



My kitchen... it's... disintegrating!

Simple edge detection

There isn't an edge detection filter in the iOS version of Core Image, but, with the Gaussian Blur filter, you can create a rudimentary version. Edge detection is used in a lot of computer vision applications. It creates a black and white image that shows the edges of whatever is in the scene. This filter chain includes six filters. Of these six only one is newly available in iOS 6, the CIGaussianBlur filter. This filter is the most important in the chain.

This filter is the most complex of the five you'll use in this project, and the most costly in terms of performance. It is a good example of how really interesting effects can be achieved by combining filters.

This filter was adapted from a Photoshop tutorial about how to create a sketch effect on a photograph. You can check out that tutorial here:

<http://www.photoshopessentials.com/photo-effects/portrait-to-sketch/>

Add this method for the edge detection filter somewhere in the file:

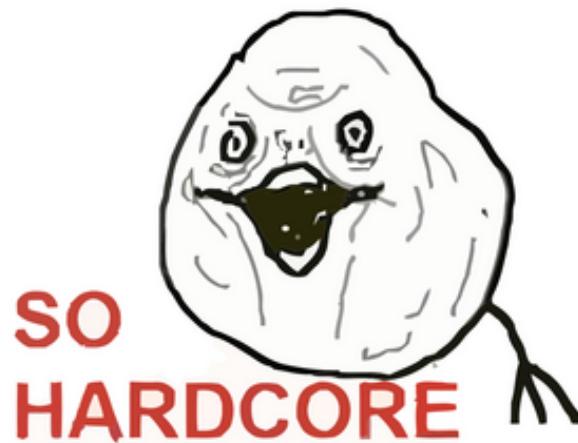
```
-(CIImage *)simpleEdgeDetection:(CIImage *)inputImage {
    CIFilter *desaturate = [CIFilter
        filterWithName:@"CIColorControls"];
    [desaturate setValue:inputImage forKey:kCIInputImageKey];
```

```
[desaturate setValue:@0.0f forKey:@"inputSaturation"];  
  
CIFilter *blur = [CIFilter  
    filterWithName:@"CIGaussianBlur"];  
[blur setValue:desaturate.outputImage  
    forKey:kCIIputImageKey];  
[blur setValue:@3.0f forKey:@"inputRadius"];  
  
CIFilter *inverted = [CIFilter  
    filterWithName:@"CIColorInvert"];  
[inverted setValue:blur.outputImage  
    forKey:kCIIputImageKey];  
  
CIFilter *blendDodge = [CIFilter  
    filterWithName:@"CIColorDodgeBlendMode"];  
[blendDodge setValue:inverted.outputImage  
    forKey:kCIIputBackgroundImageKey];  
[blendDodge setValue:desaturate.outputImage  
    forKey:kCIIputImageKey];  
  
CIFilter *blendBurn = [CIFilter  
    filterWithName:@"CIColorDodgeBlendMode"];  
[blendBurn setValue:blendDodge.outputImage  
    forKey:kCIIputImageKey];  
[blendBurn setValue:inputImage  
    forKey:kCIIputBackgroundImageKey];  
  
    return blendBurn.outputImage;  
}
```

The first filter converts the color image to a grayscale version by completely desaturating it.

In the second step, that grayscale image is blurred. The `CIGaussianBlur` filter is an expensive filter in terms of performance, so you'll want to pay close attention to performance and image size when using it.

Next, the blurred image is inverted. In the fourth step, the inverted, blurred image is blended with the original grayscale. If the inverted grayscale and the original grayscale were blended (without blurring), you'd end up with a completely white image. By using the blurred image, you get black lines where the blur has changed the pixels.



You can understand this intuitively if you consider that blurring a blue sky would have almost no visible effect. But, if you blur an image of a person in front of that blue sky, the blur will fuzzy around the edges of the head, the eyes, and anywhere that there was a hard edge.

This inversion and blurring, then blending, creates visual dissonance between those areas where the blurring made the most difference. More complex, and accurate, edge detection algorithms still rely on this principle.

At this point, you have black lines (on a white canvas) wherever there's an edge. Finally, you blend that with the original image. This colors those black lines with the original color.

As I said, the whole process is modeled after a Photoshop tutorial on creating a colored pencil effect from a photograph. The cool thing about these Core Image filters is that they are named after the processes found in image editors. So you can create video effects based on all the great photo editing tutorials out there for Photoshop.

Add the edge detection method to case 0 of the switch statement:

```
image = [self simpleEdgeDetection:image];
```

Build, run, and select the first segment control:



You can vary the radius on the blur filter in order to change the width of the lines in the final output (and get more or less noise, like what you're seeing on my shirt).

The old timey filter

In this filter chain, you're going to create a black and white image, add a vignette (darkening around the edges), and introduce a flicker effect. You'll create the flicker effect by varying the brightness of the image, as well as by introducing some movement into the frame.

You'll use `CIColorControls`, `CIVignette`, and you'll apply a transform to move the frame around. In order to create a more natural flicker, you'll use Perlin noise to generate a random number that controls the brightness and movement of the frame. There's an open source Objective-C implementation of Perlin noise created by Christopher Garrett and distributed under the MIT license. I've included these files in the sample project. None of the filters in this chain are new to iOS 6. All of these effects are supported in iOS 5.

To start, import this class into **ViewController.m**. Add this line at the top after the existing import statements:

```
#import "CZGPerlinGenerator.h"
```

Then add an instance variable to the class extension (the `@interface` section):

```
CZGPerlinGenerator *_perlin;
```

Next, add the initial setup of the Perlin generator to `viewDidLoad` (at the end):

```
_perlin = [CZGPerlinGenerator perlinGenerator];
_perlin.zoom = 100;
```

I won't cover the ins and outs of Perlin noise generation here. To give a quick justification for using it, Perlin noise will generate pseudo-random noise. However, unlike a simple random number generator, Perlin noise is smooth.

In this case, you'll use it to adjust the size of the vignette, the brightness, and the position of the frame. Using Perlin noise means that these adjustments will happen smoothly instead of jumping about haphazardly.

Note: Perlin noise has all kinds of applications. If you'd like to know more, here's the Wikipedia article:

http://en.wikipedia.org/wiki/Perlin_noise

In the above code, you set the zoom to 100. Raising this number will increase the speed at which the output noise changes.

Now to implement the `oldTimey` filter. Add this code after the edge detection filter:

```
-(CIImage *)oldTimey:(CIImage *)inputImage {
    CFAbsoluteTime timeNow = CFAbsoluteTimeGetCurrent();

    float first = [_perlin perlinNoiseX:sin(timeNow) * 1000.0
                  y:10.0 z:10.0 t:10.0];
    float second = [_perlin perlinNoiseX:cos(timeNow) * 1000.0
                  y:105.0 z:10.0 t:10.0];
    float third = [_perlin perlinNoiseX:sin(timeNow) * 1000.0
                  y:200.0 z:10.0 t:10.0];

    CIFilter *blackandwhite = [CIFilter
        filterWithName:@"CIColorControls"];
    [blackandwhite setValue:@0.1f forKey:@"inputSaturation"];
    [blackandwhite setValue:@(first * 0.05)
        forKey:@"inputBrightness"];
    [blackandwhite setValue:inputImage forKey:kCIInputImageKey];

    CIFilter *vignette = [CIFilter
        filterWithName:@"CIVignette"];
    [vignette setValue:blackandwhite.outputImage
        forKey:kCIInputImageKey];
    [vignette setValue:@10 forKey:@"inputRadius"];
    [vignette setValue:@(third + 2) forKey:@"inputIntensity"];
```

```
CGAffineTransform transform =
    CGAffineTransformMakeTranslation(first * 1.0,
        1.0 + (second * 10));
CIIImage *returnImage = [vignette.outputImage
    imageByApplyingTransform:transform];

    return returnImage;
}
```

First you create three random numbers using the Perlin noise generator. This generator takes four arguments and returns a random number between -1 and 1. You seed the x dimension with the sine or cosine of the current time in order to generate a rotating value. To be honest, I just messed with the inputs for the Perlin generator until I got the “look” I wanted.

Next, you create the `CIColorControls` filter. This changes the image to be almost grayscale. It also pulsates the brightness using the `first` random number.

Next, you create a `CIVignette` filter to darken the edges. This further adds to the old look. You use a random number here, as well, to cause unpredictable adjustments to the vignette intensity.

Finally, you use the built-in `CIIImage` method, `imageByApplyingTransform`. This method takes a `CGAffineTransform` and applies it to a `CIIImage`. There's also a `CIAffineTransform` filter that would work, but I like the Core Image method better because it's fewer lines of code. The transform moves the image up and down to simulate an old projector.

There's one more thing you must do – can you guess? Ha! Of course you can. Add the method to the switch statement, this time to `case 1`:

```
image = [self oldTimey:image];
```

Build and run. If you select the second segment, you should have something like this:



Take off your hat, turn on the wireless (radio), and pretend you're not holding a device more powerful than all the computers of your grandparents' era put together!

There's a new Face Detection in town!

We've saved the best, and most involved, for last. ☺

In iOS 5, Apple added Face Detection to Core Image. This API was cool, but if you tried to use it with live video, you may have wished it was a little faster.

Well, there's a new face detection API in iOS 6, and it is faster. Whether it's fast enough, you will have to decide for yourself.

The new API doesn't rely on Core Image. Instead, it's part of AVFoundation. It's part of the camera capture pipeline and you'll need a new `AVCaptureOutput` object and a new delegate callback to handle the data.

The object that you'll need to add as an output to your session is called `AVCaptureMetadataOutput`. This operates like your `AVCaptureVideoDataOutput` object. You'll set your `viewController` class as the delegate, and implement the `-captureOutput:didOutputMetadataObjects:fromConnection:` method. This method is called every time you have new face objects detected.

The callback provides an array, one dictionary per object detected. The array contains `AVMetadataFaceObjects`. An `AVMetadataFaceObject` is a subclass of `AVMetadataObject`, currently the only one. If there are other types of metadata

objects detectable in the future, this same API will be used for them as well, but at the moment, it's only faces (just like the Core Image API).

Each instance of `AVMetadataFaceObject` has a number of properties:

- The **ID** of the face (as long as a face continues to be visible from frame to frame, it will retain the same ID number).
- The **bounds** of the detected face.
- The **time** the face was detected.
- The **roll and yaw** of the face.

There are a few things to note. First, while the face detection can keep track of the same face from one frame to the next, if a face leaves the frame and returns, it will get a new ID.

Second, the face detection occurs in the camera's default orientation, regardless of the actual position of the device. What this means is that you will get a roll angle of the face relative to that orientation.

The yaw is a measure of whether the face is looking directly at the camera or to the left or right. The tolerance is 130 degrees. If a face is turned more than this, it won't be detected. There is no pitch for face detection (the face looking upwards or downwards).

Let's return to your code. First, add the `AVCaptureMetadataOutputObjectsDelegate` protocol to the `@interface` section:

```
@interface ViewController : UIViewController
<AVCaptureVideoDataOutputSampleBufferDelegate,
 AVCaptureMetadataOutputObjectsDelegate>
```

Next, add the following output to the `_session` in `viewDidLoad` before the `[_session commitConfiguration]` line:

```
AVCaptureMetadataOutput *metaDataOutput =
    [[AVCaptureMetadataOutput alloc] init];
[metaDataOutput setMetadataObjectsDelegate:self
    queue:dispatch_get_main_queue()];
if ([_session canAddOutput:metaDataOutput]) {
    [_session addOutput:metaDataOutput];
} else {
    NSLog(@"Can't add metadata output");
}
NSLog(@"Available metadata object types - %@", metaDataOutput.availableMetadataObjectTypes);
```

The above code creates an `AVCaptureMetadataOutput` object instance, checks if it can be added to the session (and outputs an error if it cannot), and once it's added to the session, logs the available metadata object types.

Do note that face detection might not be available on all hardware. (In fact, it appears that it might only be available on iPhone 4S and iPad 3.) So if you don't see any available metadata object types in the output, the rest of this section probably will not work for you. ☺

Finally, add the delegate method. For now you'll just log the output:

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputMetadataObjects:(NSArray *)metadataObjects
fromConnection:(AVCaptureConnection *)connection {
    NSLog(@"%@", metadataObjects);
}
```

Build and run. If you put your face, or a picture of a face in front of the camera, you should get some output like the following (if your device is supported):

```
All Output ▾
"AVMetadataFaceObject: 0x1fdb0530> id 1, bounds {0.6,0.5 0.2x0.3}, roll 180.0, yaw 0.0, time 208889868865583"
)
2012-09-05 16:08:58.072 FilterMe[9619:907] objects (
    "<AVMetadataFaceObject: 0x208a1380> id 1, bounds {0.6,0.5 0.2x0.3}, roll 180.0, yaw 0.0, time 208889901429500"
)
2012-09-05 16:08:58.106 FilterMe[9619:907] objects (
    "<AVMetadataFaceObject: 0x208a6460> id 1, bounds {0.6,0.5 0.2x0.3}, roll 180.0, yaw 0.0, time 208889934791625"
)
2012-09-05 16:08:58.139 FilterMe[9619:907] objects (
    "<AVMetadataFaceObject: 0x1fdBeef0> id 1, bounds {0.6,0.5 0.2x0.3}, roll 180.0, yaw 0.0, time 208889968155083"
)
2012-09-05 16:08:58.173 FilterMe[9619:907] objects (
    "<AVMetadataFaceObject: 0x1fd947d0> id 1, bounds {0.6,0.5 0.2x0.3}, roll 180.0, yaw 0.0, time 208890001518208"
)
2012-09-05 16:08:58.206 FilterMe[9619:907] objects (
    "<AVMetadataFaceObject: 0x1fd8bfc0> id 1, bounds {0.6,0.5 0.2x0.3}, roll 180.0, yaw 0.0, time 208890034882541"
)
2012-09-05 16:08:58.239 FilterMe[9619:907] objects (
    "<AVMetadataFaceObject: 0x1fd94020> id 1, bounds {0.6,0.5 0.2x0.3}, roll 180.0, yaw 0.0, time 208890068243125"
)
(Uldb)
```

In this next filter chain, you'll use whatever faces have been detected in this metadata callback and apply a distortion to each one.

Note: You may notice that if you try to detect a bunch of faces (like four or more), you'll begin to see some slowdown in the video frames per second. While this new API is much faster than the Core Image face detector, it still has limitations.

You need to store the information you're getting from the metadata callback so you can deal with it in the filter chain. Add the following instance variable to the `@interface` section:

```
NSArray *_faces;
```

Next, alter `captureOutput:didOutputMetadataObjects:` by replacing the log statement with this code:

```
NSMutableArray *newAr = [NSMutableArray array];
for (AVMetadataFaceObject *fo in metadataObjects) {
    CGPoint center = CGPointMake(fo.bounds.origin.x +
        fo.bounds.size.width/2, fo.bounds.origin.y +
        fo.bounds.size.height / 2);
    float width = fo.bounds.size.width;
    NSDictionary *d = @{@"position" : [NSValue
        valueWithCGPoint:center],
        @"size" : @(width)};
    [newAr addObject:d];
}
_faces = (NSArray *)newAr;
```

This is pretty easy code. You create a distortion about the size of the face on each face in the incoming video frame. This method constructs a dictionary that contains the position and size of the face.

You create the center `CGPoint` by finding the origin point of the `CGRect` that frames the face, and then calculating the center point of that bounding rect.

Now create the method that applies the distortions to the faces:

```
-(CIImage *)makeFaces:(CIImage *)inputImage {
    // 1
    CIFilter *filter;
    // 2
    for (NSDictionary *f in _faces) {
        // 3
        CIFilter *distortionFilter = [CIFilter
            filterWithName:@"CIPinchDistortion"];
        int indx = [_faces indexOfObject:f];
        if (indx == 0) {
            [distortionFilter setValue:inputImage
                forKey:kCIInputImageKey];
        } else {
            [distortionFilter setValue:filter.outputImage
                forKey:kCIInputImageKey];
        }
        // 4
        CGPoint center =
            [[f objectForKey:@"position"] CGPointValue];
        center = CGPointMake(640.0 * center.x,
            480 - 480.0 * center.y);
        [distortionFilter setValue:
            [CIVector vectorWithCGPoint:center]
            forKey:@"inputCenter"];
```

```
// 5
float size = [[f objectForKey:@"size"] floatValue] *
    640.0 / 2.0;
[distortionFilter setValue:@(size)
    forKey:@"inputRadius"];
// 6
[distortionFilter setValue:@1.0 forKey:@"inputScale"];
// 7
filter = distortionFilter;
}
// 8
CIImage *returnImage;
if (filter) {
    returnImage = filter.outputImage;
} else {
    returnImage = inputImage;
}
// 9
return [returnImage imageByCroppingToRect:
    [inputImage extent]];
}
```

In this code block, you need to create as many `CIPinchDistortion` filters as you have face objects in your `faces` array. To chain filters together in a block, you have to do a several things:

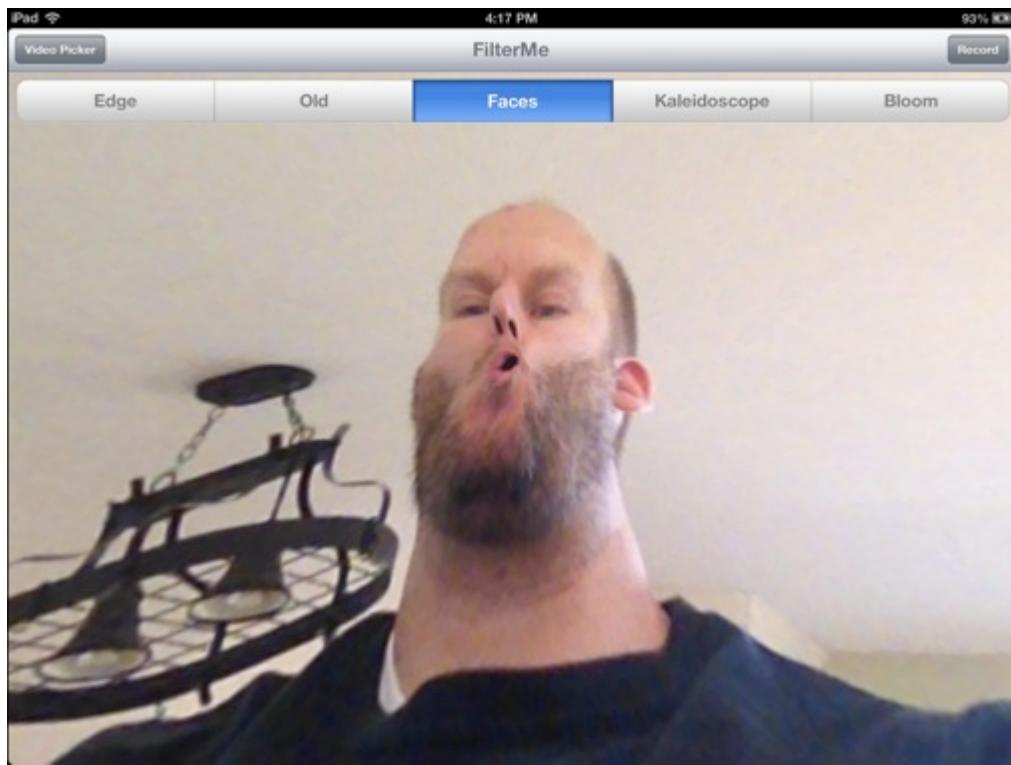
1. Create a reference to a `CIFilter` that you'll use throughout the `for` loop in order to chain the output from one filter to the next.
2. Start the `for` loop, running it once for each face in the `faces` array.
3. Create the filter first. You need to pass the `inputImage` `CIImage` the first run through the loop. Every other time, you pass the `outputImage` property of the filter created in the previous loop.
4. Start setting the parameters for this filter. The first is the `inputCenter` property, which is the location of the pinch. You'll use the `position` key from the dictionary to get the face's center. However, the face detection has a coordinate system that ranges from 0.0 to 1.0. Also, it's vertically flipped from the coordinate system that Core Image uses, so you have to scale it up, and flip it. You're using hard-coded values here, but in a real app where you'd be using variable resolutions on your incoming images, you'd need to use those values.
5. Next you set the size, or the `inputRadius`. You use the value you store under the `size` key in the dictionary. This value, if you remember, is the width of the bounding box around the face. You have to scale this value up as well (same coordinate system, ranging from 0.0 to 1.0). Also, since the radius is half the diameter of a circle, you cut it in half in order to generate a circle effect about the size of the face.

6. The last parameter to set is `inputScale`. Just set it to 1.0 here. You could use other values to get more or less effect.
7. The last step in the loop is setting the filter pointer to the `distortionFilter` pointer so that in the next iteration of the loop, you can get the `.outputImage` property for the next filter in the chain.
8. Once you are out of your loop, you check to see if the loop ran at all. It's possible that there are no faces in the array. If that's the case, you pass the `inputImage` parameter as the output, instead of the result of the final filter in the chain (`filter.outputImage`).
9. Finally, return the image after cropping it to fit the size of the original input image.

Back to the code, add the switch statement for case 2:

```
image = [self makeFaces:image];
```

Build and run, select the Faces button, and you'll see something like this (if your device supports face detection – otherwise, you'll see the normal image):



Wazzup, homies?!

You may try to detect multiple faces with this filter. However, the pinch distortion affects the entire frame, so while technically it can detect multiple faces, you won't be able to see them because they will have been distorted out of the visible frame.

If you want to have fun with multiple faces, try the twirl distortion filter. It only affects the radius that you give it. In my tests, I found that at three or four faces, the whole system begins to slow down.

That's it for filters. You are done writing Core Image filter chains, and, in fact, you're done with Core Image code for this tutorial. The only thing we have left to do in this project is record the output of these chains.

Recording video

Now that you've built all five filters, it's time to circle back to AVFoundation. I'm going to introduce a set of classes that you'll use to write `CVPixelBufferRef` (video) and `CMSampleBuffers` (could be video or audio, but you'll use it for audio) to a file.

So far, you've worked exclusively with AVFoundation's `AVCaptureSession` classes. These classes are designed to capture data from an input device (camera, mic, etc.) and then push that information to an output. You've used the `AVCaptureVideoDataOutput` class (and later you'll use its counterpart, `AVCaptureAudioDataOutput`) to send the raw data to a callback method.

There are classes that perform the inverse operations: they take data buffers and write them to a media file type. These classes can write to a wide variety of video and audio data formats. They can also write other types of media data, like subtitles, metadata, and more to media files.

Here are brief descriptions of the classes you'll use to write audio and video data to a file:

- **AVAssetWriter**. This class is kind of like `AVCaptureSession`, in that it will control the overall process. You'll create `AVAssetWriterInput` subclasses and add them to the `AVAssetWriter` instance.
- **AVAssetWriterInput**. You'll create different inputs for each audio and video channel you have. These inputs will be configured to the type of data that they'll receive. In the case of the video data, for example, you'll configure the size in pixels and compression codec for the video to be written.
- **AVAssetWriterInputPixelBufferAdaptor**. This object allows you to append pixel buffers (instead of `CMSampleBuffers`) to the `AVAssetWriter`. Since you'll be using pixel buffers as the output of your Core Image pipeline, you'll need this object as well.

As the data from your capture callback methods arrive, you'll append it to either the video or audio input object for your `AVAssetWriter`. This is the push method of writing data, and is appropriate for real-time incoming data. Each input has a `readyForMoreMediaData` property that you will check before adding each data sample, in order to make sure that the writer has finished the previous buffer and is ready for more.

Note: An alternative is the pull method of writing data. You can set a callback method on the input object that pulls the next buffer of data as soon as it's ready to receive it.

This method is more suitable for non-real-time data sources, such as an input video file. This way, as soon as the writer is ready (and it can be faster or slower than real time), it will process and pull in the next sample. We'll not be covering that method in this chapter.

Once the `AVAssetWriter` object has all the appropriate input objects configured and added, you start the writing process. And once the process has started, there are limits to which configuration options can be changed.

There are two calls that need to be sent to the `AVAssetWriter` in order to initiate a writing session. The first, `startWriting`, gets the writer ready. The second, `startSessionAtSourceTime:`, sets up the timestamp of the first sample.

Video and audio data is simply a data buffer at a time interval. Video is a series of images presented (ideally) every thirtieth or sixtieth of a second. Audio is similar.

This means when you write media data, you must provide the data and a time interval for when that data should present itself. You'll see that when you append pixel buffer data, you also need to supply a timestamp for that data.

`startSessionAtSourceTime:` gives the writer a start time from which to calculate the intervals for the subsequent data.

That's a quick overview of writing buffers with an `AVAssetWriter`. As with the other AVFoundation material that we're covering here, it's just skimming the surface of a very deep topic.

Time to get back to the code!

Making the Record button, record!

The first thing you need is a handful of new instance variables. Add these lines to the `@interface` section in **ViewController.m**:

```
AVAssetWriter *_assetWriter;
AVAssetWriterInput *_assetWriterAudioInput;
AVAssetWriterInputPixelBufferAdaptor
    *_assetWriterPixelBufferInput;
BOOL _isWriting;
CMTime currentSampleTime;
```

Now initialize the `_isWriting` Boolean. Add this to the end of `viewDidLoad`:

```
_isWriting = NO;
```

Next, add two helper methods, `movieURL` and `checkForAndDeleteFile`:

```
- (NSURL *)movieURL {
    NSString *tempDir = NSTemporaryDirectory();
    NSString * urlString = [tempDir
        stringByAppendingPathComponent:@"tmpMov.mov"];
    return [NSURL fileURLWithPath:urlString];
}

-(void)checkForAndDeleteFile {
    NSFileManager *fm = [NSFileManager defaultManager];
    BOOL exist = [fm fileExistsAtPath:[self
        movieURL].absoluteString];
    NSError *err;
    if (exist) {
        [fm removeItemAtURL:[self movieURL] error:&err];
        NSLog(@"file deleted");
        if (err) {
            NSLog(@"file remove error, %@", err.localizedDescription );
        }
    } else {
        NSLog(@"no file by that name");
    }
}
```

Both these methods are convenience methods that deal with the management of the file. You'll be saving your completed videos to the Photo Album. However, you cannot write the file directly to that location, so you'll use a temporary file location during the writing process. This first method just gets a path location in the app's temporary directory with a filename of **tmpMov.mov**, and returns the `NSURL` for that path.

The second method checks for the `tmpMov.mov` file, and deletes it if it exists. `AVAssetWriter` will fail if the file already exists – so you'll be calling this delete method before you initialize the `AVAssetWriter`. To delete the file, the method calls `removeItemAtURL`. It then logs to the console whether or not the file existed, and whether it was successfully deleted.

The next step is to create the `AVAssetWriter`. An `AVAssetWriter` exists in order to write a single file, so every time you wish to record a new video, you'll call this creation method again.

Add this method::

```
- (void)createWriter {
    // 1
    [self checkForAndDeleteFile];
    // 2
    NSError *error;
    _assetWriter = [[AVAssetWriter alloc] initWithURL:
        [self movieURL] fileType:AVFileTypeQuickTimeMovie
        error:&error];
    if (error) {
        NSLog(@"Couldn't create writer, %@", error.localizedDescription);
        return;
    }
    // 3
    NSDictionary *outputSettings = @{
        AVVideoCodecKey : AVVideoCodecH264,
        AVVideoWidthKey : @640,
        AVVideoHeightKey : @480
    };
    AVAssetWriterInput *assetWriterVideoInput =
        [AVAssetWriterInput
            assetWriterInputWithMediaType:AVMediaTypeVideo
            outputSettings:outputSettings];
    // 4
    assetWriterVideoInput.expectsMediaDataInRealTime = YES;
    // 5
    NSDictionary *sourcePixelBufferAttributesDictionary =
        @{@"kCVPixelBufferPixelFormatTypeKey":
            @(kCVPixelFormatType_32BGRA),
        (id)kCVPixelBufferWidthKey: @640,
        (id)kCVPixelBufferHeightKey: @480};
    _assetWriterPixelBufferInput =
        [AVAssetWriterInputPixelBufferAdaptor
            assetWriterInputPixelBufferAdaptorWithAssetWriterInput:
                assetWriterVideoInput
            sourcePixelBufferAttributes:
                sourcePixelBufferAttributesDictionary];
    // 6
    if ([_assetWriter canAddInput:assetWriterVideoInput]) {
        [_assetWriter addInput:assetWriterVideoInput];
    } else {
        NSLog(@"can't add video writer input %@", assetWriterVideoInput);
    }
}
```

Note: My setup code is based heavily on the GPUImage project, so some of these dictionary names and structure are similar to the code from that project. I recommend you check it out if you are interested in video filtering/recording.

Here's what the above method does:

1. Calls your delete method in order to get rid of the temporary recording file, if one exists.
2. Initializes the `AVAssetWriter` object. The initialization takes a URL to a video file, the type of file being recorded, and an error parameter. The file type parameter supports a wide variety of audio and video recording formats.
3. Creates an `outputSettings` `NSDictionary` and supplies it to an `AVAssetWriterInput` object. This `AVAssetWriterInput` object can be one of a variety of types, including video, audio, subtitles, closed caption, metadata, etc.

The dictionary supplies the options for this input. In this case, you're simply specifying the size of the video and the codec. One other option available in this settings dictionary is the scaling mode.

4. Sets `expectsMediaDataInRealTime` to YES. This is required for real-time encoding from a camera capture source.
5. Creates a settings dictionary and provides it to an `AVAssetWriterInputPixelBufferAdaptor` object. This is used to append `CVPixelBuffers` to a video file that is being recorded. Because you'll be providing pixel buffers, you need this adaptor object. You initialize this object with an instance of `AVAssetWriterInput`.

The dictionary sets the pixel buffer size, as well as the pixel format. The other settings in this dictionary all serve to describe the format and size of the incoming buffer.

6. Finally, adds this input, the adaptor, to the `AVAssetWriter` object. You use the `canAddInput` method to determine if this object can be added to the writer object. This can fail if the type or settings aren't compatible. If it is compatible, it's added to the `assetWriter`.

That finalizes your setup. There are a few more things you need to do to start recording the video, but you're getting close.

Modify the `captureOutput:didOutputSampleBuffer:` callback method to do some writing steps for when the `_isWriting` flag is set to YES. Put this code after the end of the `switch()` statement:

```
// 1  
currentSampleTime =  
CMSampleBufferGetOutputPresentationTimeStamp(sampleBuffer);  
// 2
```

```
    if (_isWriting &&
_assetWriterPixelBufferInput.assetWriterInput.isReadyForMoreMediaD
ata) {
    // 3
    CVPixelBufferRef newPixelBuffer = NULL;
    CVPixelBufferPoolCreatePixelBuffer(NULL,
        [_assetWriterPixelBufferInput pixelBufferPool],
        &newPixelBuffer);
    // 4
    [_coreImageContext render:image
        toCVPixelBuffer:newPixelBuffer
        bounds:CGRectMake(0, 0, 640, 480) colorSpace:NULL];
    // 5
    BOOL success = [_assetWriterPixelBufferInput
        appendPixelBuffer:newPixelBuffer
        withPresentationTime:currentTime];
    // 6
    if (!success) {
        NSLog(@"Pixel Buffer not appended");
    }
    // 7
    CVPixelBufferRelease(newPixelBuffer);
}
```

Here's a step-by-step breakdown of the above code:

1. Stores the sample time for this sample buffer. This will be needed later.
2. Tests if the writing Boolean is on, and if the asset writer is ready for data.
3. Sets up a new pixel buffer to write to. This is necessary for most filters.
Sometimes you can use the existing pixel buffer and write over the contents already there, but in many cases, you'll get strange artifacts by doing this.

The `CVPixelBufferPoolCreatePixelBuffer` call creates a pixel buffer based on a pixel buffer pool. Using a pool is more efficient than creating a pixel buffer by the more direct method.

Also, the `[_assetWriterPixelBufferInput pixelBufferPool]` call uses the settings that you passed to the `_assetWriterPixelBufferInput` object to create the pixel buffer. Otherwise, you'd need to set up all the parameters for your pixel buffer again.

However, be aware that if you call `[_assetWriterPixelBufferInput pixelBufferPool]` before the asset writer has started writing, you'll get NULL back for the pixel buffer pool, and the pixel buffer won't be created. The whole process fails (I know this from painful experience).

Timing is critical to audio and video data. What you have is a series of frames, and the `assetWriter` needs to know when to show them. There are a variety of ways to do this.

For example, you could programmatically create the time value and place each new buffer at 1/30 of a second after the previous one. This would create a perfect 30fps video. However, if the buffers aren't actually coming in that quickly (maybe the filter chain takes longer to render the image, or maybe the camera callback isn't running at full speed), the resulting video won't represent the timing realistically.

The sample buffer comes in with a time stamp on it, and you can use the `CMSampleBufferGetOutputPresentationTimeStamp` to get that `cmtTime` value. That's what you pass into the `_assetWriterPixelBufferInput` object to append the pixel buffer each frame.

Note: The Core Video and Core Media frameworks provide the methods for dealing with `CMSampleBuffers` and `cvPixelBuffers`. There is a long list of methods that can be used on a `CMSampleBufferRef` to get information about the data that is in the `CMSampleBuffer`. Among other things, you can get the size of the pixel data, timing information, the configuration of the incoming audio, etc.

4. Once you have your pixel buffer set up, you render into it.
5. The `appendPixelBuffer:withPresentationTime:` method returns a Boolean representing whether it successfully wrote the frame.
6. If appending the pixel buffer failed, you output that to the console.
7. Releases the pixel buffer you created. A pixel buffer is not controlled by ARC (it's a C object) – you must keep track of it yourself, because you created it.

Even though you have recorded the rendered image to the video file, you still need to render it to the screen. The pre-existing code in this method accomplishes this. It calls `drawImage:inRect:fromRect:` to display the contents to the render buffer.

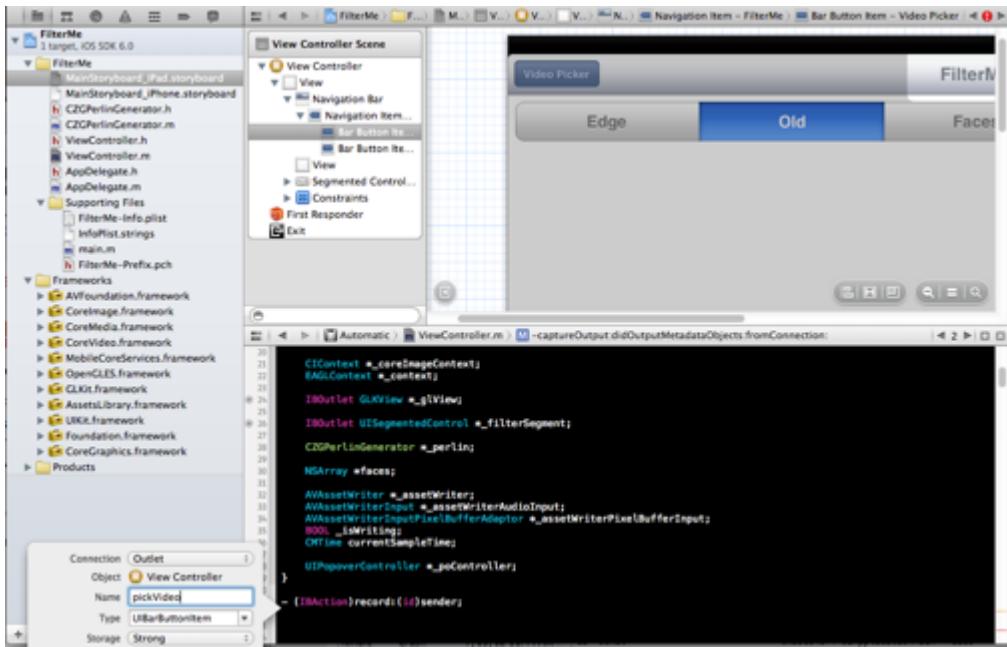
Hooking up the Record button

All that's left is hooking up your button.

Go to Interface Builder by clicking on the storyboard file (whichever you're using; my screenshots are from the iPad version).

Open the Assistant Editor and navigate to **ViewController.m** like you've done before.

Click and drag from the Record button to the `@interface` section (but remember to drag to a line below the closing curly brace). Change "Connection" from outlet to action, and name this method "record" like you see here:



If you want your app to work on both iPhone and iPad, repeat the process (except just link instead of creating a new method) in the other storyboard file.

Now replace the empty `record` method at the end of the file with the following:

```

    -(IBAction)record:(id)sender {
        UIBarButtonItem *button = (UIBarButtonItem *)sender;
        if (!_isWriting) {
            [self createWriter];
            _isWriting = YES;
            button.title = @"Stop";
            [_assetWriter startWriting];
            [_assetWriter
                startSessionAtSourceTime:currentSampleTime];
        } else {
            _isWriting = NO;
            button.title = @"Record";
            [_assetWriter finishWriting];
        }
    }
}

```

First, you get a reference to the `UIBarButtonItem`, because you'll change the title based on whether the recording needs to be started or stopped.

If the device isn't currently recording, the first item of business is setting up the writer. Then you set the flag to indicate that you're recording. Finally, you update the button title accordingly.

The next two methods start the recording process. First, you always have to call `startWriting`. But that isn't all – you also have to call `startSessionAtSourceTime:`. This gets the latest time from the callback, `currentSampleTime`. If you don't call both methods and you try to call `appendPixelBuffer`, it won't work. You'll end up with a video that has no frames in it.

That starts the video writing process. What stops it? You only have to call `finishWriting`.

And that's the whole video recording chain. You've successfully created a video recording device!

But wait, if you run the app and record now, you will be recording a video, but you'll only be able to see the button change. You have no way of looking at these videos.

One more thing will fix that. I've already included the `AssetsLibrary` framework, so you just need to add the `#import` statement at the top of **ViewController.m**:

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

Now add this method to move the recorded video into the photo album:

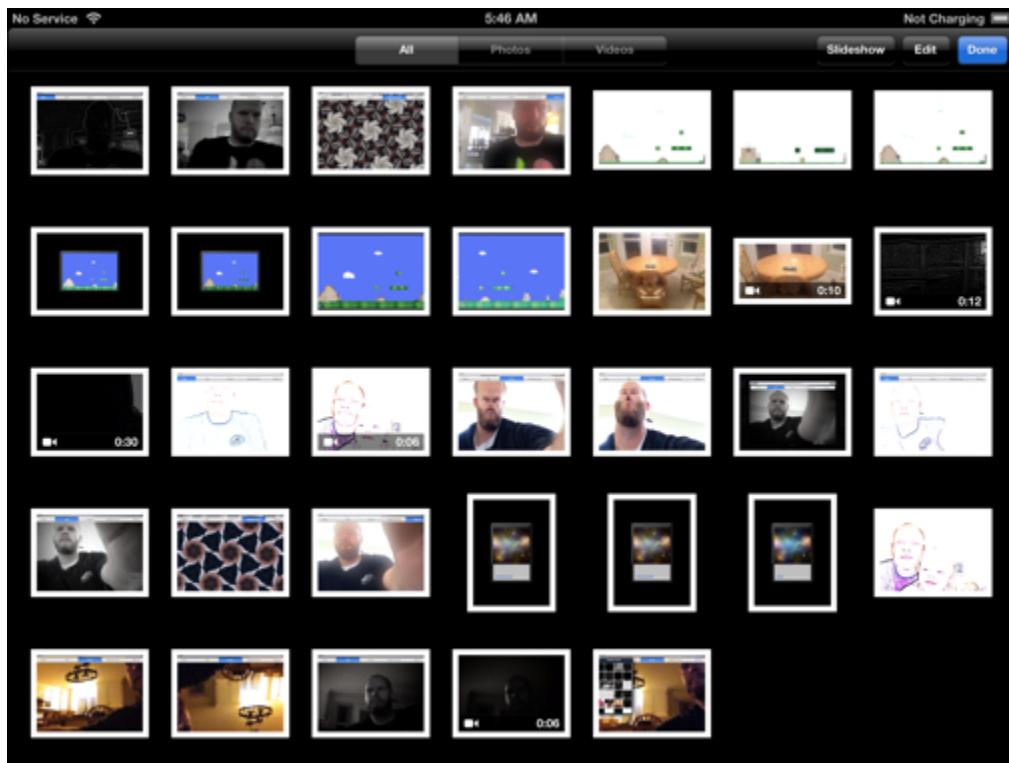
```
- (void)saveMovieToCameraRoll {
    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];
    [library writeVideoAtPathToSavedPhotosAlbum:[self movieURL]
        completionBlock:^(NSURL *assetURL, NSError *error) {
            if (error) {
                NSLog(@"Error %@", [error localizedDescription]);
            } else {
                [self checkForAndDeleteFile];
                NSLog(@"finished saving");
            }
        }];
}
```

This code should be easy to read, even if you're not familiar with the `AssetsLibrary`. You are using a convenience method to get the movie location and save it to the photo album. The completion block deletes the temporary file if the save was successful; otherwise it logs the output.

Now, just add this to the `record` method after the call to `finishWriting`:

```
[self saveMovieToCameraRoll];
```

Build and run. Pick a filter and press the record button. If you want to get crazy, change the filters while the video is still recording. Then press Stop. Go to your camera roll and you will have a video of what you just saw. EPICALLY AWESOME!!



What about the sound? Your videos are silent movies. Time to fix it.

Sound on, sound off

Video and audio are separate pipelines (for one, you're not doing anything with the audio, just passing it to the asset writer). Also, they have separate inputs (we haven't created an audio input or microphone device in our capture session, yet). They will, however, both use the same `didOutputSampleBuffer` delegate method.

You need to do three things to add audio to this app:

- Create an audio input device and input object in the capture session.
- Create an audio input in the `_assetWriter` object.
- Take different approaches to handling the audio and video `CMSSampleBuffers` coming in from the `didOutputSampleBuffer` method.

You'll do these tasks one at a time.

Start by adding the following code to `viewDidLoad`, right before the

```
AVCaptureVideoDataOutput *videoOutput = [[AVCaptureVideoDataOutput alloc] init]
```

```
AVCaptureDevice *mic = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeAudio];
AVCaptureDeviceInput *audioDeviceInput =
```

```
[AVCaptureDeviceInput deviceInputWithDevice:mic
    error:&err];
if (err) {
    NSLog(@"Video Device Error %@", [err
        localizedDescription]);
}
[_session addInput:audioDeviceInput];
AVCaptureAudioDataOutput *audioOutput =
    [[AVCaptureAudioDataOutput alloc] init];
[audioOutput setSampleBufferDelegate:self
    queue:dispatch_get_main_queue()];
[_session addOutput:audioOutput];
```

These methods look a lot like those you used to set up the camera. The default device for the `AVMediaTypeAudio` is the built-in mic, or headphones if they are plugged in.

As with the camera, you need:

- A device.
- An input object.
- An output object.

The output object needs to set a delegate and queue for the data to be processed. In this case you're using the same delegate, which means that you'll be receiving the data in the same callback method as the video data. You need to set the delegate protocol for the class, so change the `@interface` line to the following:

```
@interface ViewController ()  
<AVCaptureVideoDataOutputSampleBufferDelegate,  
AVCaptureAudioDataOutputSampleBufferDelegate,  
AVCaptureMetadataOutputObjectsDelegate>
```

Now you can receive the sample buffers in this class.

Next, add this to the end of `createWriter`:

```
_assetWriterAudioInput = [AVAssetWriterInput  
    assetWriterInputWithMediaType:AVMediaTypeAudio  
    outputSettings:nil];
if ([_assetWriter canAddInput:_assetWriterAudioInput]) {  
    [_assetWriter addInput:_assetWriterAudioInput];
    _assetWriterAudioInput.expectsMediaDataInRealTime = YES;
}
```

This input looks a lot like your others. You're passing `nil` into the options for the `AVAssetWriterInput` because the default settings are satisfactory here.

Finally, you need to deal with the incoming audio buffers in `captureOutput:didOutputSampleBuffer:`. Because both audio and video sample buffers are going to be running through the same method, you need to distinguish between them and handle them differently. The place to do this is right at the beginning of the method!

Add this code to the beginning of the method and enclose all the existing code in the `else` block of the `if` statement:

```
NSString *outputClass = NSStringFromClass(
    [captureOutput class]);
if ([outputClass
    isEqualToString:@"AVCaptureAudioDataOutput"]) {
    if (_isWriting &&
        _assetWriterAudioInput.isReadyForMoreMediaData) {
        BOOL succ = [_assetWriterAudioInput
            appendSampleBuffer:sampleBuffer];
        if (!succ) {
            NSLog(@"audio buffer not appended");
        }
    }
} else {
    // All the code you had in the method before this addition
}
```

There are a couple ways of detecting which `AVCaptureDataOutput` (audio or video) is being used. Here you use `NSStringFromClass` to get the name of the class, and then check if it's the `AVCaptureAudioDataOutput` class.

Alternatively, you could have made the `AVCaptureAudioDataOutput` (or its video counterpart) an instance variable, and checked the `captureOutput` variable from the callback method to see if it matched.

If it is audio, all you need to do is check whether you're currently writing to a file, and then call `appendSampleBuffer` on your `_assetWriterAudioInput`.

Everything that was previously in this method should now be encased in the `else` block.

If you build and run now, you should be able to record the audio as well as the video.

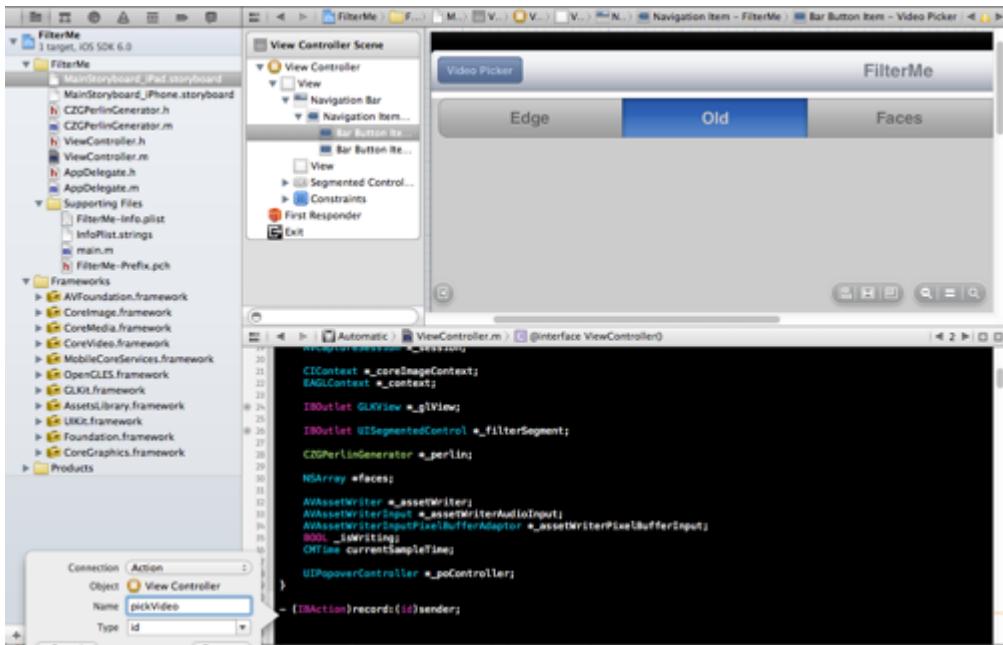


I'm sad because none of you can hear what I'm saying.

Final move: the picker

You are all but finished! The last item to add to this project is a media picker that will let you watch these videos in a modal view. You're not going to actually use it for picking media, but it's an easy way to view the videos you've recorded without having to jump out of the app and go to the Camera or Photos app.

First, connect the Video Picker button to a method called `pickvideo`, the same way that you created a method for the Record button. Make sure that you change the "Connection" from Outlet to Action (a mistake I make almost every time I create an `IBAction` method).



Now you need a popover controller instance variable. Add this to the @interface section in **ViewController.m**:

```
UIPopoverController *_poController;
```

Here's the code to launch the picker from the Video Picker button – add it to the empty `pickvideo` method implementation:

```
if (_isWriting) {
    return;
}
[_session stopRunning];
if (_poController.isPopoverVisible) {
    [_poController dismissPopoverAnimated:YES];
} else {
    if ([UIImagePickerController isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeSavedPhotosAlbum])
    {
        UIImagePickerController *imagePicker =
            [[UIImagePickerController alloc] init];
        imagePicker.delegate = self;
        imagePicker.sourceType =
            UIImagePickerControllerSourceTypeSavedPhotosAlbum;
        imagePicker.mediaTypes =
            @[(NSString *)kUTTypeMovie];
        imagePicker.allowsEditing = YES;
        if (UI_USER_INTERFACE_IDIOM() ==
            UIUserInterfaceIdiomPad) {
```

```
    _poController = [[UIPopoverController alloc]
        initWithContentViewController:imagePicker];
    _poController.delegate = self;
    [_poController
        presentPopoverFromBarButtonItem:sender
        permittedArrowDirections:
            UIPopoverArrowDirectionDown
        animated:YES];
} else {
    [self presentViewController:imagePicker
        animated:YES completion:^{
        [_session startRunning];
    }];
}
}
```

You'll notice that you get a few errors and warnings when you add this code. You need to add three new delegate protocols to your `@interface`, `UIImagePickerControllerDelegate`, `UINavigationControllerDelegate`, and `UIPopoverControllerDelegate`. But you don't need to implement any methods for the delegates, since you're only using the picker to get a look at what you've just recorded. WOOT!

You'll see the issue is with the `kUTTypeMovie` constant, which is found in the `MobileCoreServices` framework. You need to import that framework by adding this line to **ViewController.m**:

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

Now you should have silenced all those helpful (or pesky, depending on your disposition) errors and warnings. So let's go through the new code.

The first thing you want to do is check whether you're recording. AVFoundation will have a hard time showing and recording video at the same time. So you need to make sure that no recording is going on.

If you're not recording, you can proceed, but you need to stop the capture. If you want to know why, go ahead and try it without stopping the capture and see what happens. Try watching a video, and then go back and try to record something. It'll be blank. So, you stop the capture.

Next, you check to see if the popover is already up, and if it is, you dismiss it.

If all those conditions are satisfied, then you can create your image picker. You check to see if you have anything to pick from, because an empty picker is just sad.

The next four lines create and configure the picker. The source type tells the picker what kind it is. There is another type of picker that you can use to take video directly from that camera, for example. You just want a list of videos, so you're using `UIImagePickerControllerSourceTypeSavedPhotosAlbum` here.

Setting the media type to `kUTTypeMovie` means that it won't show you any photos, just videos, which is appropriate for your use.

Finally, if you are on iPad, you wrap the `UIImagePickerController` in a popover controller, set its delegate, and present it from the Video Picker button.

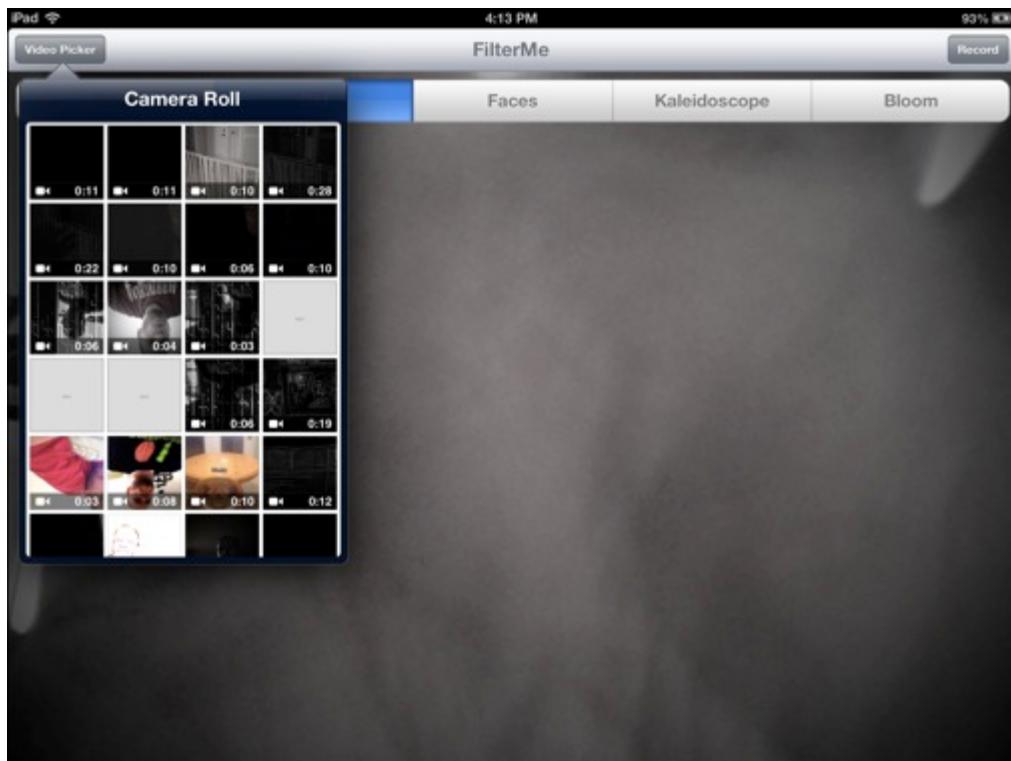
On the other hand, since this is a universal app, you might be on the iPhone or iPod touch. In this case, you don't have the option of using a popover controller. So you simply present the view using the new iOS 6

`presentViewController:animated:completion:` method that replaces the older `presentModalViewController:` method.

The completion block in the above method restarts the capture session. Of course, you have to do the same thing for the iPad version, but in this case, you wait until the popover is dismissed. Add this method to your file to accomplish that:

```
-(void)popoverControllerDidDismissPopover:(UIPopoverController *)popoverController {
    [_session startRunning];
}
```

Build and run now. Record a video. Hit the button. You should see all the videos on your device, including the one you just shot.



Hey, you did it! You built an awesome live video effects app! Enjoy it.

And for now, th-th-th-that's all folks! ☺

Where to go from here?

We've covered most of the capabilities of Core Image, and we've skimmed the surface of AVFoundation. You should be feeling pretty good about yourself right now.

Now that you've built a video recording application with Core Image, you can do just about anything!

However, there are important aspects of Core Image that we didn't cover. Some of the new capabilities of Core Image allow you to interface directly with an OpenGL environment. This means you could use Core Image to spice up games or other graphically-intensive applications.

Here is Apple's comprehensive resource explaining what all the Core Image filters do:

https://developer.apple.com/library/mac/#documentation/graphicsimaging/Conceptual/CoreImaging/ci_intro/ci_intro.html

Chapter 18: What's New with MapKit

By Matt Galloway

MapKit has been around since iOS 3.0 (or iPhone OS 3.0, as it was called back in 2009). iOS 4.0 introduced overlays – a powerful feature that enabled the drawing of shapes and routes on top of a map – but apart from that, MapKit hasn't changed all that much since iOS 3.0. That is, until iOS 6.0, which introduces two incredible new API features, `MKMapItem` and `MKDrectionsRequest`.

Another huge change with iOS 6.0 is that Apple will now use its own mapping service rather than licensing Google Maps. This brings a few new features to the built-in Maps app: vector maps that are crisp at any zoom level (rather than the old image-based ones, which can be blurry in-between zoom levels); a 3D "Flyover" mode, which is a very cool way of viewing a city; and turn-by-turn navigation, which turns your phone into a GPS navigation device. Woot!

The bad news is that 3D and turn-by-turn are not available for use in apps. The good news is that vector maps are. The *great* news is that you don't have to change a single line of code to get the new vector maps in your existing apps!

The two new API features are great because they are going to enable developers to create a whole new range of applications. The first feature, `MKMapItem`, provides the ability to open Maps very easily from your own app. Before iOS 6.0, if you wanted to do this, you had to create a specially-formed Google Maps URL and ask the application to open it. This was error-prone, and didn't allow for choosing the user's current location as one of the points to show on the map. Now we have the ability to open Maps with a clean, Objective-C API that does allow us to show the user's location.

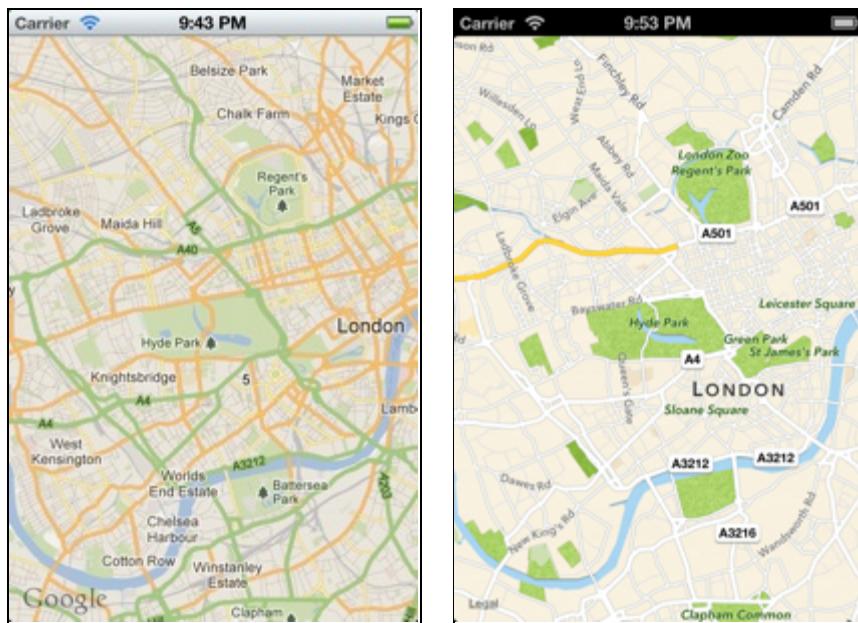
The second new feature, `MKDrectionsRequest`, lets you register your app as a provider of directions. How cool is that? So, if a user is in the Maps app and wants to know how to get from point A to B, they can either use the standard built-in routing tools, or they can ask for other options and get a list of apps that are known to be able to perform routing. The best bit is that Maps will even show apps that aren't yet on the user's phone, and allow purchasing and downloading from within the app. The possibilities are endless!

For example, consider an app that takes your directions request and sends a taxi to meet you at your start location, automatically charging your credit card for the trip. Or perhaps an app that takes your directions request and shows you possible walking tours that will not only take you from point A to B, but will also show you interesting sites along the way.

In this chapter, you'll create an app to help users navigate London using the Underground, the city's subway system. The app will show off the new vector maps and recap some of the things you can do with an `MKMapView`. You will then learn how to open Maps directly from your app. Finally, you'll register the app for providing directions, and learn how to service routing requests from within the app.

A look at the new maps

Before we get started, you must be itching to see what the new maps look like. Well, if you've got a device with iOS 6.0 already on it, then you can open the Maps app and take a look for yourself. If not, here's a comparison of the new maps to the old maps.

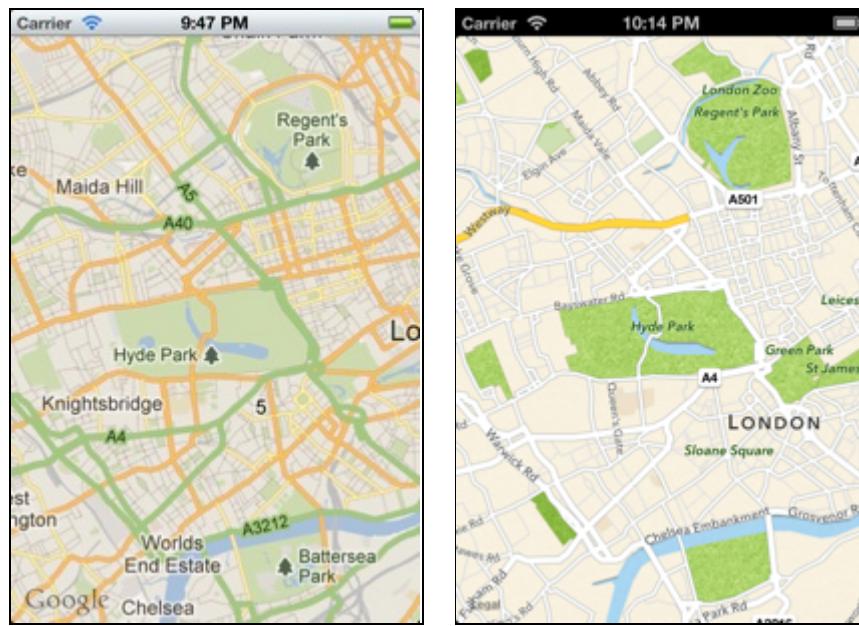


The screenshot on the left shows an old map and the screenshot on the right shows a new vector-based map. Notice how the new map is much clearer.

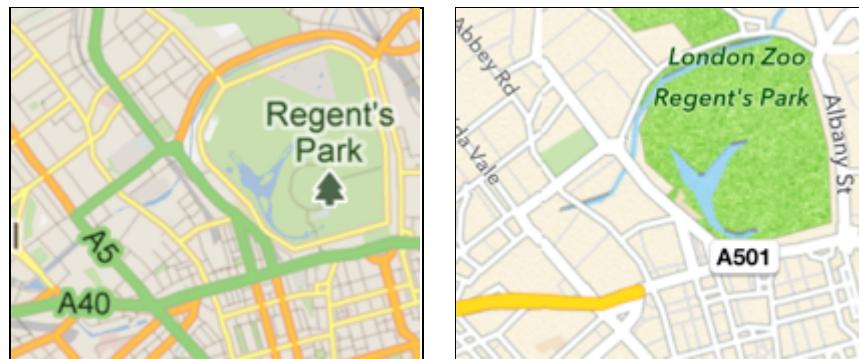
One problem with the old maps was that, because they were based on tiled sets of images at different zoom levels, when you zoomed in you wouldn't necessarily have a crisp image to view.

You can see this problem in the screenshots below. The old map is still showing the same tiles as in the previous set of images, but the new map is still able to show a crisp image because each of the roads, rivers, labels, etc. are individual UI

components that can be rendered at any zoom level. It might be difficult to see from the screenshots exactly what I mean so I encourage you to create a little app that shows just a map view and fire this up on both an iOS 5 device and an iOS 6 device. You should be able to see what I mean as you zoom in and out of the map. In particular look at road and place names which will be crisp at any zoom level in the new maps.

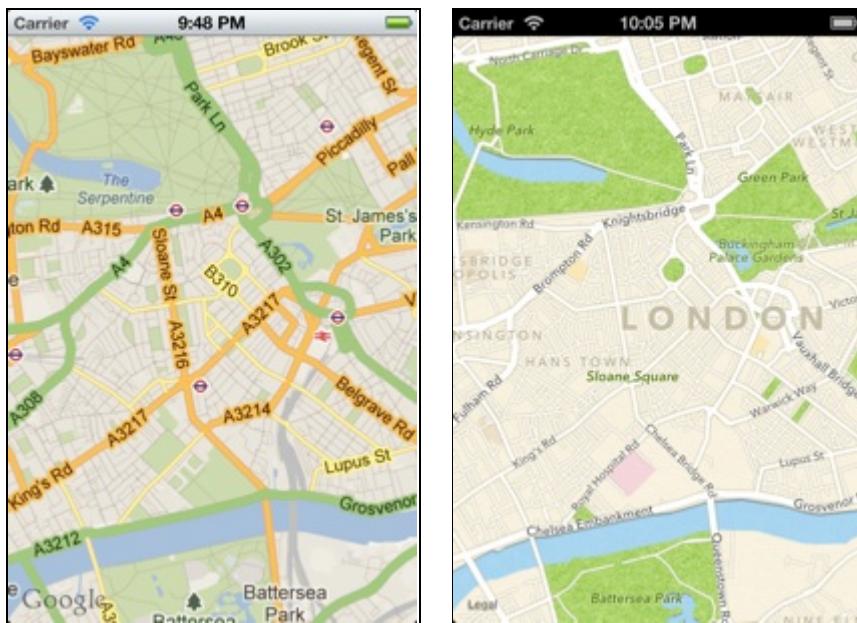


And here are a zoomed in section of those:

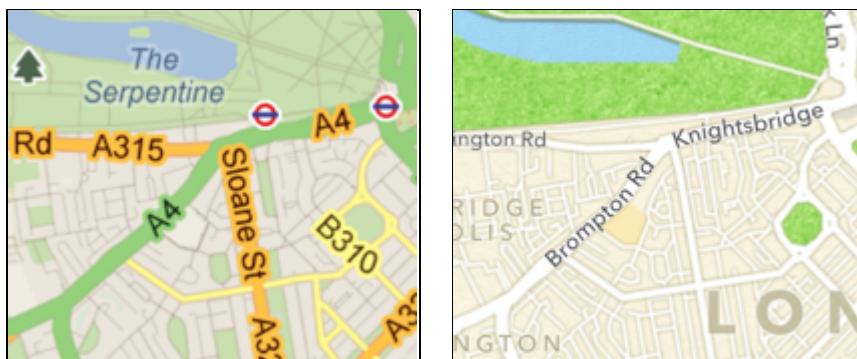


Also, when zoomed in, the new maps can rearrange UI components or display new ones to make the map as clear and concise as possible. If you compare the road name labels between the first and second images of the maps, you'll see that in the zoomed in version of the new map, some labels have been moved around, with new ones added. With the old-style map, however, it looks as though we've just increased the scale of the same image, and nothing more. Well, I guess that's because we have!

The two screenshots below show an even more zoomed in view. Notice how the old maps are blurry, because again, we're at a zoom level in between tile set changes, whereas the new maps are once again extremely clear.



And here is a zoomed in section of each of those:

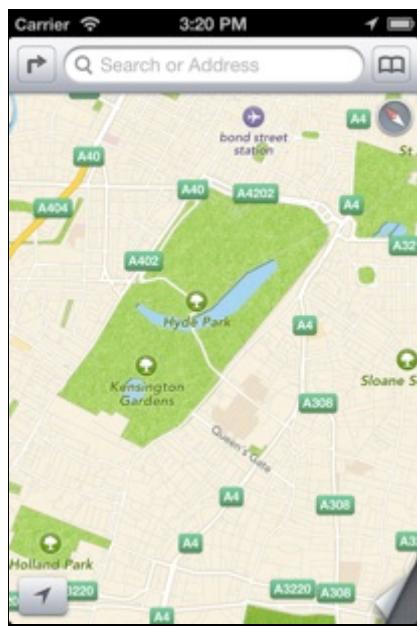


This comparison does, however, highlight one flaw in the new maps as I see it: roads are less prominent. The old maps are very good at showing the different kinds of roads – i.e. major, minor and even the footpaths through the park – and the new maps just aren't as clear in that respect. But if you're looking to get an overview of an area and its landmarks, the new maps keep you from feeling lost in a maze of roads!

You might also notice that in the maps screenshot on the right, the underground stops aren't visible. However, if you were to zoom in a little on the map, after you zoom in enough they would appear. The new maps is smart enough to make certain elements appear only at certain zoom level, in order to make the map uncluttered as you're zoomed out.

This illustrates something that you should consider when making map-related apps: you should consider the level of detail on the standard map at various zoom levels. Think about what users of your app will need to be able to see to get the job done, and play around with the maps at various zoom levels. Then choose your default zoom level to be consistent with a level of detail that provides the best experience, as you don't want your users having to frequently zoom in or out.

Another neat feature of the new iOS 6.0 maps is shown in the screenshot below, taken from the Maps app:



Notice how the map is rotated about 45 degrees, but the labels are still the right way up. This is an awesome feature of the new maps, making them easy to read at any rotation! Try rotating a map (using the compass feature) on an old version of iOS, and you'll see that when you rotate the map it can become incredibly hard to read because the words rotate with the map.

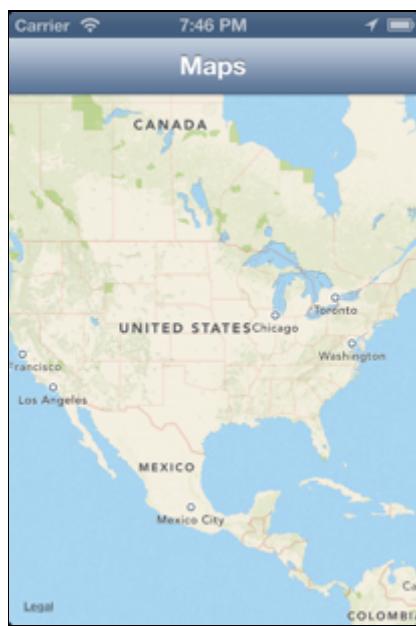
It should be obvious by now that the new maps are the way to go, and if you already have a map-based app then you'll be itching to see how the maps perform in it. Remember, no recompiling and resubmission is needed – the new maps just work! Read on to learn about how to use the new APIs available to apps.

Getting started

So you're ready to put these stunning new maps to work for yourself, right? Well, let's jump right into some actual code. Since the app you'll make is quite complex, I've created a starter project for you so that you can focus on the mapping parts, rather than the boilerplate code. You will transform the app into a tool for navigating around London, England via one of the underground train lines. Along

the way, you'll work with various features of MapKit, including all of the juicy new bits from iOS 6.0.

You can find the starter project in the resources for this chapter – it's called RWMapping Starter. Open it up, and spend some time getting familiar with how it works. Notice that it's using a storyboard that presents a navigation controller with just one view controller in it. Build and run the project, and you'll see something like this:



Go ahead and play around with the map – it should feel very similar to the Maps app in iOS, except that the cool new features are missing. So, during the rest of this chapter you'll take the following steps to expand the app's capabilities:

- Load a JSON file containing the coordinates and names of the London underground line's stations, and store the information in a custom data model.
- Show the stations and train line on the map. This will illustrate how to use the `MKAnnotation` protocol to present pins on a map, and `MKPolyline` to draw a line as an overlay to the map.
- Perform routing from point A to B, and enabling your app to be opened directly from the Maps app if the user is located within London and asks Maps for directions.
- Open the built-in Maps app to show the stations or ask for directions. Even though your app will itself perform routing, this will show you how to use `MKMapItem` and the various options available to control how Maps shows the data you give it.

But first, here's a little addition you should make to the project so that your users have the best experience. There's a key in the `Info.plist` file that specifies what capabilities the device should have to be able to run the app. This means that if the device isn't up to scratch, it won't run the app or even show it in the App Store as

an available download. Since this app depends on location services, it's wise to add that as a required capability.

Open the target settings by clicking the project root in the project navigator on the left hand side of the Xcode window. Then click on the RWMapping target in the list in the middle and select the Info tab. Open the "Custom iOS Target Properties" section (it's open by default, but if you have a different layout, you might need to open it manually), find the key called "Required device capabilities" and open it. Select Item 0 and click the round (+) button that appears. Then under the Value column, enter "location-services."

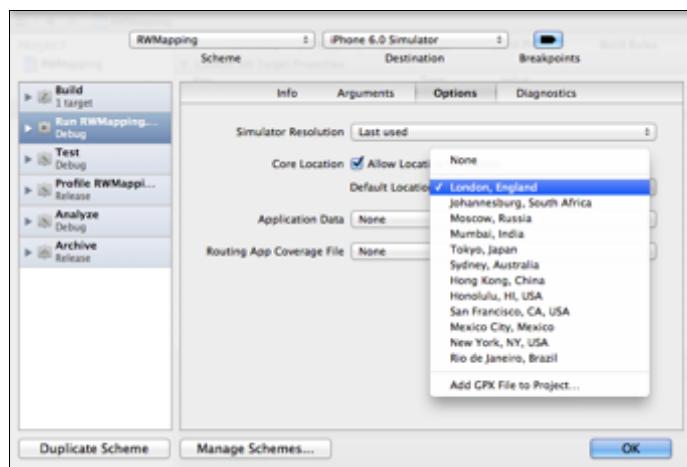
When you're finished, the section should look like this:

| Custom iOS Target Properties | | |
|--------------------------------|--------|-------------------------------------|
| Key | Type | Value |
| Required device capabilities | Array | (2 items) |
| Item 0 | String | armv7 |
| Item 1 | String | location-services |
| Bundle identifier | String | com.example.\$ PRODUCT_NAME:rfc1034 |
| InfoDictionary version | String | 6.0 |
| Main storyboard file base name | String | MainStoryboard |

Simulating your location

If you don't have a device with GPS, or if you're developing an app specifically targeted for a location other than your current one, it becomes tricky to test the app. I'm betting most of you are not presently in London! However, there's a handy feature in Xcode where you can ask both the simulator and your device to pretend to be somewhere they're not.

To do this, you must first enable the functionality in the scheme for your app. Select Product>Edit Scheme... from Xcode's menu. Select "Run RWMapping" on the left and then select the Options tab on the right. Make sure "Allow Location Simulation" is checked, and select London, England as the default location, since that's what's needed by the app you're making. This means when you run the app later on, you'll be teleported to England. OK, not really, but iOS will think you have been!

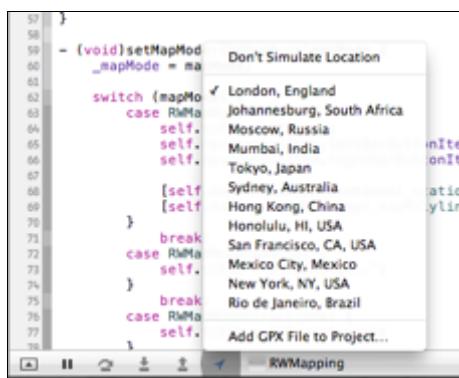


Note: If you build and run now, you'll notice it doesn't zoom into London quite yet. That's because the app currently doesn't have any code to zoom into the current location. You'll add that in a bit!

You have set a default location, but you can also change the location while the app is running. This can be useful if you want to simulate the user changing locations over time, and is done using a GPX file that contains coordinates and timestamps to say where the device should "be" at a given point in time. You can find out more about GPX files here:

http://en.wikipedia.org/wiki/GPS_eXchange_Format

To change the location while the app is running, use the button at the bottom of the editor window in Xcode:



Reading the data

Take a look at the starter project and you will see a file called `victoria_line.json`. If you have a look at the contents of the file, you'll see it's a simple JSON file containing an array of objects. These objects describe the stations on the Victoria train line (part of the London underground network). There is a name, latitude and longitude for each station. You'll now create a class to hold this information.

From the Xcode menu select `File\New\File...` then select Objective-C class and click Next. Call the class `RWStation`, make it a subclass of `NSObject` and click Next. Finally, save it along with the rest of the project.

Now open **RWStation.h** and replace it with:

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

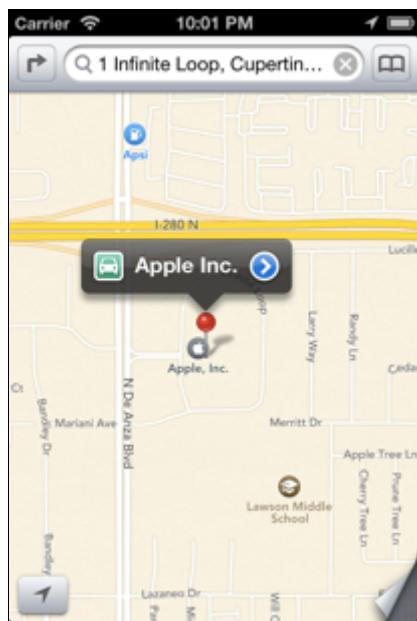
@interface RWStation : NSObject <MKAnnotation>
```

```
@property (nonatomic, copy) NSString *title;
@property (nonatomic, assign) CLLocationCoordinate2D coordinate;

- (MKMapItem*)mapItem;

@end
```

This is a good time to explain the `MKAnnotation` protocol. When a class implements this protocol, it can be added to an `MKMapView` to display a pin at a certain coordinate location and to present the familiar pop-up label, like this:



To be implemented, the `MKAnnotation` protocol requires an `NSString*` property named **title** and a `CLLocationCoordinate2D` property named **coordinate**. Once your class is declared as implementing this protocol, you can add instances of the class to a map, where they will handle all of the work of displaying the annotations (pins) at the correct locations, along with their pop-ups when the annotations are tapped.

Back to `RWStation` – apart from the properties for name and location, there is a method declared that returns an instance of `MKMapItem`, one of the new classes in iOS 6.0.

All about `MKMapItem`

So, what is this thing? Well, you use the `MKMapItem` class to define a point of interest on a map. Its primary use is for apps to pass information to the Maps app. An instance of `MKMapItem` can be created in one of two ways: either by providing an `MKPlacemark` instance, or by using the class method `mapItemForCurrentLocation`, which returns a special, singleton map item to be used when the program needs to refer to the user's current location.

Note: A singleton is an object that is instantiated only once and then shared. It's used in this context as a way of communicating the notion of the user's current location that will change over time. Note that you can easily access this special `MKMapItem` from anywhere with this simple snippet of code:

```
MKMapItem * curLoc = [MKMapItem mapItemForCurrentLocation];
```

This will be useful later when asking for routing directions to or from the current location.

There are a few different pieces of information that you can set on `MKMapItem` (again, so that this information will show up when you pass it to the Maps app):

- **NSString *name:** the name of the location. E.g., "Ray's Cake Shop".
- **NSString *phoneNumber:** a phone number for the location, if there is one. E.g., "+1-800-123-4567".
- **NSURL *url:** a URL for the location, if there is one. E.g., "http://www.raywenderlich.com/".

Note from Ray: I do not recommend you eat at this establishment, I am not known to be a great cook! ☺

In addition, there are two other properties:

- **MKPlacemark *placemark:** an `MKMapItem` is created using an `MKPlacemark` instance and this read-only property is used to retrieve it after creation. The placemark contains more information about the item such as its address or country code.
- **BOOL isCurrentLocation:** set to YES if this is a map item representing the user's current location (i.e., created using `mapItemForCurrentLocation`) and NO otherwise.

There are also a couple of methods that can be used to open the Maps app with one or more `MKMapItems`. The first is an instance method that opens the Maps app with a series of options:

```
- (BOOL)openInMapsWithLaunchOptions:(NSDictionary*)options
```

The second is a class method that opens the Maps app with a list of many locations, again with a series of options:

```
+ (BOOL)openMapsWithItems:(NSArray *)items  
                      launchOptions:(NSDictionary *)options
```

The options dictionary allows you to control what the Maps app displays when it opens. You create a dictionary using pre-defined keys supported by MapKit:

- **MKLaunchOptionsDirectionsModeKey**, if set, tells Maps to treat the items you send as the start and end positions for a routing request. You can tell it to choose walking (`MKLaunchOptionsDirectionsModeWalking`) or driving (`MKLaunchOptionsDirectionsModeDriving`) as the routing type. When this key is set, the number of items should be either two, for start and end places in that order, or a single item signifying the end location with the start place assumed to be the current location.
- **MKLaunchOptionsMapTypeKey** tells Maps what map type should be displayed. The value of this key should be an `NSNumber` with a value of `MKMapTypeStandard` for a standard map, `MKMapTypeSatellite` for satellite imagery, or `MKMapTypeHybrid` for a mix of the two.
- **MKLaunchOptionsMapCenterKey** tells Maps the point to center the map on. Use `+[NSValue valueWithMKCoordinate:]` to create an `NSValue*` to set as the value for this key.
- **MKLaunchOptionsMapSpanKey** tells Maps what region to display – in other words the “bounding coordinates” for the map view. Use `+[NSValue valueWithMKCoordinateSpan:]` to create an `NSValue*` to set as the value for this key.
- **MKLaunchOptionsShowsTrafficKey** tells Maps whether or not to show traffic on the Map. Set an `NSNumber` with a Boolean value for this key.

Map overlays and polylines

Just like annotations are used to display single points on a map, overlays are used to draw extra details on top of a map. For instance, you can draw a line to highlight a road, or in the case of the app you’re making in this chapter, a line to highlight the route of a train.

An overlay is a class that implements the `MKOOverlay` protocol, and the view that is drawn on top of the map is a subclass of `MKOOverlayView`. Creating a custom overlay is a lot harder than a custom annotation, but fortunately there are a few different overlays and associated overlay views that come with MapKit to make things easier:

- **MKPolyline** defines a single line connecting a series of coordinates. **MKPolylineView** is the associated overlay view.
- **MKCircle** defines a region enclosed within a circle. **MKCircleView** is the associated overlay view.
- **MKPolygon** defines a generic region enclosed within a series of coordinates. **MKPolygonView** is the associated overlay view.

You’ll be making use of the polyline later on, which is instantiated with a C-style array of coordinates using the `polylineWithCoordinates:count:` method. It’s not

entirely clear why Apple decided to use a C-style array rather than an `NSArray`. It makes it a lot harder to use and is certainly not in keeping with the rest of the iOS SDK, but I'm sure there are good reasons.

Note: A C-style array is a block of memory of size equal to the size of each element multiplied by the number of elements – basically the amount of memory needed to store all the elements. In the case of the array of coordinates just discussed, each element is of type `CLLocationCoordinate2D`. To create such an array, `malloc` is used to allocate the required amount of memory. This memory must be freed when done by calling `free`.

Back to the data

Now that you're armed with knowledge of `MKMapItem`, you can write the remaining method on `RWStation`. Open `RWStation.m` and replace it with:

```
#import "RWStation.h"

#import <AddressBook/AddressBook.h>

@implementation RWStation

- (MKMapItem*)mapItem {
    // 1
    NSDictionary *addressDict = @{
        (NSString*)kABPersonAddressCountryKey : @"UK",
        (NSString*)kABPersonAddressCityKey : @"London",
        (NSString*)kABPersonAddressStreetKey : @"10 Downing Street",
        (NSString*)kABPersonAddressZIPKey : @"SW1A 2AA"};

    // 2
    MKPlacemark *placemark = [[MKPlacemark alloc]
        initWithCoordinate:self.coordinate
        addressDictionary:addressDict];

    // 3
    MKMapItem *mapItem =
        [[MKMapItem alloc] initWithPlacemark:placemark];
    mapItem.name = self.title;
    mapItem.phoneNumber = @"+44-20-8123-4567";
    mapItem.url =
        [NSURL URLWithString:@"http://www.raywenderlich.com/"];

    return mapItem;
}
```

```
@end
```

Here's what the above method does:

1. It creates a dictionary to describe the location using the keys defined in the **AddressBook Framework**. There are a number of allowed keys and they can be found in `ABPerson.h`, which is part of the framework. In this case, you're faking it by giving each station the address of the prime minister of the UK. In reality you'd have some real data in there!
2. It creates an `MKPlacemark` object using the dictionary defined previously and the coordinates of this station. This object tells Maps where to display the pin for the station.
3. It creates an `MKMapItem` object using the placemark from before and extra information such as the name, phone number and URL for the station. Finally, the method returns the new `MKMapItem` instance.

You'll use this method later on, but for now just notice how simple it is to create map items.

Loading the list of stations

Now that you have your model object ready to store the station data, it's time to read in from the JSON file included in the starter project, `victoria_line.json`. Go ahead and open `RWViewController.m`, and add an import at the top for the station model:

```
#import "RWStation.h"
```

You need a way to store the stations. You also need a reference to the polyline added to the map to show where the train line runs. Add a couple of variables to the class extension category at the top of the file to handle this. It should look like this:

```
@interface RWViewController () <MKMapViewDelegate> {
    NSMutableArray *_stations;
    MKPolyline *_mapPolyline;
}
```

Note: The class extension category is a special category that, if used, must be included in the same file (or translation unit to be precise) as the main implementation of the class. It is the only category where extra instance variables can be added, and can be thought of as an extension to the `@interface` declaration in the class's header file. For more information, check out Chapter 2, "What's New with Objective-C".

Now add this method above viewDidLoad:

```
- (void)loadData {
    // 1
    NSData *data =
        [NSData dataWithContentsOfFile:
            [[NSBundle mainBundle] pathForResource:@"victoria_line"
                ofType:@"json"]];
    NSArray *stationData =
        [NSJSONSerialization JSONObjectWithData:data
            options:0
            error:nil];
    NSUInteger stationCount = stationData.count;

    // 2
    NSInteger i = 0;
    CLLocationCoordinate2D *polylineCoords =
        malloc(sizeof(CLLocationCoordinate2D) * stationCount);
    _stations =
        [[NSMutableArray alloc] initWithCapacity:stationCount];

    // 3
    for (NSDictionary *stationDictionary in stationData) {
        // 4
        CLLocationDegrees latitude =
            [[stationDictionary objectForKey:@"latitude"] doubleValue];
        CLLocationDegrees longitude =
            [[stationDictionary objectForKey:@"longitude"] doubleValue];
        CLLocationCoordinate2D coordinate =
            CLLocationCoordinate2DMake(latitude, longitude);
        polylineCoords[i] = coordinate;

        // 5
        RWStation *station = [[RWStation alloc] init];
        station.title =
            [stationDictionary objectForKey:@"name"];
        station.coordinate = coordinate;
        [_stations addObject:station];

        i++;
    }

    // 6
    _mapPolyline =
        [MKPolyline polylineWithCoordinates:polylineCoords
            count:stationCount];
```

```
// 7  
free(polylineCoords);  
}
```

This method loads in the data like so:

1. First it reads the data into an array using `NSJSONSerialization`. This method usually returns either a dictionary or an array. In this case, since you know the structure of the data, you know it's an array. Also, since you'll need the number of stations a few times in your code, keep that around in a variable.
2. To create a polyline, you need to pass in a C-style array of coordinates, so the method uses `malloc` to create an array of the right size. Also, it initializes the internal stations array to an empty mutable array of the correct size.
3. It loops over each station's JSON data, represented by an `NSDictionary`.
4. It reads the latitude and longitude and creates a new pair of coordinates. Then it adds the pair of coordinates to the polyline array.
5. It creates a new station object with the pair of coordinates that has just been created and the name. Then add this station to the array of stations.
6. After loading all stations, it creates a polyline using all the coordinates. It also needs to know the count, since it's a C-style array and there's no way to implicitly know how big the array is.
7. Finally, it cleans up after itself. ARC will handle most of the memory management, but one thing it can't do is to free any memory allocated using `malloc`.

Annotations and overlays

Now that you've written the code to load the stations, you just need to display them! Since this app will eventually have three display modes – map, loading route, and displaying route – a lot of switching of the display is done in `setMapMode`:

In this particular case, you just need to add the stations as map annotations (recall the `MKAnnotation` protocol) and add the polyline as a map overlay. So update `setMapMode`: for the `RWMapModeNormal` case to look like this:

```
case RWMapModeNormal: {  
    self.title = @"Maps";  
    [self.mapView addAnnotations:_stations];  
    [self.mapView addOverlay:_mapPolyline];  
}
```

A requirement for adding annotations and overlays to a map is telling MapKit how to display them. This is as simple as overriding a couple of `MKMapViewDelegate` methods. Add the following method to the end of the file:

```

- (MKAnnotationView*)mapView:(MKMapView *)mapView
    viewForAnnotation:(id<MKAnnotation>)annotation
{
    if ([annotation isKindOfClass:[RWStation class]]) {
        static NSString *const kPinIdentifier = @"RWStation";
        MKPinAnnotationView *view = (MKPinAnnotationView*)[mapView
dequeueReusableAnnotationViewWithIdentifier:kPinIdentifier];
        if (!view) {
            view = [[MKPinAnnotationView alloc]
                initWithAnnotation:annotation
                reuseIdentifier:kPinIdentifier];
            view.canShowCallout = YES;
            view.calloutOffset = CGPointMake(-5, 5);
            view.animatesDrop = NO;
        }

        view.pinColor = MKPinAnnotationColorRed;

        return view;
    }
    return nil;
}

```

This method creates a new view that the map will automatically display at the correct location for an annotation. MapKit comes with a class called `MKPinAnnotationView` that displays the usual pin graphic. In this method, an `MKPinAnnotationView` is created and set up as desired.

Next, add this method to the end of the file:

```

- (MKOverlayView*)mapView:(MKMapView *)mapView
    viewForOverlay:(id<MKOverlay>)overlay {
    MKPolylineView *overlayView =
    [[MKPolylineView alloc] initWithPolyline:overlay];
    overlayView.lineWidth = 10.0f;
    overlayView.strokeColor = [UIColor blueColor];
    return overlayView;
}

```

This method creates a new view that is overlaid on top of the map for the polyline you created earlier. Again, MapKit comes with a handy class to do the job for you – this time it's called `MKPolylineView`. It's created with a polyline, and then the color and width of the line are set.

Let's display those stations!

Phew! That was quite a bit of code and you're almost at the point of displaying stations. But not quite! You need to call `loadData` somewhere. So change `viewDidLoad` to look like this:

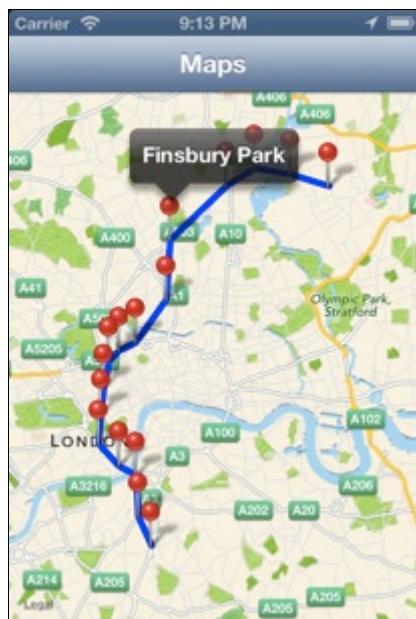
```
- (void)viewDidLoad {
    [super viewDidLoad];

    [self loadData];
    self.mapMode = RWMapModeNormal;

    CLLocationCoordinate2D center =
    CLLocationCoordinate2DMake(51.525635, -0.081985);
    MKCoordinateSpan span =
    MKCoordinateSpanMake(0.12649, 0.12405);
    MKCoordinateRegion regionToDisplay =
    MKCoordinateRegionMake(center, span);
    [self.mapView setRegion:regionToDisplay animated:NO];
}
```

This will load the data, set the map mode to normal (which you'll recall adds the stations as annotations and the polyline as an overlay), and then sets the map region so that it displays all the stations completely.

Build and run the app and you'll see the train line and stations! Woop!



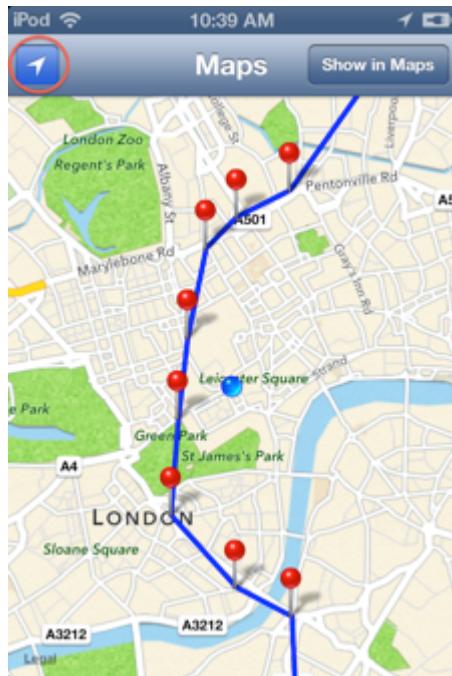
A button for this, a button for that

Now that you've loaded the data and displayed it on the map, it's time to start playing around with it. The first thing to do is add buttons to track the user and to open the Maps app to show the stations.

Open **RWViewController.m** again and modify the `RWMapModeNormal` case in `setMapMode:` to look like this:

```
case RWMapModeNormal: {
    self.title = @"Maps";
    self.navigationItem.leftBarButtonItem =
    [[MKUserTrackingBarButtonItem alloc]
        initWithTitle:@"Show in Maps"
        style: UIBarButtonItemStyleBordered
        target:self
        action:@selector(showInMaps:)];
    [self.mapView addAnnotations:_stations];
    [self.mapView addOverlay:_mapPolyline];
}
```

Here you add one button to perform the `showInMaps:` action that you'll implement momentarily, and another button that is a special `MKUserTrackingBarButtonItem`. This special button allows you to toggle a map between not tracking the user, and tracking the user with and without heading (rotating the map to the direction the user is facing). How cool is that?



With just one line of code, you've implemented a very useful button, just like in the Maps app, that allows your users to make the map home in on their present location and track them as they move around.

Now add the following method to the file:

```
- (void)showInMaps:(id)sender {
    // 1
    NSMutableArray *mapItems = [[NSMutableArray alloc] init];
    for (RWStation *station in _stations) {
        [mapItems addObject:[station mapItem]];
    }
    [mapItems addObject:[MKMapItem mapItemForCurrentLocation]];

    // 2
    MKMapRect boundingBox = [_mapPolyline boundingMapRect];
    MKCoordinateRegion boundingBoxRegion =
    MKCoordinateRegionForMapRect(boundingBox);

    // 3
    NSValue *centerAsValue =
    [NSValue valueWithMKCoordinate:boundingBoxRegion.center];
    NSValue *spanAsValue =
    [NSValue valueWithMKCoordinateSpan:boundingBoxRegion.span];

    // 4
    NSDictionary *launchOptions =
```

```
    @{@"MKLaunchOptionsMapTypeKey" : @(MKMapTypeHybrid),  
     MKLaunchOptionsMapCenterKey : centerAsValue,  
     MKLaunchOptionsMapSpanKey : spanAsValue};  
  
    // 5  
    [MKMapItem openMapsWithItems:mapItems  
                  launchOptions:launchOptions];  
}
```

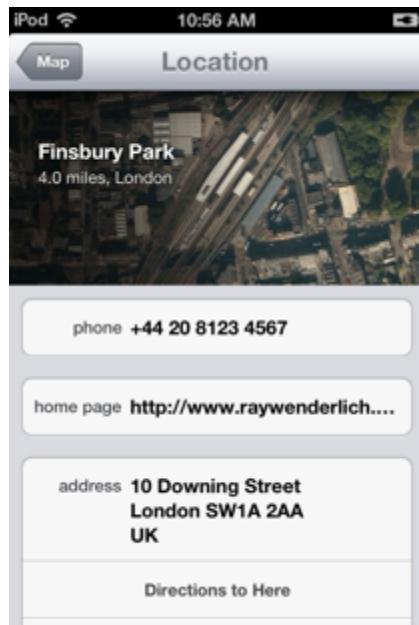
This is the method you set as the target of the right bar button you created above. Here's what it does:

1. It creates an array of `MKMapItem` objects, one for each station. Recall the method you created earlier to get one of these objects from a station.
2. It finds out the bounding box of the train line in terms of map points and then converts that to a coordinate region. Map points are an internal MapKit representation of the globe, whereas coordinate regions are latitude and longitude values. It's the latter you need to pass to the Maps app.
3. It wraps the center and span portions of the coordinate region into `NSValue` objects.
4. It creates a launch options dictionary using the various keys described earlier and the relevant values. It sets the map type to hybrid, and the center and span properties to the values created previously.
5. It opens Maps with the items and the options.

And that really is it! Build and run the app, and tap the right button. It will launch the Maps app in hybrid mode and display your underground locations, as shown below:



Try tapping on the blue “right” arrow next to one of the stations. You’ll see a detail screen appear with all of the information that you populated from your app.



Pretty cool, eh? This might be useful if you were making an app for a store – you could have a button in your app to show the closest store location in the Maps app along with this useful information like the phone number, home page, and address.

Dropping the pin

You’ve almost definitely used the feature in the Maps app where you can drop a pin on the map and move it around. You’ve probably thought that this would be hard to add to your own app. Well, it’s really not, and I’m going to show you how to do that now using just a few lines of code.

Open **RWViewController.m** and add an instance variable called `_droppedPin` to the class extension category – the final result should look like this:

```
@interface RWViewController () <MKMapViewDelegate> {
    NSMutableArray *_stations;
    MKPolyline *_mapPolyline;
    RWStation *_droppedPin;
}
```

You’ll make use of the `RWStation` class to store the location of the current dropped pin. You reuse this class for the sake of simplicity, since the dropped pin will be added to the map as an annotation and the station class already does everything you need.

Now add the following code at the end of viewDidLoad:

```
UILongPressGestureRecognizer *longPressRecognizer =
    [[UILongPressGestureRecognizer alloc]
        initWithTarget:self
            action:@selector(handleLongPress:)];
longPressRecognizer.minimumPressDuration = 1.0;
[self.mapView addGestureRecognizer:longPressRecognizer];
```

This adds a gesture recognizer to the map view to detect when there's a screen touch that lasts for 1 second or longer. Now, implement the handleLongPress: method by adding the following code:

```
- (void)handleLongPress:(UIGestureRecognizer*)recognizer {
    // 1
    if (recognizer.state == UIGestureRecognizerStateBegan) {
        // 2
        if (_droppedPin) {
            [self.mapView removeAnnotation:_droppedPin];
            _droppedPin = nil;
        }

        // 3
        CGPoint touchPoint =
[recognizer locationInView:self.mapView];
        CLLocationCoordinate2D touchMapCoordinate =
[self.mapView convertPoint:touchPoint
                      toCoordinateFromView:self.mapView];

        // 4
        _droppedPin = [[RWStation alloc] init];
        _droppedPin.coordinate = touchMapCoordinate;
        _droppedPin.title = @"Dropped Pin";

        // 5
        [self.mapView addAnnotation:_droppedPin];
    }
}
```

This is what that method does:

1. Gesture recognizers go through many states. You only want to drop a pin when the long press begins, which is just after the 1 second has elapsed with a single touch on the map.
2. If there's already a pin on the map, then the method removes it and clears the instance variable, as it's about to create a new one.

3. It finds the point within the map view and the matching coordinates (i.e. latitude and longitude) where the long press occurred.
4. It creates a new `RWstation` object to represent this point with a relevant title.
5. Finally, it adds it to the map as an annotation.

There's now just one final thing that needs to be done. Remember implementing the map view delegate method for creating a view for a given annotation? You probably want to display this dropped pin differently than the normal stations, so that it's easily distinguishable.

What's more, you get a very nice feature for free with the pin view that comes with MapKit – the ability to allow a pin to be dragged around! That's exactly what you need, and to use it requires just a small change to a delegate method.

Find the `mapView:viewForAnnotation:` method and replace it with this:

```
- (MKAnnotationView*)mapView:(MKMapView *)mapView
    viewForAnnotation:(id<MKAnnotation>)annotation
{
    if ([annotation isKindOfClass:[RWStation class]]) {
        static NSString *const kPinIdentifier = @"RWStation";
        MKPinAnnotationView *view = (MKPinAnnotationView*)[mapView
dequeueReusableAnnotationViewWithIdentifier:kPinIdentifier];
        if (!view) {
            view = [[MKPinAnnotationView alloc]
                initWithAnnotation:annotation
                reuseIdentifier:kPinIdentifier];
            view.canShowCallout = YES;
            view.calloutOffset = CGPointMake(-5, 5);
            view.animatesDrop = NO;
        }

        // 1
        if ((RWStation*)annotation == _droppedPin) {
            // 2
            view.pinColor = MKPinAnnotationColorPurple;
            view.draggable = YES;
        } else {
            // 3
            view.pinColor = MKPinAnnotationColorRed;
            view.draggable = NO;
        }

        return view;
    }
    return nil;
}
```

```
}
```

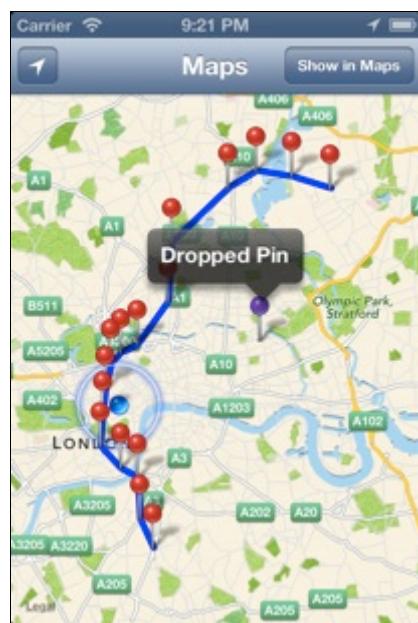
There are only a few lines that have changed:

1. The annotation is checked to see if it is the dropped pin or any other.
2. If it's the dropped pin, the pin's color is set to purple and it is set to be draggable.
3. If it's not the dropped pin (i.e., it's a normal station), then the color is set to red as before and the pin set to not draggable.

Note: The reason for setting the pin to not draggable in the case of normal stations, even though that's the default, is that map views reuse annotation views. So, if the de-queued pin view is one that previously was used for the dropped pin, then the draggable state has to be reset. This is similar to the pattern of reusable table view cells.

And that really is it to enable dragging of the pin! I bet you thought that was going to take much more code than simply "view.draggable = YES"!

Build and run the app and have a play around with dropping a pin and dragging it about.



From here to there

So far, you're able to drop pins for underground stations, draw lines showing the underground route, move pins around, and display all of the above in the official

Maps app. Now you're getting to the *really* fun stuff – allowing your app to be a route provider!

You'll write some code to enable your app to calculate the best underground route to take to get to a particular destination. And later on, you'll set things up so Maps will actually ask *your* app for directions. Here's what needs to be done:

- Create a model to hold a route plan.
- Implement a method to find the nearest station for a given pair of coordinates so that the start and end stations can be found.
- Calculate the route between two given stations.

Most of this is pretty easy, but there's a fair bit of code, so you might want to grab your favorite caffeinated beverage before continuing on. ☺ Now that you're well supplied, let's get started!

The route model

To store the route, you need a model class to store the start and end stations and a map polyline joining up these two stations, as well as all the stations in between them. Add a new file by selecting File\New\File... then select Objective-C class and click Next. Call the class **RWRoute**, make it a subclass of **NSObject** and click Next. Finally, save it along with the rest of the project.

Now open **RWRoute.h** and replace it's contents with this:

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

@class RWStation;

@interface RWRoute : NSObject

@property (nonatomic, strong) RWStation *fromStation;
@property (nonatomic, strong) RWStation *toStation;
@property (nonatomic, strong) MKPolyline *mapPolyline;

@end
```

That's all there is to it. Since there are no methods to implement, and thanks to the auto-synthesis of the new compiler, there's nothing to edit in the implementation file! You're all set to create some routes.

Finding the nearest station

When the user is looking for a route, remember that they might not be directly at a station – they might be somewhere nearby. So you'll be allowing users to create routes from any coordinate to any other coordinate, not just station to station.

So first you need a method that takes a set of coordinates and returns the nearest station for those coordinates. Open **RWViewController.m** and add the following method:

```
- (RWStation*)nearestStationToCoordinate:
    (CLLocationCoordinate2D)coordinate
{
    // 1
    __block RWStation *nearestStation = nil;
    __block CLLocationDistance nearestDistance = DBL_MAX;
    CLLocation *coordinateLocation =
        [[CLLocation alloc] initWithLatitude:coordinate.latitude
                                      longitude:coordinate.longitude];

    // 2
    [_stations enumerateObjectsUsingBlock:^(RWStation *station,
    NSUInteger idx, BOOL *stop) {
        // 3
        CLLocation *thisLocation = [[CLLocation alloc]
                                     initWithLatitude:station.coordinate.latitude
                                     longitude:station.coordinate.longitude];

        // 4
        CLLocationDistance thisDistance =
            [coordinateLocation distanceFromLocation:thisLocation];

        // 5
        if (thisDistance < nearestDistance) {
            nearestDistance = thisDistance;
            nearestStation = station;
        }
    }];

    // 6
    return nearestStation;
}
```

Here's what the above method does:

1. To work out which station is closest, you need to enumerate all stations. This is best done with a handy method on `NSArray` called `enumerateObjectsUsingBlock:`. This means you need some temporary variables to hold the current nearest station and the current nearest distance. They need to be declared `__block` so that they can be changed from within the enumeration block. Also, the method makes use of `CLLocation`'s handy method for determining distances between locations. A temporary location object is created, based on the passed-in coordinates, to match each candidate station against.

2. The method begins iterating through the stations using the block enumeration method.
3. It creates a temporary location object from the candidate station's coordinates.
4. It finds the distance from the target coordinates to this station. If the new distance is less than the current nearest distance, then it's found a better candidate, so it sets the values to indicate that.

Since you declared the variables as `__block`, you were able to initialize them outside the block, change them within the block, and refer to the changed value once the block completed. Normally – if you don't use the `__block` keyword – variables used in blocks are copied for use within the block, and any changes made to that variable inside the block are not visible outside the block.

5. Finally, the method returns the station that was found to be nearest to the target coordinates.

What about routing from the current location?

You want to allow your users to route from their current location as well as from point A to point B. To do this, it's easiest to use a feature of the map view that allows you to display the user's location automatically. You can then get access to this location through the map view's `userLocation` property.

However, since the GPS in the device is not 100% reliable, and doesn't instantaneously find a lock, when you need access to the location it might not be ready yet. No worries, I'm going to show you a quick and simple method for finding the user's current location asynchronously. You'll call a method passing a custom block that takes a coordinate parameter, to be called once the location has been found. The block might be called immediately or after some time.

First, create a new type – a custom block type – by adding the following code at the top of **RWViewController.m** right underneath all the `#imports`:

```
typedef void (^RWLocationCallback)(CLLocationCoordinate2D);
```

This means you can now define a variable of type `RWLocationCallback`, which is a block that takes a single parameter, a pair of coordinates.

Next you need to store the current callback block to be run when the location has been found. Again, there's no better place to do this than the class extension category. So add a variable there like this:

```
@interface RWViewController () <MKMapViewDelegate> {
    NSMutableArray *_stations;
    MKPolyline *_mapPolyline;
    RWStation *_droppedPin;
    RWLocationCallback _foundLocationCallback;
}
```

Then you need to implement the method that will be called to asynchronously get the user's current location. Add the following to the file:

```
- (void)performAfterFindingLocation:(RWLocationCallback)callback
{
    if (self.mapView.userLocation != nil) {
        if (callback) {
            callback(self.mapView.userLocation.coordinate);
        }
    } else {
        _foundLocationCallback = [callback copy];
    }
}
```

This method first checks if the map view already knows the current location. If it does, it executes the callback block immediately with the user's location. But if it's not yet known, then it copies the block and stores it for later when the location is known.

Note: If you are wondering why a copy of the block is made via `[callback copy]` rather than setting directly to the instance variable, it's because the block that is passed in might either be on the stack or the heap. If it's on the stack, then you need to copy it to put it on the heap, since when the block is eventually used that original stack frame may be long gone. If the block that was passed in is already on the heap, then copying the block will have no effect other than to increment its retain count by 1, so no need to worry about redundantly copying blocks in that case.

You may be wondering how on earth to determine when the GPS finally kicks into action and delivers the current location. Well, as I like to say, "There's a delegate method for that." ☺ Add the following method to the end of the file:

```
- (void)mapView:(MKMapView *)mapView
didUpdateUserLocation:(MKUserLocation *)userLocation
{
    if (_foundLocationCallback) {
        _foundLocationCallback(userLocation.coordinate);
    }
    _foundLocationCallback = nil;
}
```

This is called whenever the map view is informed of an update to the user's current location. All that's needed is to call the location callback, which was set earlier. Then the callback is set back to nil, such that it's not called over and over again every time the user moves around.

Calculating the route

Calculating the route is simple now that you can find the nearest station to the start and end points. It's just a matter of walking the stations array between the start and end stations, building up a list of the stations to pass through along the route. You need this list, as you want to show a line on the map plotted through the stations.

First, import the `RWRoute` header into `RWViewController.m`, as that will be used to store the currently displayed route:

```
#import "RWRoute.h"
```

Next, add a variable to the class extension category to store the current route:

```
@interface RWViewController () <MKMapViewDelegate> {
    NSMutableArray *_stations;
    MKPolyline *_mapPolyline;
    RWStation *_droppedPin;
    RWLocationCallback _foundLocationCallback;
    RWRoute *_currentRoute;
}
```

Now add the following method, which will be the method used to calculate the route and create the route object:

```
- (void)routeFrom:(CLLocationCoordinate2D)from
              to:(CLLocationCoordinate2D)to
{
    // 1
    RWStation *fromStation =
        [self nearestStationToCoordinate:from];
    RWStation *toStation =
        [self nearestStationToCoordinate:to];

    // 2
    NSUInteger fromStationIdx =
        [_stations indexOfObject:fromStation];
    NSUInteger toStationIdx =
        [_stations indexOfObject:toStation];

    // 3
    BOOL forwards = (toStationIdx > fromStationIdx);

    // 4
    NSUInteger stationCount = 0;
    if (forwards) {
```

```
        stationCount = toStationIdx - fromStationIdx + 1;
    } else {
        stationCount = fromStationIdx - toStationIdx + 1;
    }

// 5
CLLocationCoordinate2D *polylineCoords =
    malloc(sizeof(CLLocationCoordinate2D) * stationCount);
for (NSUInteger i = 0; i < stationCount; i++) {
    // 6
    NSUInteger stationIndex = 0;
    if (forwards) {
        stationIndex = fromStationIdx + i;
    } else {
        stationIndex = fromStationIdx - i;
    }

    // 7
    RWStation *thisStation = _stations[stationIndex];
    polylineCoords[i] = thisStation.coordinate;
}

// 8
RWRoute *newRoute = [[RWRoute alloc] init];
newRoute.fromStation = fromStation;
newRoute.toStation = toStation;
newRoute.mapPolyline =
    [MKPolyline polylineWithCoordinates:polylineCoords
                                count:stationCount];
_currentRoute = newRoute;

// 9
free(polylineCoords);

// 10
self.mapMode = RWMapModeDirections;
}
```

That may look daunting, but don't worry – we'll go over it section by section.

1. It finds out which stations are closest to the start and end points.
2. It finds the index into the internal stations array for the start and end stations.
3. It determines if the route is taken forward or backward with respect to the ordering of the stations in the array – i.e., the direction of movement along the train line.

4. It works out how many stations the train will go through along the route.
5. It loops over the number of stations and, as before, creates a C-style array of coordinates. This is needed to create the polyline that shows the portion of the train line used on the route.
6. The current station index in the loop is equal to the starting station index \pm the current loop index, depending on the direction of movement along the train line.
7. It builds the polyline using the next station's coordinates.
8. After the polyline has been completely built, the method creates a new route object using the gathered data, and sets it as the current route.
9. Like before, it cleans up the C-style array at the end.
10. Finally, it sets the map mode to `RWMapModeDirections`. Currently this doesn't result in any display changes, but you'll get to that shortly.

You also want to supply a method to route *from* the current location *to* a certain place, and to route *to* the current location *from* a certain place. This can be achieved using the method you've just written along with the previous asynchronous current location lookup method.

Add the following two methods to **RWViewController.m**:

```
- (void)routeFromCurrentLocationTo:(CLLocationCoordinate2D)to
{
    self.mapMode = RWMapModeLoading;
    [self performAfterFindingLocation:
     ^(CLLocationCoordinate2D coordinate) {
        [self routeFrom:coordinate to:to];
    }];
}

- (void)routeToCurrentLocationFrom:(CLLocationCoordinate2D)from
{
    self.mapMode = RWMapModeLoading;
    [self performAfterFindingLocation:
     ^(CLLocationCoordinate2D coordinate) {
        [self routeFrom:from to:coordinate];
    }];
}
```

These methods are straightforward. They find the current location and then use that, along with the passed-in coordinates, to call the routing method as appropriate. The map mode is set to `RWMapModeLoading` before finding the user's location – this will show a "Loading..." message while the location is being found.

Displaying the route on the map

The final piece of the puzzle left to implement is the rest of `setMapMode:`, which now has to handle both the `RWMapModeLoading` case and the `RWMapModeDirections` case. Change the whole method to look like this:

```
- (void)setMapMode:(RWMapMode)mapMode {
    _mapMode = mapMode;

    switch (mapMode) {
        case RWMapModeNormal: {
            self.title = @"Maps";
            self.navigationItem.leftBarButtonItem =
                [[MKUserTrackingBarButtonItem alloc]
                    initWithTitle:self.mapView];
            self.navigationItem.rightBarButtonItem =
                [[UIBarButtonItem alloc]
                    initWithTitle:@"Show in Maps"
                    style:UIBarButtonItemStyleBordered
                    target:self
                    action:@selector(showInMaps:)];
        }

        // 1
        if (_currentRoute) {
            [self.mapView removeAnnotation:_currentRoute.fromStation];
            [self.mapView removeAnnotation:_currentRoute.toStation];
            [self.mapView removeOverlay:_currentRoute.mapPolyline];
            _currentRoute = nil;
        }

        [self.mapView addAnnotations:_stations];
        [self.mapView addOverlay:_mapPolyline];

        // 2
        if (_droppedPin) {
            [self.mapView addAnnotation:_droppedPin];
        }
    }
    break;
}

case RWMapModeLoading: {
    // 3
    self.title = @"Loading...";
    self.navigationItem.leftBarButtonItem = nil;
    self.navigationItem.rightBarButtonItem = nil;

    // 4
}
```

```
        if (_currentRoute) {
            [self.mapView removeAnnotation:_currentRoute.fromStation];
            [self.mapView removeAnnotation:_currentRoute.toStation];
            [self.mapView removeOverlay:_currentRoute.mapPolyline];
            _currentRoute = nil;
        }

        [self.mapView removeAnnotations:_stations];
        [self.mapView removeOverlay:_mapPolyline];
        if (_droppedPin) {
            [self.mapView removeAnnotation:_droppedPin];
        }
    }
    break;
case RWMapModeDirections: {
    // 5
    self.title = @"Directions";
    self.navigationItem.leftBarButtonItem = nil;
    self.navigationItem.rightBarButtonItem = nil;

    // 6
    [self.mapView removeAnnotations:_stations];
    [self.mapView removeOverlay:_mapPolyline];
    if (_droppedPin) {
        [self.mapView removeAnnotation:_droppedPin];
    }

    // 7
    [self.mapView addAnnotation:_currentRoute.fromStation];
    [self.mapView addAnnotation:_currentRoute.toStation];
    [self.mapView addOverlay:_currentRoute.mapPolyline];
}
break;
}
}
```

Wow, that's a long method! But you'll notice quite a bit of duplication in there. This is because there's a lot of UI switching, depending on what mode the app's in. There are a few bits that have changed:

1. If there's a current route, then the annotations and overlays from that route need to be removed when going back to normal mode.
2. If there is a dropped pin, then it should be added back when returning to normal mode.
3. In the case of `RWMapModeLoading`, the title should be set appropriately and the navigation buttons removed.

4. When loading (i.e. the user's current location hasn't been found yet), no annotations or overlays should be shown (since there's nothing to show yet), so they are all removed.
5. As with loading, in the case of `RWMapModeDirections` the title is set appropriately and the navigation buttons removed.
6. When displaying a route, you don't want to show the entire list of station pins or any drop pins – you just want to show the pins related to the route the user is taking. So here you remove all pins.
7. The annotations and overlay are added from the current route.

There's just one more method to tweak – the one for displaying map overlays. Since you've added a new overlay for the route, you need to tell the map view how to display it. This can be done by editing the delegate method.

Find `mapView:viewForOverlay:` and change it to look like this:

```
- (MKOverlayView*)mapView:(MKMapView *)mapView
    viewForOverlay:(id<MKOverlay>)overlay
{
    MKPolylineView *overlayView =
        [[MKPolylineView alloc] initWithPolyline:overlay];
    overlayView.lineWidth = 10.0f;

    if (overlay == _mapPolyline) {
        overlayView.strokeColor = [UIColor blueColor];
    } else if (overlay == _currentRoute.mapPolyline) {
        overlayView.strokeColor = [UIColor greenColor];
    }

    return overlayView;
}
```

All you've done here is added another case for when the overlay to be displayed is the current route's polyline and set the stroke color for the overlay to be green rather than blue.

What about trying it out?

You're going to have to hold off for a bit longer before trying out the routing. (But you probably should compile your code at this point, if you haven't already, to make sure that everything you've added so far compiles without any issues.)

Well, before you can try it out you need to add some UI so the user can find a route! We'll do this by adding a button the user can tap on a station to find the route to that station.

All aboard!

Having just implemented routing, I guess you're dying to try it out. And while I've already covered how to open Maps with multiple items, you might want to open it with just one item. You will now add the ability to do both of these things.

This is also a good time to show another of the Maps launch options: the ability to ask for a route calculation.

Open **RWViewController.m** again and find `mapView:viewForAnnotation:`. Replace the current implementation with this:

```
- (MKAnnotationView*)mapView:(MKMapView *)mapView
    viewForAnnotation:(id<MKAnnotation>)annotation
{
    if ([annotation isKindOfClass:[RWStation class]]) {
        static NSString *const kPinIdentifier = @"RWStation";
        MKPinAnnotationView *view = (MKPinAnnotationView*)[mapView
dequeueReusableAnnotationViewWithIdentifier:kPinIdentifier];
        if (!view) {
            view = [[MKPinAnnotationView alloc]
                initWithAnnotation:annotation
                reuseIdentifier:kPinIdentifier];
            view.canShowCallout = YES;
            view.calloutOffset = CGPointMake(-5, 5);
            view.animatesDrop = NO;
        }

        if ((RWStation*)annotation == _droppedPin) {
            view.pinColor = MKPinAnnotationColorPurple;
            view.draggable = YES;
        } else {
            view.pinColor = MKPinAnnotationColorRed;
            view.draggable = NO;
        }

        // LINES ADDED HERE
        if (self.mapMode == RWMapModeNormal) {
            view.rightCalloutAccessoryView = [UIButton
buttonWithType:UIButtonTypeDetailDisclosure];
        } else {
            view.rightCalloutAccessoryView = nil;
        }

        return view;
    }
}
```

```
    return nil;
}
```

The code that has been added here tells the annotation views to show an accessory in the popup that's displayed when an annotation is tapped (called the "callout" view). The accessory you want for this app is a standard detail disclosure button. Since users won't want to be messing around with opening stations in Maps when in the loading or directions modes, the accessory view is set back to none in those cases.

To handle when the button is tapped, you need to add another map view delegate method. There are two steps to implement: first an action sheet is displayed so the user can select what they want to do: either show in maps, or find a route from the current location. Once the user selects an action, it's performed via an action sheet delegate method.

This requires another instance variable in the class extension category. Call it `_selectedStation`. While you're there, declare that you're implementing the `UIActionSheetDelegate` protocol. The class extension should now look like this:

```
@interface RWViewController () <MKMapViewDelegate,
UIActionSheetDelegate> {
    NSMutableArray *_stations;
    MKPolyline *_mapPolyline;
    RWStation *_droppedPin;
    RWLocationCallback _foundLocationCallback;
    RWRoute *_currentRoute;
    RWStation *_selectedStation;
}
```

Now add the delegate method to the file:

```
- (void)mapView:(MKMapView *)mapView
annotationView:(MKAnnotationView *)view
calloutAccessoryControlTapped:(UIControl *)control
{
    _selectedStation = (RWStation*)view.annotation;

    // 1
    UIActionSheet *sheet = [[UIActionSheet alloc]
        initWithTitle:@""
        delegate:self
        cancelButtonTitle:nil
        destructiveButtonTitle:nil
        otherButtonTitles:@"Show in Maps",
                    @"Route from current location",
```

```

        nil];

// 2
if (_droppedPin && view.annotation != _droppedPin) {
    [sheet addButtonWithTitle:@"Route from dropped pin"];
}

// 3
[sheet addButtonWithTitle:@"Cancel"];
sheet.cancelButtonIndex = sheet.numberOfButtons - 1;

// 4
[sheet showInView:self.view];
}

```

This is what the above method does:

1. It creates an action sheet with two buttons that are always available for every selected pin.
2. If there is a dropped pin and the selected annotation is not it (meaning it must be a station), then the method adds an option for routing from the dropped pin to the selected station.
3. The method adds a cancel button and tells the action sheet which button it is.
4. Finally, it shows the action sheet in the current view.

Now add the action sheet delegate method:

```

- (void)actionSheet:(UIActionSheet *)actionSheet
didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    // 1
    if (buttonIndex != actionSheet.cancelButtonIndex) {
        if (buttonIndex == 0) {
            // 2
            MKMapItem *mapItem = [_selectedStation mapItem];
            NSDictionary *launchOptions =
                @{@"MKLaunchOptionsDirectionsModeKey" :
                   MKLaunchOptionsDirectionsModeWalking};
            [mapItem openInMapsWithLaunchOptions:launchOptions];

        } else if (buttonIndex == 1) {
            // 3
            [self
             routeFromCurrentLocationTo:_selectedStation.coordinate];
        } else if (buttonIndex == 2) {

```

```
// 4
    [self routeFrom:_droppedPin.coordinate
        to:_selectedStation.coordinate];
}
}

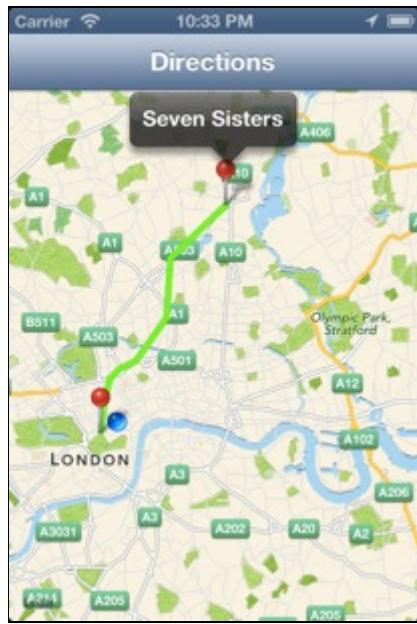
// 5
_selectedStation = nil;
}
```

So, to explain this method:

1. As long as the action sheet wasn't cancelled, an action needs to be performed.
2. In this case, the Maps app needs to be opened with the selected station and a request for walking directions to it. This is done in the same way as was used before to open multiple stations, but here you use the instance method on the station's `MKMapItem` and pass in the relevant value for the `MKLaunchOptionsDirectionsModeKey` in the launch options dictionary.
3. Here, a route has been requested to the selected station from the current location. All you need to do is call the routing method created earlier.
4. Here, a route has been requested to the selected station from the dropped pin. Again, it's as simple as calling the method created earlier.
5. Finally, the selected station is reset and is ready for re-use.

You're ready to go! Build and run the app and play around. Here are a few things to try out (note you will have to restart between testing each of these):

- Simulate the "London, England" location, tap on a station, then tap on the blue arrow and select "Route from current location."
- Drop a pin anywhere on the map, then tap on a station and blue arrow and select "Route from dropped pin."
- Drop a pin, then tap on it and the blue arrow and select "Route from current location."
- Select a station, tap on the blue arrow and select "Show in Maps."



Sir, I'd like some more buttons!

You're probably trying out the fun new routing capability and wondering how to clear a route. Let's add that functionality now. While you're at it, you might as well add the ability to open a route in Maps and ask it to do the routing for you. Maybe your users don't trust you and want a second opinion?

Open **RWViewController.m** and change the top of directions case in `setMapMode:` (up to, but not including section #6) to look like this:

```
case RWMapModeDirections: {
    self.title = @"Directions";
    self.navigationItem.leftBarButtonItem =
        [[UIBarButtonItem alloc]
            initWithTitle:@"Clear"
            style:UIBarButtonItemStyleBordered
            target:self
            action:@selector(clearDirections:)];
    self.navigationItem.rightBarButtonItem =
        [[UIBarButtonItem alloc]
            initWithTitle:@"Route in Maps"
            style:UIBarButtonItemStyleBordered
            target:self
            action:@selector(routeInMaps:)];
    ...
}
```

Here, the navigation buttons are set to two buttons called **Clear** and **Route in Maps**, which call `clearDirections:` and `routeInMaps:`, respectively. Now you need to implement these methods. Add the following code to the file:

```
- (void)routeInMaps:(id)sender {
    if (_currentRoute) {
        NSArray *mapItems =
            @[_currentRoute.fromStation mapItem],
            [_currentRoute.toStation mapItem];
        NSDictionary *launchOptions =
            @{@"MKLaunchOptionsDirectionsModeKey" :
                MKLaunchOptionsDirectionsModeDriving};
        [MKMapItem openMapsWithItems:mapItems
            launchOptions:launchOptions];
    }
}

- (void)clearDirections:(id)sender {
    self.mapMode = RWMapModeNormal;
}
```

Similar to opening a single station in Maps, `routeInMaps:` asks the Maps app to perform a routing operation using the start and end stations as the start and end points. Instead of asking for walking directions, this time the driving mode is selected.

The `clearDirections:` method is very simple, as all that needs to be done is to set the map mode back into normal. This invokes `setMapMode:`, which handles taking the UI back to where it was before.

Build and run the app and give those two new features a whirl!



Registering as a routing provider

One of the new features in iOS 6.0 is that you can register your app as a provider of directions. This means that when an iOS 6 device asks Maps to perform a routing operation, Maps will include your app in the list of routing providers displayed to the user. Maps will display a list of the apps that are already installed on the device, as well as apps available for purchase and download from the App Store. In a way, it's free advertising!

In order to ensure a manageable list instead of a huge unwieldy mess, every app must indicate in which region(s) of the world it can fulfill directions requests. The idea is that if you're asking for routing in New York, then you certainly don't want to try using an app that provides routing in Sydney, Australia! I'll explain how to specify the region you support shortly, but first, I'll show you how to become a routing provider.

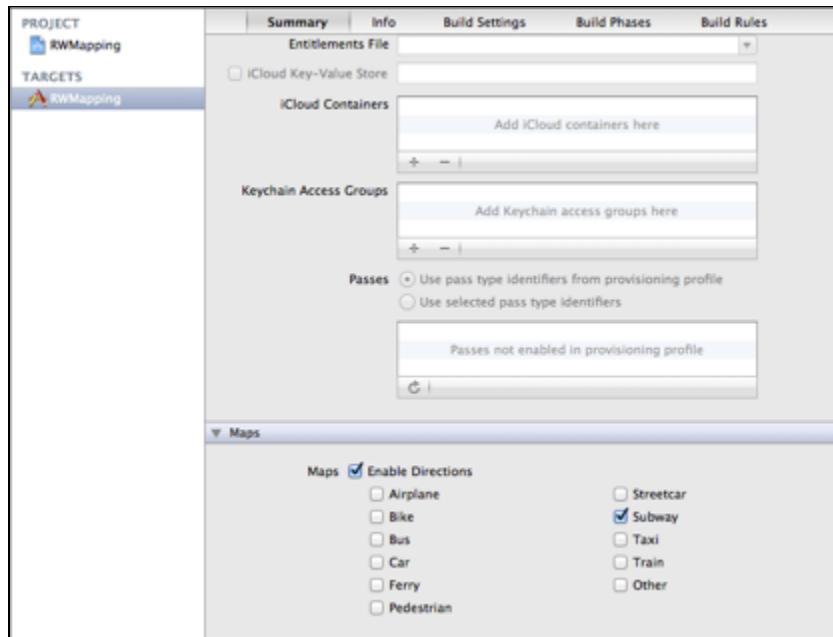
Changes to the project

There are a couple of things that you must do to the project in order to become a routing provider.

First you must edit parts of the **Info.plist** file. You can either edit it directly or use the handy GUI provided by Xcode. The latter is easier, so click on the project root in the project navigator and select the RWMapping target in the list that appears in the main editor area. Then select the Summary tab and scroll down to the bottom, where you should see a Maps section. Ensure it is open, then tick the Enable Directions box and then the Subway option.

You can select multiple categories if your app supports them, but this app just supports routing via the subway (or underground, as it's called in London).

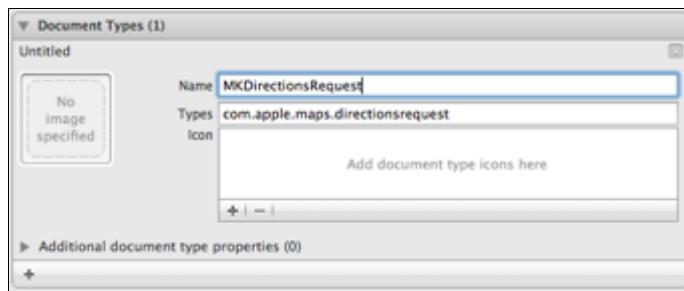
After you finish, it should look like this:



Now select the Info tab and open the Document Types section (click the arrow next to Document Types to expand the section). Then click the (+) button to will add a new document type to your app. Fill in the following details:

- **Name:** MKDirectionsRequest
- **Types:** com.apple.maps.directionsrequest

The box should now look like this:



You need to add this document type so that iOS knows your app accepts being opened with a special directions request file format. This is how it passes the information about start and end points to you. It's just like you would register your app to handle opening of text files for example.

Now onto the code...

Changes to the code

When a request comes in from Maps, your app delegate's `application:openURL:sourceApplication:annotation:` delegate method will be called. It will give you the start and end places as `MKMapItem` objects, and you are required to handle those as you see fit. If one of the places is the user's current location, then the `isCurrentLocation` flag will be set for that item.

In order to handle the request in this app, you first need to expose the routing methods within the view controller so that the application delegate can call them.

To do this, just put their method definitions in the public interface block. Open **RWViewController.h** and add the following code between the `@interface` and `@end` lines:

```
- (void)routeFrom:(CLLocationCoordinate2D)from
              to:(CLLocationCoordinate2D)to;
- (void)routeFromCurrentLocationTo:(CLLocationCoordinate2D)to;
- (void)routeToCurrentLocationFrom:(CLLocationCoordinate2D)from;
```

Now that the application delegate has access to the routing methods, it's time to add the delegate callback method executed when a routing request comes in. Open **RWAppDelegate.m** and add the following method to the end of the file:

```
- (BOOL)application:(UIApplication *)application
    openURL:(NSURL *)url
  sourceApplication:(NSString *)sourceApplication
    annotation:(id)annotation
{
    // 1
    if ([MKDirectionsRequest isDirectionsRequestURL:url]) {
        // 2
        MKDirectionsRequest *request =
        [[MKDirectionsRequest alloc] initWithContentsOfURL:url];
        MKMapItem *startItem = request.source;
        MKMapItem *endItem = request.destination;

        // 3
        if ([startItem isCurrentLocation]) {
            [self.mapViewController
            routeFromCurrentLocationTo:endItem.placemark.location.coordinate];
        } else if ([endItem isCurrentLocation]) {
            [self.mapViewController
            routeToCurrentLocationFrom:startItem.placemark.location.coordinate];
        } else {
            [self.mapViewController
```

```
        routeFrom:startItem.placemark.location.coordinate  
        to:endItem.placemark.location.coordinate];  
    }  
  
    // 4  
    return YES;  
}  
return NO;  
}
```

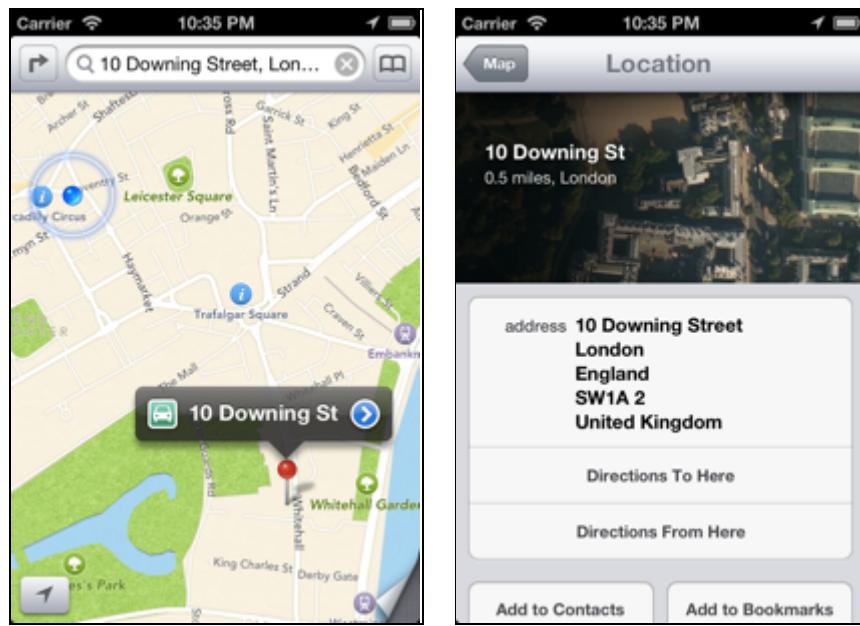
Here's is what this method does:

1. First it must check that the request that has come in is a direction request. If it's not, then the app doesn't support it.
2. It allocates a new `MKDirectionsRequest` object with the URL that has been passed in. This will handle the decoding of the data (start and end points) bundled in the URL for you.
3. It handles the various types of routing:
 - a. From the user's current location, to a specific place.
 - b. To the user's current location, from a specific place.
 - c. Both from and to a specific place.
4. Finally, it must return `YES` to signify that the URL has been handled.

And that's it. Yes, really, that's it! But now that you're done with coding, you'll want to try it out. It's not a complicated process, but I shall walk you through it none-the-less.

Build and run the app, and then immediately go out of your app and back to the home screen by pressing the home button (or press Shift-Cmd-H, **if you're on the simulator**). Then open the Maps app and turn on location simulation for London, England.

Search for "10 Downing Street, London, UK," select the pin that appears and click the blue arrow. Then select Directions To Here.



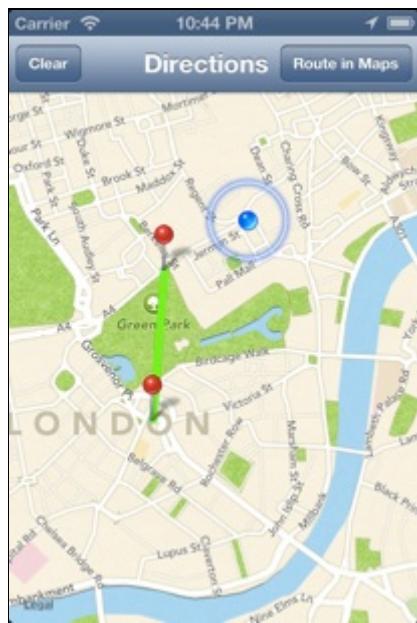
the screen that appears, select the third tab at the top, which looks like this: . Then select Route from the navigation bar.



A list will then appear where you should see RWMapping right at the top and a list of apps from the App Store at the bottom. Select the button entitled Route next to your app.



Finally, watch in awe as the app is opened and dropped into a routing screen!



What about the region definition?

You'll recall me mentioning that every app must declare for which region(s) of the world it can provide directions. You may then be wondering why you were able to see your app listed as a routing provider in Maps when you haven't yet defined the region your app services.

Well, it's because during development Apple ensured that your app will always appear in the list of available apps for *any* directions request made from your

device, so that you don't need to worry too much about setting the region during development. After all, you probably have plenty of other things on your mind. ☺

Once you're ready to submit your app however, you will need to define a region. This is done using a GeoJSON file. If you're curious then you can read the specification for it here:

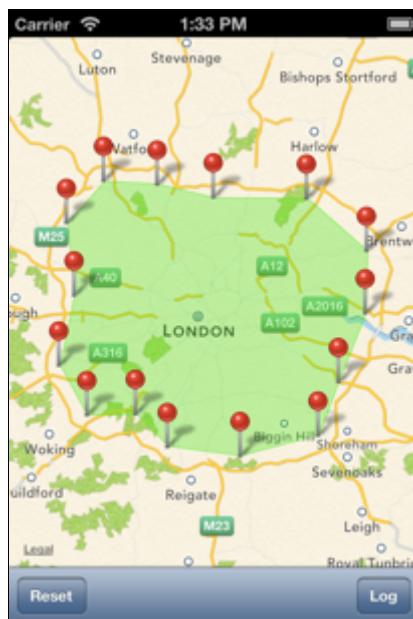
<http://geojson.org/geojson-spec.html>

All you really need to know is that a GeoJSON is a file containing a bunch of coordinates serialized using JSON. Apple requires the region to be defined using a MultiPolygon that contains an ordered list of coordinates joined to form the required region. It would be very hard to make one of these polygons by hand. So, Apple has kindly provided a sample project called RegionDefiner that is available with the iOS 6.0 documentation, but I've also included it for you in the resources for this chapter.

The project contains a sample GeoJSON file (called London.geojson) that defines the London region; however, you should familiarize yourself with how to create one yourself.

Open the RegionDefiner project and run it. You will be presented with a map and a couple of buttons at the bottom. If you start clicking and holding the map, pins will start to appear. Do this a few times and a region will start to build up. To reset the region, click the Reset button.

Find London, England and define a region like the one shown in the screenshot:



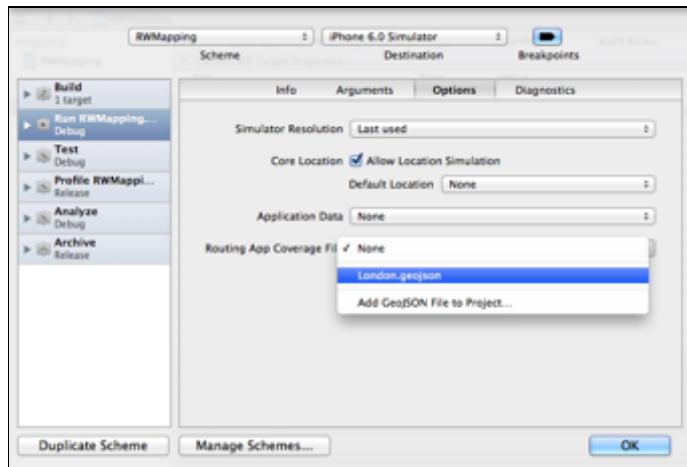
Now click the Log button. You should see something appear in the console that looks similar to this:

```
{ "type": "MultiPolygon",
  "coordinates": [
    [[[[-0.399576, 51.693038],
      [-0.272512, 51.688722],
      [-0.135004, 51.668216],
      [0.089534, 51.667136],
      [0.234005, 51.588259],
      [0.230524, 51.495158],
      [0.164381, 51.391007],
      [0.117384, 51.311650],
      [-0.072342, 51.280087],
      [-0.246403, 51.294238],
      [-0.322990, 51.343192],
      [-0.441351, 51.341017],
      [-0.507494, 51.417067],
      [-0.472682, 51.521159],
      [-0.488886, 51.630332],
      [-0.399576, 51.693038]]]
  ]
}
```

This is a GeoJSON-encoded version of the region you just defined, and this is what you need to upload to iTunes Connect when you upload your app for review. The beauty of uploading the file to iTunes Connect rather than bundling it with the app is twofold:

1. It enables Maps to include your app in a list of apps that might be able to service a user's routing request, even though they haven't yet been installed on the user's device. This is because by uploading the file to iTunes Connect, Apple will start including your app in search results within your specified locations.
2. It means that you don't have to submit an update to your app if your app suddenly becomes able to service a larger area. For instance, your app might get its data from a server and you have just added more stations and more train lines outside the previous area, and therefore a wider area that can be serviced. Instead of having to update the app to return information about this, you can just update the GeoJSON file.

If you would like to test if the file you've created works, then you can actually force your device to take the file into account instead of simply showing your app in the list for any directions request. To do this, click on Product>Edit Scheme... in the Xcode menu bar. Then select Run RWMapping on the left and the Options tab on the right. Under Routing App Coverage File, select the London.geojson file that I included in the project for you. The screen will look like this:



Now if you test the app, you'll find that routing requests outside the London area won't show RWMapping as an option. This is a useful way to test if your GeoJSON file is valid and/or set up correctly.

Where to go from here?

If you've made it this far, then you are now a true navigator of the MapKit world. Well done!

You've learned how to make an app that can read locations from a JSON file and display them on a map, along with a line that joins them together, making use of annotations and overlays. You've also learned how to add routing capabilities to an app such that a user can navigate an area, in this case using the subway system. And finally, you've learned how to tell the operating system that your app can provide routing for a certain region.

Check out the documentation provided by Apple for more information, and use your new knowledge to create awesome apps to help iOS users get around town!

Chapter 19: What's New with EventKit

By Kauserali Hafizji

“Good plans shape good decisions. That’s why good planning helps to make elusive dreams come true.” – Lester Robert Bittel

A well-planned event can turn out even better than imagined. But effective planning involves keeping track of a lot of details and coordinating between different people – in other words, a lot of crazy synchronization!

Over time, Apple has provided several frameworks and apps to help users make good plans and stay organized:

- **iOS 4:** Way back in iOS 4, Apple introduced EventKit, a framework that allows your apps to access and modify calendars and events displayed by the official Calendar app. Many apps have used this framework to allow users to conveniently schedule or view events directly within their apps.
- **iOS 5/OSX Mountain Lion:** In iOS 5, Apple introduced a new Reminders app, which helps users create and track to-do lists. In OS X Mountain Lion, Apple brought the Reminders app to the Mac. Now you can get reminders from all your Apple devices. It’s like having your own team of event coordinators... or your own Nag Squad!
- **iOS 6:** In the lastest release of iOS, Apple has introduced a new API that gives you access to reminders displayed in the official Reminders app (with user permission of course!) You can now programmatically create reminders for the user in your app, which can be extremely convenient for your users.

Since this book is focused on iOS 6, we will primarily focus on the new Reminders API. However, since the Reminders API is heavily integrated with the rest of EventKit, we will start with a refresher on EventKit, starting with how to add basic calendar events. If you’ve used EventKit before and are just looking for some help with the new API, feel free to flip ahead to the Remindes API section.

And speaking of planning – my plan is to help you become an EventKit master by the end of this chapter!

The conference planner app

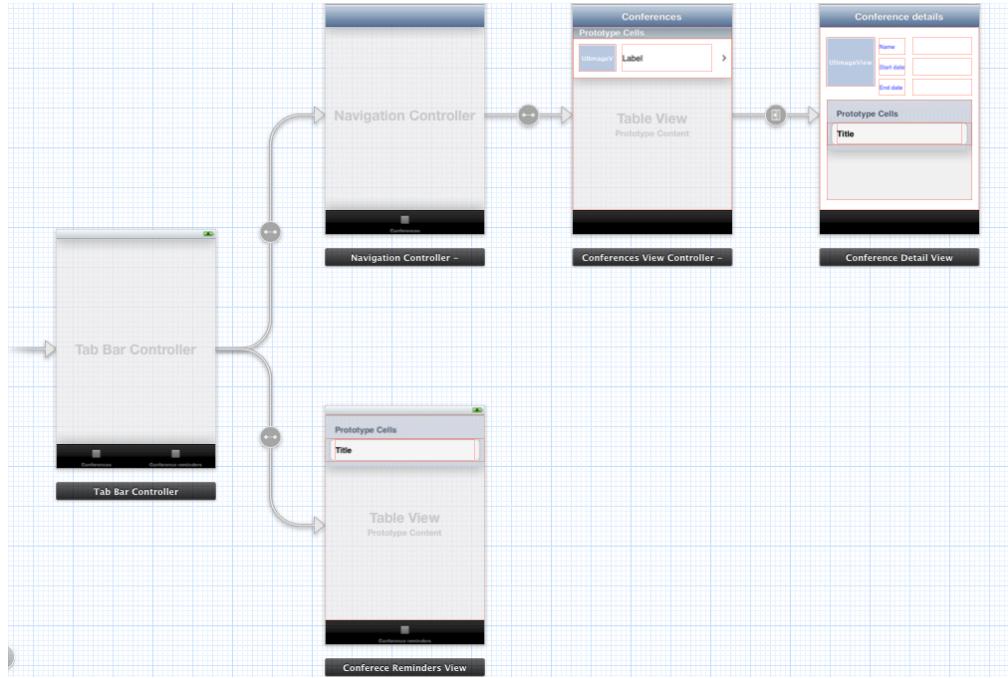
The project that will drive the instruction in this chapter is an app called *ConferencePlanner*, which will allow the user to keep track of iOS-related conferences happening throughout the year. The user will be able to do such things as add reminders to buy a ticket, or mark an event on his/her calendar for the day of the conference.

The app itself is very simple and consists of two screens. The first screen consists of a `UITabBarController` with two tabs: the first shows a list of conferences, and the second shows a list of all the reminders added by the user. To view the details of a given conference, the user simply taps on any row in the conferences list. The detail screen presents options to add events and reminders.



To keep the focus of this chapter on EventKit rather than creating a user interface, I've created a started project containing the user interface (but no EventKit code) in this chapter's resources. Open up the project and take a look around, and make sure you have a good understanding of how things work so far.

A good place to start is **MainStoryboard.storyboard**.



You'll notice that the initial view controller is a tab bar controller with two tabs:

1. The first goes to a Navigation Controller with the `ConferencesViewController` followed by a `ConferenceDetailViewController`.
2. The second tab goes to the `RemindersViewController`.

If you look at **AppDelegate.m**, you'll see that it loads a list of conferences from **conferences.plist** and stores the information into an array of `Conference` objects. This is then passed to the `ConferencesViewController`. Then:

- `ConferencesViewController` displays the information from the `Conference` objects in a table view. When a row is tapped, it passes a `Conference` to display to the `ConferenceDetailViewController`.
- `ConferenceDetailViewController` has a top section with information about the `Conference`, and a bottom section that is a table view of potential actions to take. There are two sections of actions: event related actions, and reminder related actions. Right now, the actions show up in the table view but tapping them does nothing.
- `ConferenceRemindersViewController` is pretty bare bones at the moment. It is just an empty table view.

There's also some icons for the conferences and a few other items in the project but that's it, it's ready for you to add some EventKit fun!

What is EventKit?

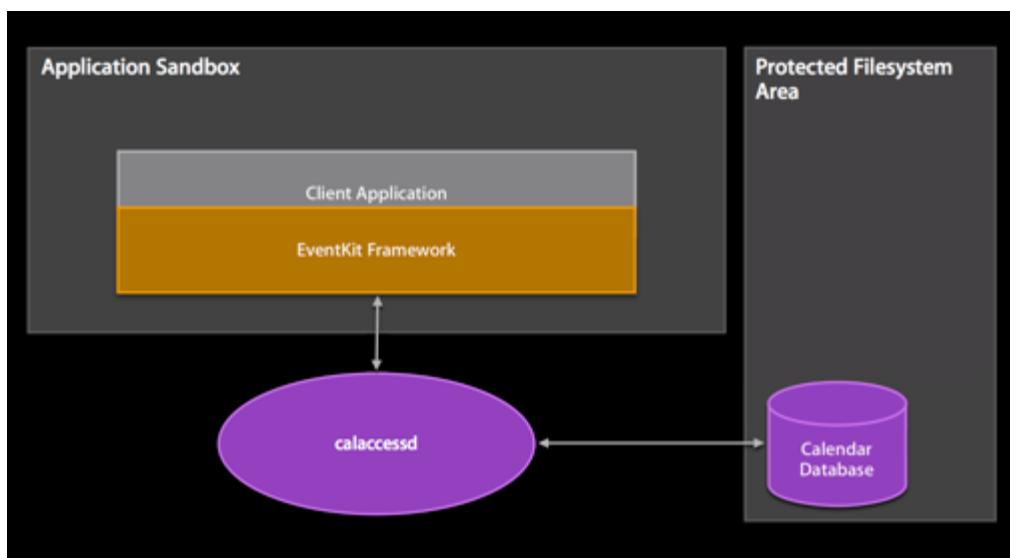
As mentioned in the introduction to this chapter, EventKit is a high-level API used to access calendar data on the device. Calendar data includes both standard events and reminders.

You should not think of EventKit as a synchronization API – it's not that low-level. Although it's true, whenever you make a change to an event or a reminder, the changes are automatically synchronized to the Calendar app (and also different devices with iCloud) for you automatically.

EventKit consists of two parts:

1. **The non-UI component** allows you to interact with event/reminder data programmatically, through a set of classes and methods.
2. **The UI component** consists of a bunch of view controllers. You can use these view controllers to display and edit events/reminders in a standard way.

In iOS, the calendar data lives in a calendar database, which is in a protected area of the file system, sandboxed out of your application. So how do you talk to it? Well, that's where EventKit comes into the picture.



As you can see in the illustration above, the EventKit library talks to a daemon called `calaccesssd`, which communicates with the calendar database. On iOS, except for `calaccesssd` and the sync daemons, nothing else can interact with the calendar database directly.

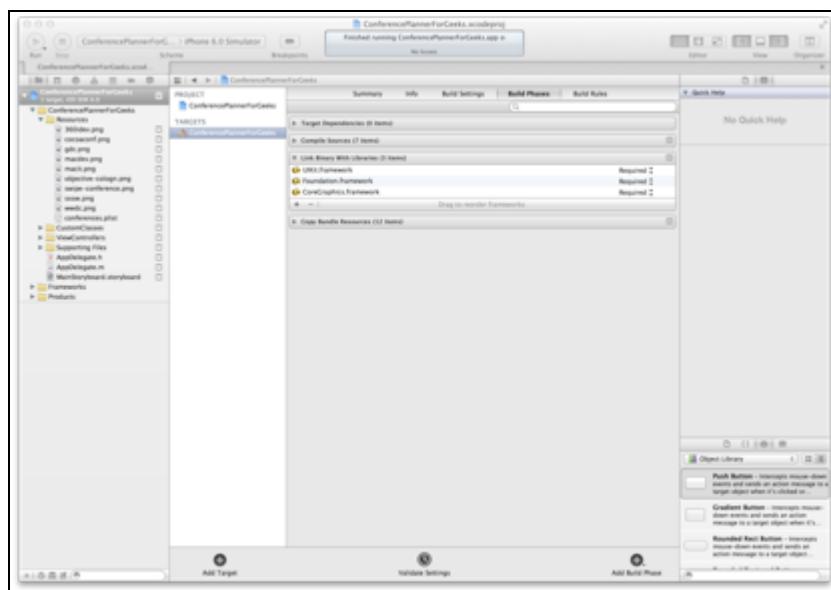
This way, even though multiple apps might try to access the calendar data at the same time, a single source can control it all and keep things in sync.

Adding EventKit to your app

Now that you know what EventKit is and how it works, let's get back to ConferencePlanner and start putting some EventKit APIs to good use.

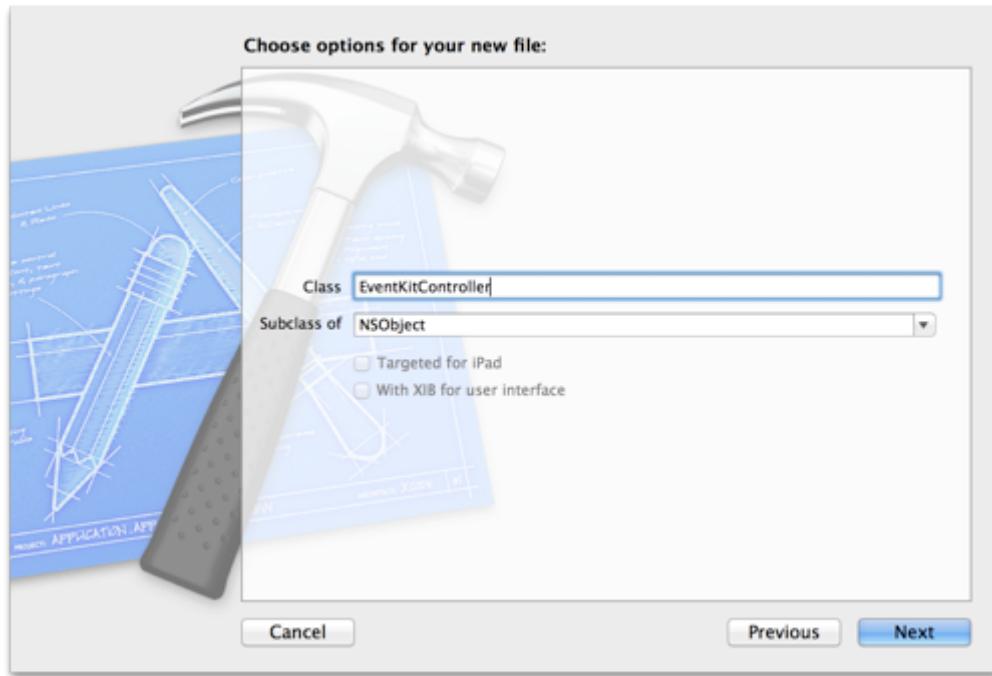
Open the starter kit project using Xcode 4.5. First, add the EventKit framework. To do this, open the **ConferencePlannerForGeeks.xcodeproj** file, tap on the project root in the Project Navigator on the left, select the ConferencePlannerForGeeks target in the middle pane, and then go to the Build Phases tab.

Expand the Link Binary With Libraries section, click the (+) button at the bottom of the section, and add `EventKit.framework` to the project. Do a quick compile to make sure everything works.



Now that you have EventKit integrated into your project, let's create a new group for the new EventKit code you'll write. As my authorship of this chapter might imply, I'm a big fan of organizing things. ☺

To do this, right-click on the ConferencePlannerForGeeks folder in the Project Navigator, select New Group, and name it **EventKitClasses**. Now right-click on the newly-created folder and select New File ..., then the Objective-C class template, and click Next. Name the class `EventKitController` and make it a subclass of `NSObject`.



Your class has been created – it's time to write some code!

First add the following include statement to the top of **EventKitController.h**, below the existing #import line:

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

You've imported the EventKit header – check! Next, create few properties by adding the following code between the @interface and @end:

```
@property (strong, readonly) EKEventStore *eventStore;
@property (assign, readonly) BOOL eventAccess;
@property (assign, readonly) BOOL reminderAccess;
```

Now switch to **EventKitController.m** and create the initializer for your class – init (the code should go between @implementation and @end):

```
- (id) init {
    self = [super init];
    if (self) {
        _eventStore = [[EKEventStore alloc]
                      init];
        [_eventStore requestAccessToEntityType:EKEntityTypeEvent
                           completion:^(BOOL granted, NSError *error) {
            if (granted) {
                _eventAccess = YES;
            } else {
```

```
        NSLog(@"Event access not granted: %@",  
              error);  
    }  
};  
[_eventStore  
    requestAccessToEntityType:EKEEntityTypeReminder  
    completion:^(BOOL granted, NSError *error) {  
        if (granted) {  
            _reminderAccess = YES;  
        } else {  
            NSLog(@"Reminder access not granted: %@",  
                  error);  
        }  
    }];  
}  
return self;  
}
```

Let's pause here for a moment and talk about `EKEventStore`. An object of this class is your connection to the calendar database. When you instantiate one of these objects, you are opening a connection to that database. This in turn will start the `calaccesssd` daemon if it is not currently running. Most of the time though, the daemon will be running in the background.

Since creating an object of `EKEventStore` could lead to starting up a daemon, which is not a lightweight operation, you generally want to have this object around for as long as possible and reuse it, instead of creating a new instance whenever you need it.

Initializing an instance of `EKEventStore` is really easy – all you have to do is specify the entity types you want access to. In the ConferencePlanner app, to get access to both event and reminder data, you are going to specify `EKEntityMaskEvent` and `EKEntityMaskReminder` as the entity types.

At the time of writing this chapter, after you initialize the event store, you need to separately request access to the entity types your app needs to access by calling `requestAccessToEntityType:completion`. This is a new API introduced in iOS 6, added in order to more safely guard a user's privacy. The idea is you call these methods to request access to the calendar or reminders from the user, much like when an app asks to access the user's location.

If this is the first time you've called this method, the app will present a popup asking the user for their permission. After the user chooses whether to give access or not, your callback will be called. In future runs, if the user has already decided to give your app access or not, your callback will be called immediately with the user's previous decision. The user can also change their minds by changing the permission in Settings.

It is important to call this method because otherwise attempting to get calendars or store events in calendars will fail. Here, you simply record whether we've been given permission or not for use later.

Using the EventKitController class

Now let's make use of your newly-created `EventKitController` class. Open `ConferenceDetailViewController.m` and import `EventKitController.h` below the other `#import` statement:

```
#import "EventKitController.h"
```

Add an instance variable for the `EventKitController` class to the existing class extension:

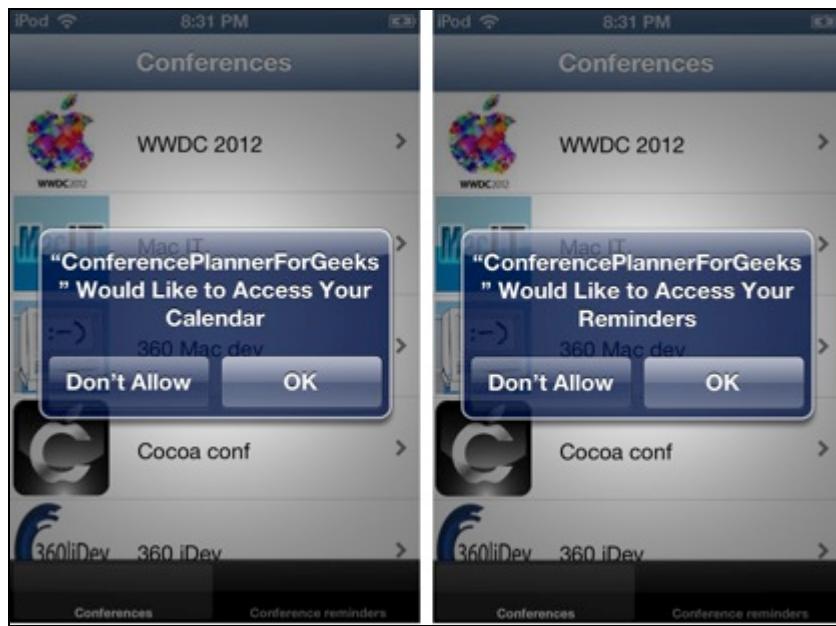
```
@interface ConferenceDetailViewController () {
    . . .
    EventKitController *_eventKitController;
}
```

Next, initialize the new variable in `initWithCoder`:

```
- (id) initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder:aDecoder];
    if (self) {
        . .
        _eventKitController = [[EventKitController alloc] init];
    }
    return self;
}
```

Build and run the app. As soon as you move to the detail screen (by selecting a conference from the main screen), you will see a two system-generated alert that ask the user for permission to access calendar and reminder data.

For now select OK for each alert. You will learn more about these popups in detail in the sections to come.



Hurray! You have successfully created an `EKEventStore` object; this will be your connection to the calendar database. If you think that was easy, creating events and reminders is going to be a walk in the park!

Before we go any further, let's discuss two important classes in EventKit: `EKCalendar` and `EKEvent`.

EKCalendar

Let's start by discussing `EKCalendar`.

Every event/reminder is stored in a calendar. iOS supports multiple types of calendars such as Exchange, CalDAV, MobileMe and local. Whenever you save an event/reminder to the calendar, it is synced automatically to the underlying calendar store.

iOS supports read-only and read-write calendars; however, most calendars are read-write, with the exception of subscribe calendars and the birthday calendar, which was introduced with iOS 4.0.

There is also the concept of a default calendar that can be specified through the Settings app. Every time you create an event/reminder through the Calendar or Reminders app, it is stored in this default calendar. For example, if you add an event – say “Meeting at 2pm” – to the Calendars app and your default calendar is iCal, the event will automatically sync with your Mac and iCloud accounts.

`EKCalendar` is the object that represents a single calendar, whether it be your Work calendar, your Home calendar, or something else. Getting a list of `EKCalendar`s is

simple - you can simply ask the `EKEventStore` for a list of calendars by entity type. You can then loop through them to find the ones you want.

Below is an example of looping through the calendars to find the writable ones:

```
NSArray * allCalendars = [_eventStore  
    calendarsForEntityType:EKEntityMaskEvent |  
    EKEntityMaskReminder];  
NSMutableArray * writableCalendars = [NSMutableArray array];  
for (EKCalendar * calendar in allCalendars) {  
    if (calendar.allowsContentModifications) {  
        [writableCalendars addObject:calendar];  
    }  
}
```

Alternatively, if you just want to access the default calendar for new events, you can get it with a simple property on `EKEventStore`:

```
EKCalendar *calendar = eventStore.defaultCalendarForNewEvents;
```

There is also a similar property called `defaultCalendarForNewReminders` that returns the default calendar for reminders.

EKEvent

Now let's discuss the second class you should know about – `EKEvent`.

An event is something that happens on a fixed date and time – for example, a doctor's appointment might be on the 20th of September at 10:00 am. The user can set this event on his/her calendar, and an `EKEvent` class would represent the event.

An `EKEvent` has multiple properties associated with it, as you can see in the diagram below:



Here is a brief description of the important ones:

- **Start and end dates**: These properties specify when the event is going to start and when it will end. You can also specify an event to be an all day event.
- **Repeat**: This property adds recurrence rules to the event. This is useful when you want to create an event that repeats. For example, you want an event called "Meeting with Ray" to happen every week. So, instead of setting individual events on your calendar, you can set the recurrence to weekly and new events will automatically be added for you every week from the start date!

Note from Ray: Please don't sign me up for any weekly meetings, I try to avoid those 😊

- **Alarms**: You can set alarms to remind you of events. Quite useful for the chronically late!

It's easy to create an event. Here's an example:

```
//1
EKEvent *event = [EKEvent eventWithEventStore:eventStore];
event.title = @"Hello world!";
event.startDate = startDate;
event.endDate = endDate;
event.calendar = eventStore.defaultCalendarForNewEvents;
```

```
//2  
NSError *err;  
BOOL saved = [eventStore saveEvent:event span:EKSpanThisEvent  
commit:YES error:&err];  
  
if (!saved) {  
    NSLog(@"Error creating the event");  
}
```

The first section creates an instance of `EKEvent` from the event store, and sets a few properties on it like the start date, end date, and title. Note that it also specifies the calendar to create the event in – in this case, the default calendar for new events.

Note: The `startDate`, `endDate` and `calendar` are mandatory properties of an `EKEvent` and you cannot save an event unless these values are specified. EventKit will throw an error if you try to save the event without setting these properties.

Second, it saves the event to the calendar database using the `saveEvent:span:commit:error:` method of `EKEventStore`. When you store a new event, the `span` property is always `EKSpanThisEvent`. However, this value will change when you delete or modify the event. You'll see how to use this in detail in the sections to come.

The above method returns a Boolean value that specifies whether the save was successful. If not, the resulting error is available via the `NSError` object.

OK, enough theory - it's time to add some events in to your app!

Adding Events

Start by opening `EventKitController.m` and add the following code to the end of the file (before `@end`):

```
- (void) addEventWithName:(NSString*) eventName  
                      startTime:(NSDate*) startDate  
                     endTime:(NSDate*) endDate {  
  
    if (!_eventAccess) {  
        NSLog(@"No event access!");  
        return;  
    }
```

```
//1. Create an Event
EKEvent *event = [EKEvent
    eventWithEventStore:self.eventStore];

//2. Set the title
event.title = eventName;

//3. Set the start and end date
event.startDate = startDate;
event.endDate = endDate;

//4. Set an alarm (This is optional)
EKAlarm *alarm = [EKAlarm alarmWithRelativeOffset:-1800];
[event addAlarm:alarm];

//5. Add a note (This is optional)
event.notes = @"This will be exciting";

//6. Specify the calendar to store the event
event.calendar=self.eventStore.defaultCalendarForNewEvents;

NSError *err;
BOOL success = [self.eventStore
    saveEvent:event
    span:EKSpanThisEvent
    commit:YES error:&err];

if (!success) {
    NSLog(@"There was an error saving event: %@", err);
}
}
```

This method creates a new event and saves it to the default calendar. What's new is the addition of `EKAlarm`!

As discussed before, an alarm is used to alert the user of an upcoming event. Here, you add an alarm with a relative offset of -30 minutes. This relative offset is added to the start time of the event. So in this case, the alarm will ring half an hour before the start time.

Next, you need to add the method definition for your new method so that it is visible to other classes that use the `EventKitController` class. So open **EventKitController.h** and add the following before the @end:

```
-(void)addEventWithName:(NSString*)eventName
startTime:(NSDate*)startDate endTime:(NSDate*)endDate;
```

Then, switch to **ConferenceDetailViewController.m** and find `tableView:didSelectRowAtIndexPath:`. Replace the first comment in the method (that says “add code to add an event”) with the following:

```
[_eventKitController addEventWithName:self.conference.name  
startTime:self.conference.startDate  
endTime:self.conference.endDate];
```

Now, whenever the user taps on the first row of the conference detail screen, an event will be created and added to the default calendar. The title of the event will be the name of the conference, and the start and end will dates correspond to the start and end dates of the conference.

Go ahead and try it out! Build and run the application, open the detail page and tap on the row that reads, “Add event to Calendar.” Now open the Calendar application and find the date on which the event was scheduled. Here’s how it looks for WWDC:



Adding Recurrence Rules

Now that you know how to create simple events, surely you’re ready for something a bit more difficult, like setting up a recurrence rule.

A recurrence rule defines how an event will repeat. If you create a new event using the Calendar app, you can select from a bunch of repetition rules such as everyday, every week, etc. However, the API allows you to do far more complex things, like set an event to repeat every third Friday of every month.

Here's an example of setting up a simple recurrence rule:

```
//1
EKRecurrenceRule *recurrenceRule =
    [[EKRecurrenceRule alloc]
     initRecurrenceWithFrequency:EKRecurrenceFrequencyMonthly
     interval:1 end:nil];

//2
event.recurrenceRules = @[recurrenceRule];
```

Begin by creating an `EKRecurrenceRule` instance. Specify the recurrence frequency as `EKRecurrenceFrequencyMonthly` and the interval as 1. What this means is that you want the event to repeat once every month. Since the end parameter is `nil`, this event will repeat once every month until the end of time. (Or at least, until the end of Apple!)

If you want the recurrence to have an end, you can set that using `EKRecurrenceEnd`. There are two ways you can set the end date of a recurrence rule. You can specify the exact date you want the event to end; for example, you want to have a meeting every month until the 12th of June 2012. In code, this would translate to the following, where `endDate` would contain the actual end date:

```
EKRecurrenceEnd *end = [EKRecurrenceEnd
    recurrenceEndWithEndDate:endDate];
```

Another way to end a recurring event is to limit the number of times that event can occur, like so:

```
EKRecurrenceEnd *end = [EKRecurrenceEnd
    recurrenceEndWithOccurrenceCount:5];
```

In the above example, the event will end after 5 occurrences.

You now know how to add a recurrence rule to an event. Now modify **EventKitController.m** to add a recurring event, using the following code:

```
- (void) addRecurringEventWithName:(NSString*) eventName
                           startTime:(NSDate*) startDate
                           endTime:(NSDate*) endDate {

    if (!_eventAccess) {
        NSLog(@"No event access!");
        return;
    }

    //1. Create an Event
```

```
EKEvent *event = [EKEvent
                  eventWithEventStore:self.eventStore];

//2. Set the title
event.title = eventName;

//3. Set the start and end date
event.startDate = startDate;
event.endDate = endDate;

//4. Set an alarm (This is optional)
EKAlarm *alarm = [EKAlarm
                  alarmWithRelativeOffset:-1800];
[event addAlarm:alarm];

//5. Add a note (This is optional)
event.notes = @"This will be exciting";

//6. Add the recurrence rule.
EKRecurrenceRule *rule =
[[EKRecurrenceRule alloc]
 initRecurrenceWithFrequency:
 EKRecurrenceFrequencyYearly
 interval:1 end:nil];
event.recurrenceRules = @[rule];

//7. Specify the calendar to store the event to
event.calendar =
    self.eventStore.defaultCalendarForNewEvents;

NSError *err;
BOOL success = [self.eventStore
                 saveEvent:event span:EKSpanThisEvent
                 commit:YES error:&err];

if (!success) {
    NSLog(@"There was an error saving event: %@", err);
}
}
```

This method creates an event with the `name`, `startDate` and `endDate` specified in those parameters. It also adds a recurrence rule to make the event repeat every year indefinitely.

This makes sense for ConferencePlanner. That is, if the user wants to attend a conference every year, he/she could choose to add this sort of an event.

As before, switch to **EventKitController.h** and add the method definition as follows:

```
- (void)addRecurringEventWithName:(NSString*)eventName  
startTime:(NSDate*)startDate endTime:(NSDate*)endDate;
```

Now open **ConferenceDetailViewController.m** and replace the comment that says, "Add code to add a recurring event" in `tableView:didSelectRowAtIndexPath:` with:

```
[_eventKitController  
    addRecurringEventWithName:self.conference.name  
    startTime:self.conference.startDate  
    endTime:self.conference.endDate];
```

Build and run the app. When you tap on the second row of the list of actions in the detail page, a recurring event is added to your default calendar. Open the Calendar app and check for the event to verify that it works!



Deleting Events

Up to this point, you've only seen how to create events. You've only been continuously adding data to the calendar database. But how do you remove events that you've added?

Well, before you can remove an event you have to find the event you want to remove.

One way to find an event is by using a property called the `eventIdentifier`. Every event has a unique identifier, which is assigned when the event is saved to the database. However, unfortunately this identifier can change. In cases where the

user moves the event to another calendar, the event identifier will change. In a situation like this, you want to fall back on using other properties such as start date, title, and end date to find the right event.

EventKit comes with a predicate that you should use to fetch events. The predicate requires you to provide a start date, end date, and an array of calendars to search. Because the predicate uses only three fields, you always need to process the array later to do a more detailed search.

The predicate execution is a blocking call and it should be done asynchronously. To do it, you're going to use one of my favorite iOS technologies, dispatch queues.

Open **EventKitController.m** and set up a class extension with a variable for the dispatch queue as follows (add the code before the `@implementation` line):

```
@interface EventKitController() {
    dispatch_queue_t _fetchQueue;
}
@end
```

Initialize the variable in `init` by adding the following code below the `_eventStore` initialization:

```
_fetchQueue =
dispatch_queue_create("com.conferencePlannerForGeeks.fetchQueue",
DISPATCH_QUEUE_SERIAL);
```

Next, add the following method:

```
- (void) deleteEventWithName:(NSString*) eventName
                      startTime:(NSDate*) startDate
                     endTime:(NSDate*) endDate {

    if (! _eventAccess) {
        NSLog(@"No event access!");
        return;
    }

    dispatch_async(_fetchQueue, ^{
        //1. Create a predicate to find the EKEvent to delete
        NSPredicate *predicate =
        [self.eventStore
            predicateForEventsWithStartDate:startDate
            endDate:endDate
            calendars:
            @[_self.eventStore.defaultCalendarForNewEvents]];
    });
}
```

```
NSArray *events =
[_self.eventStore eventsMatchingPredicate:predicate];

//2. Post filter the events array
NSPredicate *titlePredicate =
[NSPredicate
predicateWithFormat:@"title matches %@", 
eventName];
events = [events
filteredArrayUsingPredicate:titlePredicate];

//3. Delete each of these events
NSError *err;
for (EKEvent *event in events) {
    [_self.eventStore
removeEvent:event
span:event.hasRecurrenceRules ?
EKSpanFutureEvents:EKSpanThisEvent
commit:NO error:&err];
}

BOOL success = [_self.eventStore commit:&err];
if (!success) {
    NSLog(@"There was an error deleting event");
}
});
if (!_eventAccess) {
    NSLog(@"No event access!");
    return;
}
}
```

This method takes three arguments: the name, start time, and end time of the conference whose events need to be deleted.

Here is a step-by-step explanation of what's happening in the method:

1. First a predicate is created and the start date and end date are set to be the start and end dates of the conference. Next, the calendar to be searched is set to the default calendar.
2. Once the predicate is executed, the results are filtered using another predicate. This predicate matches the title of the event with the name of the conference.
3. The method then loops through the results array from the second filter to delete each event using the `removeEvent:span:commit:error:` method of `EKEventStore`. Notice that the span is set to `EKSpanFutureEvents` in case the event being deleted

has a recurrence rule. If the event is a repeating event, you want all occurrences of the event to be removed and not just the first one.

Add the method definition to **EventKitController.h** as follows:

```
- (void)deleteEventWithName:(NSString*)eventName  
                      startTime:(NSDate*)startDate endTime:(NSDate*)endDate;
```

Finally open **ConferenceDetailViewController.m** and replace the comment "Add code to delete an event" in `tableView:didSelectRowAtIndexPath:` with the following:

```
[_eventKitController  
    deleteEventWithName:self.conference.name  
    startTime:self.conference.startDate  
    endTime:self.conference.endDate];
```

Run the application and see if selecting the "Remove all events" action deletes all the events that you added before.

Creating Reminders

So far, almost everything we've discussed has been possible back in iOS 4. Now we're on to the fun new stuff in iOS 6 – reminders!

The reminders API is a part of EventKit, and like events, reminders get saved to a calendar. You can add recurrence rules and alarms just as you do with events.

A key factor to keep in mind is that a reminder is not time-bound, in the sense that it does not require a start and end time. For example, a task like "Schedule a doctor's appointment" does not have a start and end time, though it may have a deadline, which is something different.

Creating reminders is easier than creating events, though the basic process is similar. Here's an example:

```
//1  
EKReminder *reminder =  
    [EKReminder reminderWithEventStore:eventStore];  
reminder.title = @"Creating my first reminder";  
reminder.calendar =  
    [eventStore defaultCalendarForNewReminders];  
//2  
NSError *err;  
BOOL success =  
    [eventStore saveReminder:reminder  
                  commit:YES error:&err];  
if (!success) {
```

```

    NSLog(@"Error creating reminder");
}

```

Let me walk you through this step-by-step:

1. Create an instance of `EKReminder` using the event store. Set the title and the calendar. Store the reminder in the user's default reminders calendar.
2. Finally, save the event using the `saveReminder:commit:error:` method of `EKEventStore`.

Creating a reminder is easy, isn't it? As with events, the user can specify the default calendar to store reminders. This is done in the Settings app. The screenshots below show the steps that you take to change the default calendar for reminders.



Let's try this out! Open `EventKitController.m` and add the following method:

```

- (void) addReminderWithTitle:(NSString*) title
                        dueTime:(NSDate*) dueDate {

    if (! _reminderAccess) {
        NSLog(@"No reminder access!");
        return;
    }

    //1.Create a reminder
    EKReminder *reminder =
        [EKReminder
            reminderWithEventStore:

```

```
        self.eventStore];

    //2. Set the title
    reminder.title = title;

    //3. Set the calendar
    reminder.calendar =
        [self calendarForReminders];

    //4. Extract the NSDateComponents from the dueDate
    NSCalendar *calendar =
        [NSCalendar currentCalendar];

    NSUInteger unitFlags =
        NSEraCalendarUnit |
        NSYearCalendarUnit |
        NSMonthCalendarUnit |
        NSDayCalendarUnit;

    NSDateComponents *dueDateComponents =
        [calendar components:unitFlags
            fromDate:dueDate];

    //5. Set the due date
    reminder.dueDateComponents = dueDateComponents;

    //6. Save the reminder
    NSError *err;
    BOOL success = [self.eventStore
                    saveReminder:reminder
                    commit:YES error:&err];
    if (!success) {
        NSLog(@"There was an error saving the reminder %@", err);
    }
}
```

This method takes two parameters: the title of the reminder and the due date. The due date specifies a deadline for the reminder, but it's not the same as an end date.

Notice how the year, month and day are extracted from the `dueDate` and set as the `dueDateComponents` of the reminder.

Now add the following `#define` below the `#import` line at the top of the file:

```
#define kRemindersCalendarTitle @"Conference reminders"
```

The above defines a constant value identifying the title of your app's default reminders calendar. You'll need that define in the next bit of code, where you add a method to the file that was called from the previous method:

```
- (EKCalendar*) calendarForReminders {

    //1
    for (EKCalendar *calendar in
        [self.eventStore
            calendarsForEntityType:EKEntityTypeReminder]) {

        if ([calendar.title
            isEqualToString:kRemindersCalendarTitle]) {
            return calendar;
        }
    }

    //2
    EKCalendar *remindersCalendar =
        [EKCalendar
            calendarForEntityType:EKEntityTypeReminder
            eventStore:self.eventStore];

    remindersCalendar.title = kRemindersCalendarTitle;
    remindersCalendar.source =
        self.eventStore.defaultCalendarForNewReminders.source;

    NSError *err;
    BOOL success = [self.eventStore
                    saveCalendar:remindersCalendar
                    commit:YES error:&err];
    if (!success) {
        NSLog(@"There was an error creating the reminders
calendar");
        return nil;
    }
    return remindersCalendar;
}
```

This method returns an instance of `EKCalendar`. You use this calendar to add new reminders. First the method searches all the available calendars to see if you have one that matches `kRemindersCalendarTitle`, which is set to "Conference reminders." If it finds a matching calendar, it just returns an instance of that calendar. Otherwise, it creates a new calendar, saves it to the calendar database, and returns a pointer to it.

Add the method definition for `addReminderWithTitle:dueTime:` to **EventKitController.h** as follows:

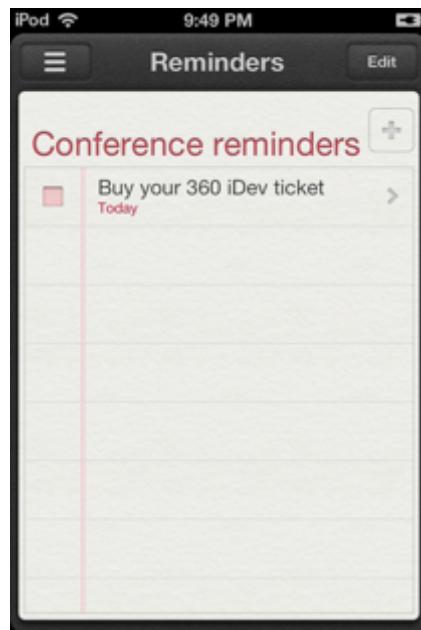
```
- (void)addReminderWithTitle:(NSString*)title  
dueTime:(NSDate*)dueDate;
```

Switch to **ConferenceDetailViewController.m**. Find the comment that says, "code to add a reminder to buy a ticket" in `tableView:didSelectRowAtIndexPath:` and replace it with:

```
[_eventKitController  
    addReminderWithTitle:  
        [NSString stringWithFormat:@"Buy your %@ ticket",  
            self.conference.name]  
    dueTime:[NSDate date]];
```

This adds a reminder for the user to buy his/her ticket, with the due date as the current date.

Give the app a quick run. Tap on the fourth action on the detail screen and then open the Reminders app. You should see a new list called "Conference reminders" with a new reminder.



Next, you're going to use the same method and add another reminder, this time to remind the user to pack their bags. What will be different here is that the due date of this reminder will be a day before the start of the event.

Using the method just created, you can easily add a reminder and set its due date, but how do you set the due date to be exactly a day before the start of the event?

Date math

It should be simple: there are 86,400 ($60 * 60 * 24$) seconds in a day, aren't there? As WWDC 2012 Session #304 points out, this is usually true but not always because of our old friend daylight savings. The 86,400 number will probably be wrong twice every year on inconsistent days. So how on earth do you find the right day?

You are going to let `NSCalendar` figure it out for you. Note that EventKit currently only supports the Gregorian calendar for reminders.

Replace the last comment in **ConferenceDetailViewController.m**'s `tableView:didSelectRowAtIndexPath:` (titled "code to send a reminder to pack your bags") with the following:

```
//1
NSDateComponents *calendar = [NSDateComponents currentCalendar];

//2
NSDateComponents *oneDayComponents =
    [[NSDateComponents alloc] init];

//3
oneDayComponents.day = -1;

//4
NSDate *dayBeforeTheEvent = [calendar
    dateByAddingComponents:oneDayComponents
        toDate:self.conference.startDate
        options:0];

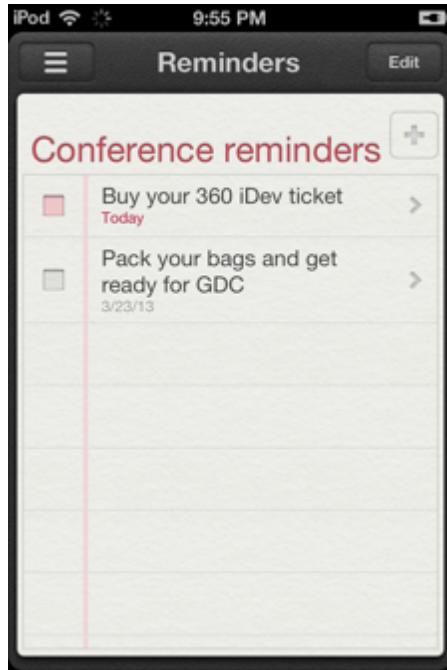
//5
[_eventKitController
    addReminderWithTitle:
        [NSString
            stringWithFormat:
                @"Pack your bags and get ready for %@", self.conference.name]
    dueTime:dayBeforeTheEvent];
```

Here is a step-by-step breakdown of this code:

1. First create an instance of `NSCalendar`.
2. Next create an instance of `NSDateComponents`.
3. Set the day component to -1.

4. Add the `NSDateComponents` to the start date of the conference. This takes you to a day before the start date of the conference.
5. Finally, call `addReminderWithTitle:dueTime:` with the calculated date.

Build and run the app to make sure it works correctly. Now, if you go to a detail view and select the fifth action, “Add a reminder to pack your bags,” you should see that reminder in your “Conference reminders” list.



Fetching reminders

So you've gone through the process of adding various reminders using your app, and deleting reminders. But how do you see the reminders you've added from within your own app?

In this part of the chapter, you'll learn how to fetch reminders from the calendar database. You'll work on `ConferenceRemindersViewController`, which represents the second tab on the main screen. It presents all the reminders that you have added to the “Conference reminders” calendar in a table view.

Let's move straight to the code! Open `ConferenceRemindersViewController.m` and add the following code at the top, right below the `#import` line, replacing the existing empty class extension:,

```
#import "EventKitController.h"

@interface ConferenceRemindersViewController : UIViewController
    EventKitController *_eventKitController;
```

```
}
```

```
@end
```

The above includes **EventKitController.h** and declares an `EventKitController` instance variable.

Instantiate the above variable within the `if` condition in `initWithCoder::`

```
_eventKitController = [[EventKitController alloc] init];
```

Now open **EventKitController.h** and add the following property:

```
@property (strong) NSMutableArray *reminders;
```

This property will be used to store the fetched reminders.

Next add a method to **EventKitController.m** to return an array of all the reminders that have been added to the “Conference reminders” calendar. But that method will refer to a constant value. So first add that constant value just above the `@implementation` line in **EventKitController.m**:

```
NSString *const RemindersModelChangedNotification =
@"RemindersModelChangedNotification";
```

Now add the new method to the end of the file:

```
- (void) fetchAllConferenceReminders {
    //1
    NSPredicate *predicate =
        [self.eventStore
            predicateForRemindersInCalendars:
                @[[self calendarForReminders]]];
    //2
    [self.eventStore
        fetchRemindersMatchingPredicate:predicate
        completion:^(NSArray *reminders){
            self.reminders =
                [reminders mutableCopy];
            [[NSNotificationCenter defaultCenter]
                postNotificationName:
                    RemindersModelChangedNotification
                object:self];
        }];
}
```

EventKit comes with a predicate to fetch reminders from the calendar database called `predicateForRemindersInCalendars`. This predicate requires an array of calendars from which to fetch reminders. In this case, you just have one.

The method you added creates a predicate and sets the calendar. To execute the predicate it uses a new method introduced in iOS 6.0, `fetchRemindersMatchingPredicate:completion`. This method fetches reminders asynchronously. Once completed, the completion block gets called with an array of reminders. The method then stores this array in the property you declared.

Since the user is free to add or remove a reminder from the "Conference reminders" calendar using the Reminders app, there should be a way in which you can know if any reminders have been added, modified or removed. EventKit sends a broadcast notification when there is a change to a calendar.

Add the following methods to **EventKitController.m** to start and stop listening to such broadcasts:

```
- (void) startBroadcastingModelChangedNotifications {
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(fetchAllConferenceReminders)
        name:EKEEventStoreChangedNotification
        object:self.eventStore];
}

- (void) stopBroadcastingModelChangedNotifications {
    [[NSNotificationCenter defaultCenter]
        removeObserver:self];
}
```

The first method starts listening for notifications from EventKit and the second stops listening. When there is a broadcast from EventKit, `fetchAllConferenceReminders` gets called and your `reminders` property gets updated.

You also need a way to notify external objects when the `reminders` array in `EventKitController` is updated. This is because external objects, such as the view that displays the fetched reminders in your app, need to know when there's a change to the list of reminders.

If you go back and check `fetchAllConferenceReminders`, you will notice that the method already sends a notification using the `RemindersModelChangedNotification` constant when the `reminders` array is updated.

But this constant is declared at the top of the implementation (.m) file, and so is not visible to external classes. You therefore need to add a declaration for this constant in **EventKitController.h**, above the `@interface` declaration, so that external classes can be aware of it:

```
extern NSString *const RemindersModelChangedNotification;
```

Also add method definitions for the three new methods you added to **EventKitController.h**:

```
-(void)fetchAllConferenceReminders;
-(void)startBroadcastingModelChangedNotifications;
-(void)stopBroadcastingModelChangedNotifications;
```

Open **ConferenceRemindersViewController.m** and add the following code to start and stop listening to broadcast events from EventKit (you can add the code anywhere, but for the sake of organizational neatness I suggest adding it below `initWithCoder:`):

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

    [_eventKitController
        startBroadcastingModelChangedNotifications];

    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(refreshView)
        name:RemindersModelChangedNotification
        object:_eventKitController];

    [_eventKitController fetchAllConferenceReminders];
}

- (void) dealloc {
    [_eventKitController
        stopBroadcastingModelChangedNotifications];

    [[NSNotificationCenter defaultCenter]
        removeObserver:self];
}
```

In `viewDidLoad` you also add an observer to the `EventKitController` object. This observer will be fired when the reminders array of the `EventKitController` object is updated. If updated, it will call `refreshView`. This method is already present in the starter kit code and all it does is reload the table view data.

Next, change the table view delegate methods. Change the existing code to match the following (only changed methods are given here):

```
- (NSInteger) tableView:(UITableView *)tableView
```

```
        numberOfRowsInSection:(NSInteger)section {
    return _eventKitController.reminders.count;
}

- (UITableViewCell *) tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

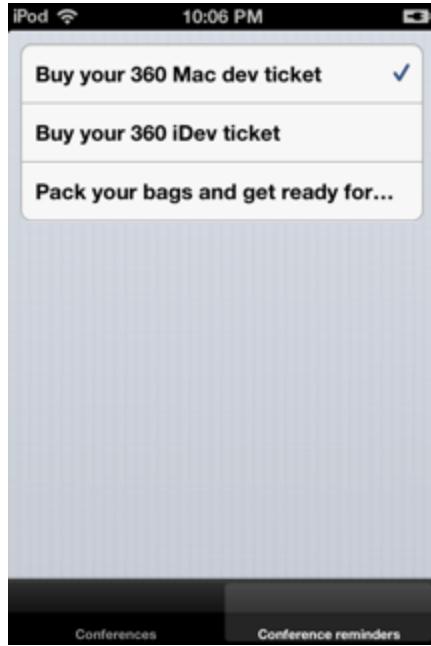
    UITableViewCell *cell =
        [tableView
            dequeueReusableCellWithIdentifier:
            @"ReminderCells"];

    EKReminder *reminder =
        (_EKReminder*)_eventKitController.
            reminders[indexPath.row];

    cell.textLabel.text = reminder.title;
    if (reminder.completed) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    } else {
        cell.accessoryType = UITableViewCellAccessoryNone;
    }
    return cell;
}
```

This is just standard code that you use to add rows to a table view.

Build and run the application. When you open the second tab, you should see all the reminders that were added to the “Conference reminders” calendar.



Marking a reminder complete

Next, it would be nice to allow the user to mark a reminder as complete/incomplete by tapping on that reminder in the second tab of the app. You can easily add this ability.

Open **EventKitController.m** and add the following method:

```
- (void) reminder:(EKReminder*) reminder
    setCompletionFlagTo:(BOOL) completionFlag {

    //1. Set the completed flag
    //Note: The completion date is automatically
    //      set to the current date

    reminder.completed = completionFlag;

    //2
    NSError *err;
    BOOL success = [self.eventStore
                    saveReminder:reminder
                    commit:YES error:&err];
    if (!success) {
        NSLog(@"There was an error editing the reminder");
    }
}
```

This method takes two arguments, a reminder and a Boolean flag that specifies whether or not to mark the reminder as complete. It sets the completed property of the reminder and saves the reminder to the calendar database. Yes, it's that simple! That's all it takes to set the status of a reminder.

Now add the method definition to **EventKitController.h**:

```
- (void)reminder:(EKReminder*)reminder  
setCompletionFlagTo:(BOOL)flag;
```

Open **ConferenceRemindersViewController.m** and replace `tableView:didSelectRowAtIndexPath:` with:

```
- (void)tableView: (UITableView *)tableView  
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {  
  
    UITableViewCell *cell =  
        [tableView cellForRowAtIndexPath:indexPath];  
  
    if (cell.accessoryType == UITableViewCellAccessoryCheckmark) {  
        cell.accessoryType = UITableViewCellAccessoryNone;  
        [_eventKitController reminder:  
            _eventKitController.reminders[indexPath.row]  
            setCompletionFlagTo:NO];  
    } else {  
        cell.accessoryType = UITableViewCellAccessoryCheckmark;  
        [_eventKitController reminder:  
            _eventKitController.reminders[indexPath.row]  
            setCompletionFlagTo:YES];  
    }  
}
```

Here the app first checks the current state of the reminder and then calls the new method in **EventKitController.m** to change it.

Build and run your app. Try tapping on a reminder on the second tab to see if you can toggle the status.



EventKit data isolation

iOS 6.0 introduces an entirely new “Privacy” section in the Settings app. Here users can control which apps have access to Location Services, Contacts, Calendar, Reminders, and Photos.

Now when your app wants access to events/reminders, the system will ask the user for permission. Only if the user grants permission will the app be able to access calendar data.

This image shows the privacy settings for reminders. Notice that ConferencePlanner appears as an app that has requested access to reminders.



At any point the user can revoke permissions for your app by using these settings.

You must be wondering, how does all this affect your code? These additions bring about a slight change in the way you should write EventKit code. It is important to note that the user is only asked for permission when you instantiate an `EKEventStore` and call `requestAccessToEntityType`.

This means you should only create an `EKEventStore` and call these methods when you actually need to. For example, in ConferencePlanner, you don't need to create an instance of `EKEventStore` and ask for permission at the start of the app. Instead, it should be created when the user opens the second tab or taps on any row on the detail page.

Data isolation in ConferencePlanner

So let's improve the code so that you only start accessing the events and reminders when you actually need to.

Open **EventKitController.h** and add the following constants right above the `@interface` declaration.

```
extern NSString *const EventsAccessGranted;
extern NSString *const RemindersAccessGranted;
```

Then define these variables in **EventKitController.m** by adding these lines at the top of the file, right below the other constant definition you added earlier:

```
NSString *const EventsAccessGranted = @"EventsAccessGranted";
NSString *const RemindersAccessGranted =
@"RemindersAccessGranted";
```

The `EventsAccessGranted` constant is a notification name that is used by the `EventKitController` class to notify other objects that the user has given permission to access event data. Similarly the `RemindersAccessGranted` constant is used to notify other objects that the user has given permission to access reminder data.

Next change the `init` method of `EventKitController` to the following:

```
- (id) init {
    self = [super init];
    if (self) {
        _eventStore = [[EKEventStore alloc]
                      init];
        [_eventStore requestAccessToEntityType:EKEEntityTypeEvent
                           completion:^(BOOL granted, NSError *error) {
            if (granted) {
                _eventAccess = YES;
                [[NSNotificationCenter defaultCenter]
                 postNotificationName:
                 EventsAccessGranted]
```

```

        object:self];
    } else {
        NSLog(@"Event access not granted: %@", error);
    }
};

[_eventStore
requestAccessToEntityType:EKEntityTypeReminder
completion:^(BOOL granted, NSError *error) {
    if (granted) {
        _reminderAccess = YES;
        [[NSNotificationCenter defaultCenter]
        postNotificationName:
        RemindersAccessGranted
        object:self];
    } else {
        NSLog(@"Reminder access not granted: %@", error);
    }
}];
_fetchQueue = dispatch_queue_create(
    "com.conferencePlannerForGeeks.fetchQueue",
    DISPATCH_QUEUE_SERIAL);
}

return self;
}
}

```

In the above method you have added changes to the completion handler of –`requestAccessToEntityType:completion:` method of `EKEventStore`. If the user grants access to Events a notification is sent. The name of this notification is “EventsAccessGranted”. A similar approach is used for reminders.

Now you are going to change the way you have used the `EventKitController` class throughout the project. Don’t worry, there aren’t many changes to make.

Open **ConferenceRemindersViewController.m** and remove the creation of the `EventKitController` object from `initWithCoder:`. You’ll be transferring the creation of the object to `viewDidLoad`. You do this because `initWithCoder:` is called when the app starts. If you kept the creation of the `EventKitController` there, then the user will be presented with an alert asking for permission (if required) as soon as the app starts. As discussed, you should only have to ask for permission when you truly need it.

You also need to add an observer for the `RemindersAccessGranted` notification sent by the `EventKitController` object. Since this notification is sent when the user has granted permission to access reminders data to your app, you should refresh the reminders list when it is sent. Replace the current implementation of `viewDidLoad` with the following:

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

    _eventKitController = [[EventKitController alloc] init];

    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(remindersPermissionGranted)
        name:RemindersAccessGranted
        object:_eventKitController];
}
```

This removes some code to set up some notifications on event changes, and starts up the `EventKitController` instead.

Next add this method, which is called when `RemindersAccessGranted:` notification is sent.

```
- (void)remindersPermissionGranted {
    [_eventKitController
        startBroadcastingModelChangedNotifications];

    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(refreshView)
        name:RemindersModelChangedNotification
        object:_eventKitController];

    [_eventKitController fetchAllConferenceReminders];
}
```

This is the code that used to be in `viewDidLoad`. Now that you know the `EventKitController` is initialized, it's safe to start listening for the broadcast model change notifications and refresh the reminders table view.

Switch to **ConferenceDetailViewController.m**. First comment out the line that creates the `EventKitController` in `initWithCoder`:

```
//_eventKitController = [[EventKitController alloc] init];
```

Then create two methods named `performEventOperations` and `performReminderOperations`.

```
- (void) performEventOperations {
    NSIndexPath *indexPath = [self.tableView
        indexPathForSelectedRow];
    if (indexPath.row == 0) {
```

```
//add code to add an event

[_eventKitController
    addEventWithName:self.conference.name
    startTime:self.conference.startDate
    endTime:self.conference.endDate];

} else if(indexPath.row == 1) {
    //add code to add a recurring event

[_eventKitController
    addRecurringEventWithName:self.conference.name
    startTime:self.conference.startDate
    endTime:self.conference.endDate];

} else if(indexPath.row == 2) {
    //add code to delete an event
    [_eventKitController
        deleteEventWithName:self.conference.name
        startTime:self.conference.startDate
        endTime:self.conference.endDate];
}

[self.tableView deselectRowAtIndexPath:indexPath
    animated:YES];
}

- (void) performReminderOperations {
    NSIndexPath *indexPath = [self.tableView
        indexPathForSelectedRow];
    if (indexPath.row == 0) {
        //code to add a reminder to buy a ticket
        [_eventKitController
            addReminderWithTitle:
                [NSString stringWithFormat:@"Buy your %@ ticket",
                 self.conference.name]
            dueTime:[NSDate date]];
    } else if(indexPath.row == 1) {
        //code to add a reminder to pack your bags
        //1
        NSCalendar *calendar = [NSCalendar currentCalendar];
        //2
        NSDateComponents *oneDayComponents =
```

```
[ [NSDateComponents alloc] init];

//3
oneDayComponents.day = -1;

//4
NSDate *dayBeforeTheEvent = [calendar
    dateByAddingComponents:oneDayComponents
    toDate:self.conference.startDate
    options:0];

//5
[_eventKitController
    addReminderWithTitle:
    [NSString
        stringWithFormat:
        @"Pack your bags and get ready for %@", self.conference.name]
    dueTime:dayBeforeTheEvent];
}

[self.tableView deselectRowAtIndexPath:indexPath
    animated:YES];
}
```

This is most of the code that used to be in `tableView:didSelectRowAtIndexPath`. Here you've pulled it out to separate methods so you can delay calling these methods until after you initialize the `EventKitController`, as you'll see next.

So now replace `tableView:didSelectRowAtIndexPath:` with the following:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    if (_eventKitController == nil) {
        _eventKitController = [[EventKitController alloc] init];

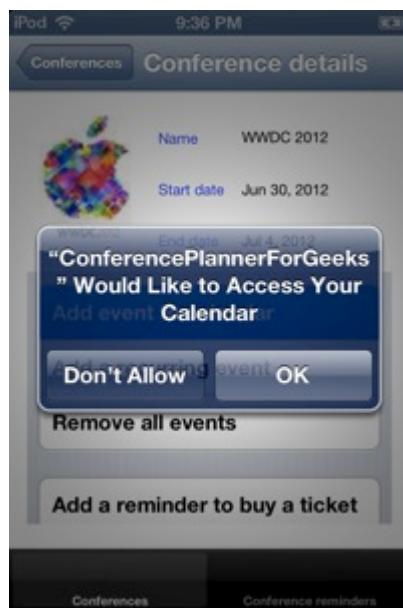
        [[NSNotificationCenter defaultCenter]
            addObserver:self
            selector:@selector(performReminderOperations)
            name:RemindersAccessGranted
            object:_eventKitController];

        [[NSNotificationCenter defaultCenter]
            addObserver:self
            selector:@selector(performEventOperations)
            name:EventsAccessGranted
            object:_eventKitController];
    }
}
```

```
    NSLog(@"w00t");
} else {
    if (indexPath.section == 0) {
        [self performEventOperations];
    } else {
        [self performReminderOperations];
    }
}
```

Now the method initializes the `EventKitController` if it hasn't already before performing the desired operations. This way, we've reached our goal of not asking the user for permission until it is actually required.

Compile and run the app. To test the new changes, make sure to reset the privacy settings. To do this open the Settings app, go to **General->Reset** and select "Reset location and privacy." Then go to a conference detail view and tap a row. The prompt will now display at the logical point!



Where to Go From Here?

That's a wrap folks! You've accomplished a lot in this chapter:

- You gained hands-on experience with EventKit, from the ground up.
- You learned how to use your app to create basic events and reminders.
- You learned how to read, update, and delete reminders and events.
- You learned how to work with the new privacy features pertaining to both.

Like all other frameworks on iOS, EventKit gives you, the developer, the flexibility to make your apps truly amazing and useful. I hope you plan to use EventKit in your next apps, and make some of your most elusive dreams come true!

Chapter 20: What's New with Cocoa Touch

By Felipe Laso Marsetti

Cocoa Touch is the heart and soul of iOS. Like Cocoa on OS X, it's the *engine* that drives your apps. Year in and year out, Apple makes vast improvements to Cocoa Touch and, more specifically, to UIKit (the subset of Cocoa Touch that deals with user interface elements).

By now, you've heard of the major new features in UIKit, which we've covered in previous chapters:

- **Auto Layout**, for fine-tuning the behavior of your interfaces (programmatically or in Interface Builder) without having to use the good old springs and struts. You learned about this in Chapters 3-4, "Beginning and Intermediate Auto Layout."
- **Collection Views**, which let you display items in a grid (finally!) or any other layout you desire. You learned about this in Chapters 5-6, "Beginning and Intermediate UICollectionView."
- **Activity View Controllers**, which allow the user to perform context-sensitive operations within your apps like copying or sharing data. You learned about this in Chapter 11, "Beginning Social Framework."
- **Attributed Strings**, which allow rich formatting and styling of strings, along with the UI components that support them. You learned about this in Chapter 15, "Attributed Strings."
- **State Preservation and Restoration**, for saving the state of your app's user interface so users can come right back to where they left off even if your app is closed by the user or iOS. You learned about this in Chapter 16, "State Preservation and Restoration in iOS 6."

So as you can see, there have been a ton of changes in Cocoa Touch, and you've already learned the most important topics.

But wait – there's more! Believe it or not, there's still plenty left to cover – a collection of improvements to UIKit that are seemingly minor compared to the big updates, but no less game-changing when it comes to your day-to-day coding life.

Here's just a sampling of what you will learn and practice in this chapter:

- Pull-to-refresh functionality inside table view controllers.
- New ways of reusing table view cells.
- New mechanics for orientation support within your view controllers.
- New `UIImage` methods that support determining the scale factor of an image.
- How to tint your app's status bar.
- New features in the page view controller.

Being aware of these new features will keep your skills up-to-date and make your life a lot easier – so let's take a look!

Note: This chapter is different than the other chapters so far, in that it covers a hodge-podge of new features that didn't warrant entire chapters. In addition, it's heavy on review – it covers some material in other chapters in this book, as well as material that advanced iOS developers may be well familiar with such as creating a Storyboard-based user interface.

So our recommendation is the following – if you are an advanced iOS developer, you might want to just skim this chapter to get a high level introduction to the new iOS 6 features not covered already, but not actually do the tutorial since it's heavy on review.

But if you are a beginner or intermediate iOS developer, you might enjoy following along with the tutorial since it will reinforce a lot of concepts for you and show you how to put everything together into a fun app. Enjoy!

Introducing RecipesKit

If you're anything like me, then you probably love food – particularly the eating part. In this chapter, you're going to create a simple cookbook app called RecipesKit to try out the new iOS 6 functionality listed above.

In the resources for this chapter, you will find a starter project called **RecipesKit Starter**. Open it in Xcode and run the project. You will see something like the following:



Right now the project is pretty bare bones. It lists some recipe names, but tapping them doesn't do anything yet. You'll be implementing the rest of the app in this chapter!

Before we begin, let's take a look through the sample project and make sure you understand what's there so far (which isn't much), and how it works.

First, open **MainStoryboard.storyboard** and have a look. The initial controller is a navigation controller that has a table view controller as the root controller.

There are two reasons why the root controller is a table view controller instead of a regular view controller: one, so it can be used with `NSFetchedResultsController` for near automatic coordination with Core Data; and two, because this is the only way to use the new pull-to-refresh control that's part of iOS 6.

Note: That's right, you cannot use pull-to-refresh with regular instances of `UITableView`, at least not in this version of iOS. That's because the refresh control is a property of `UITableViewController`, as you'll see later on in this chapter.

Look to the right of the table view controller, and you'll see an empty view controller. Later in this chapter you will add some controls to this view controller to make it a recipe detail screen, to show the recipe's picture, ingredients, and so on.

Now switch over to **AppDelegate.h**. This is the standard code included in the application delegate for Core Data-enabled projects.

Open **AppDelegate.m** and you'll see more boilerplate code for the Core Data stack in addition to some utility methods.

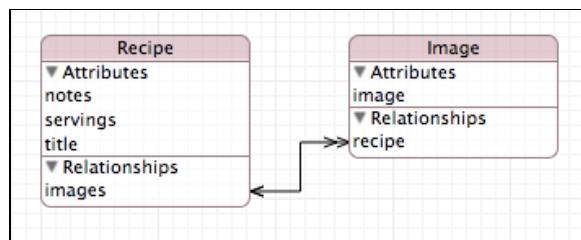
Time to check out the controllers. Open **RecipeListViewController.h** and have a look. It's a subclass of `UITableViewController` and implements the `NSFetchedResultsControllerDelegate` protocol. For properties, there's a managed object context and a fetched results controller.

Both are fairly self-explanatory. If not, open **RecipeListViewController.m** to find several methods and properties to set up the fetched results controller, and necessary stuff to be able to add and delete recipe objects to/from your projects.

RecipeDetailViewController.h/m is even simpler, with just recipe and managed object context properties and a custom setter for the recipe property.

Let's have a look at **RecipesKit.xcdatamodeld** to see the entities in Core Data. There's a recipe entity with some attributes to hold the recipe info and a one-to-many relationship to Image. The relationship is there because a recipe can include many Images of the tasty dish you will make.

The Image entity's image attribute is of type Transformable, so it can be saved as `NSData` in Core Data but retrieved as a `UIImage`.



Finally, you will find `NSManagedObject` subclasses for the Image and Recipe entities in the Core Data model. Notice how the Image property has the code for the Transformable image attribute.

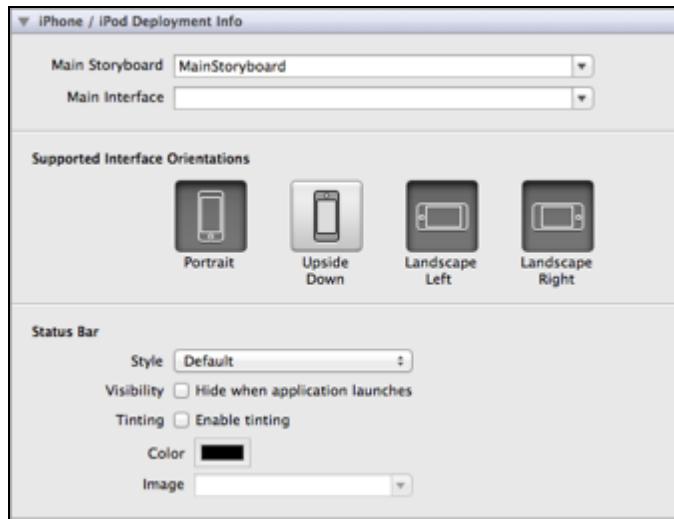
If you don't understand something, you might want to take the time to go back and look at the code and the structure of the storyboard, Core Data model, and classes. If you are familiar with Core Data, then you should have no problem following along.

Note: If you aren't familiar with Core Data, feel free to check out the Core Data tutorial series on raywenderlich.com. However, this is not required – you can get through this chapter without a great understanding of Core Data because the focus will be on the new iOS 6 Cocoa Touch features not already covered, not Core Data.

Status bar tint

Once you feel comfortable with how things work, take a look at the project settings. To do this, select the RecipesKit project root in the project navigator and go to the

Summary tab in the project's detail view. You will see the Bundle Identifier, version and build numbers, and then the following:



First of all, note that although iOS 6 has a new way of handling your app's supported interface orientations which you'll learn about later, the project summary does not show anything different that might suggest this. You'll handle the new iOS 6 orientation logic through code a little later on.

After the interface orientation section, you'll something not found in previous versions of Xcode or iOS, a Status Bar section inside the project settings.

In iOS 6, when using a navigation controller as the root view controller of your application, the status bar will automatically use the same tint color as the navigation bar. This is super-cool, because it means you no longer have to resort to a workaround to get a tint color if you don't want to stick with the default gray, black or 50% opaque black status bar styles.

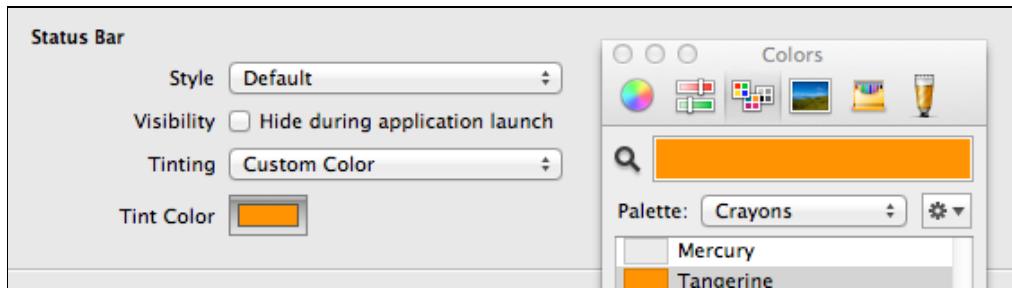
If the status bar automatically tints itself with the navigation bar's color, then why does the project have settings for it? Well, to see why this is, re-launch the project without enabling anything or making any changes.

Did you see it? As your app launches, the status bar wears the default black color. Only when the app finishes launching does the status bar's tint color change to match the blue of the navigation bar.

To smooth this transition, you can use the project settings to set the status bar's tint color to match the navigation bar right at startup, or to any color of your choosing (or even to a tint image).

Alternatively, you can hide the status bar by checking the "Hide during application launch" checkbox. This will hide the status bar for the entire app session – it's the same as if you had set that option in info.plist.

To try this out, change the Tinting dropdown to Custom Color, set the Tint Color to Crayons\Tangerine (the same color that the navigation bar is tinted).



Run your project, and look at how the status bar starts with the right color orange from the beginning, for a smoother transition!



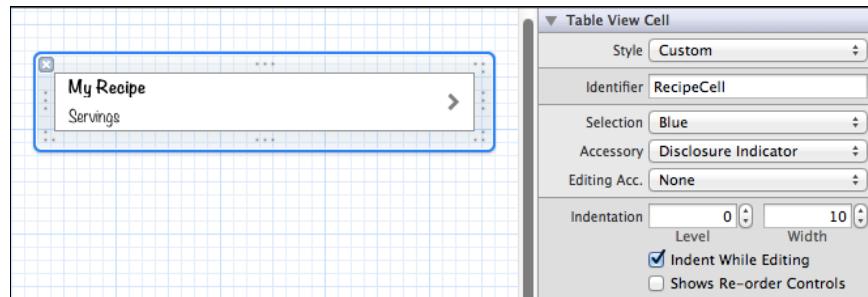
Reusable table cells improvements

Next, you'll try out some new methods in iOS 6 that make it easier to reuse table view cells, whether they are contained in NIBs or created programmatically in UITableViewCell subclasses. You'll start by trying out the new way to reuse table view cells created in NIBs.

Go to **File\New\New File...** and from the iOS\User Interface template list select **View** and click Next. Use iPhone for Device Family and click Next once again, name the nib **RecipeCell** and save it in your project directory.

Open the nib, delete the view, and drag in a table view cell. Select the cell, open the Attributes inspector and change the Style to Custom, set RecipeCell as the Identifier and give it a Disclosure Indicator for Accessory.

Drag two labels onto the cell, one for the recipe name and one for the number of servings. Give them some nice fonts and position them as you like. Afterwards, your layout should look something like this (with varying label style and position):



It's time to create a custom `UITableViewCell` subclass for this cell. Go to **File\New\New File...** and create an Objective-C Class from the available templates. Name it **RecipeCell**, make sure it's a subclass of `UITableViewCell` and save it in your project directory.

With **RecipeCell.h** open, replace the current contents with the following to define the identifier string for the cell, as well as outlets for the labels the cell will contain:

```
#define RecipeCellReuseIdentifier @"RecipeCell"

@interface RecipeCell : UITableViewCell

@property (weak, nonatomic) IBOutlet UILabel *titleLabel;
@property (weak, nonatomic) IBOutlet UILabel *subtitleLabel;

@end
```

So far, the code for the `RecipeCell` is pretty straightforward. There's a defined string to make it easy to fetch the identifier for the cell, and a property for each of the labels in the cell's nib.

With the property declarations in place, you need to connect the outlets to the corresponding labels in the nib. Open **RecipeCell.xib**, select the cell, open the Identity inspector, and set the class to `RecipeCell`. After this, right-click on the cell (it may be easier using the Document Outline) and connect the `titleLabel` outlet to the cell's top label and the `subtitleLabel` to the cell's bottom label.

The custom cell is all done, so it's time to put it to use in **RecipeListViewController.m**. Open that file and add the import declaration for `RecipeCell` at the top of the file:

```
#import "RecipeCell.h"
```

Then add this code at the end of viewDidLoad:

```
UINib *recipeCellNib = [UINib nibWithNibName:@"RecipeCell"
    bundle:[NSBundle mainBundle]];
[self.tableView registerNib:recipeCellNib
    forCellReuseIdentifier:RecipeCellReuseIdentifier];
```

This loads the nib for the recipe cell and then calls a new method in iOS 6 called registerNib:forCellReuseIdentifier:. This method allows you to register a table view cell contained in a nib for automatic cell reuse.

Now go to tableView:cellForRowAtIndexPath: and change the code that loads a cell (the first line) to the following:

```
UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:RecipeCellReuseIdentifier
    forIndexPath:indexPath];
```

Prior to iOS 6, you would have to create a table view cell here using the UINib if a reusable cell wasn't already available. Now since you called registerNib:forCellReuseIdentifier in viewDidLoad, you no longer have to do anything except call dequeueReusableCellWithIdentifier here. The OS will either use a reusable cell, or create a new one from the UINib for you if necessary!

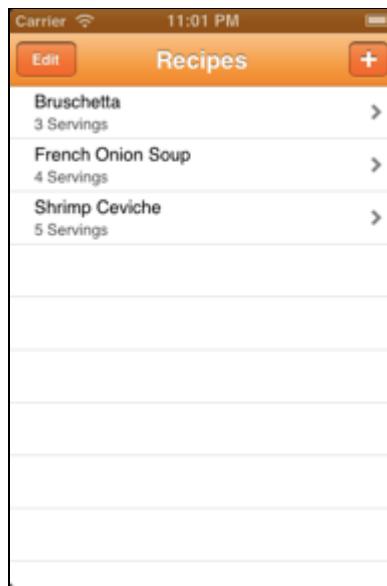
Before you see the results (and in order to avoid crashing your app), a small update is needed inside configureCell:atIndexPath:. Replace the method with the following:

```
- (void)configureCell:(UITableViewCell *)cell
    forIndexPath:(NSIndexPath *)indexPath
{
    // Get the Recipe for the corresponding row index
    Recipe *recipe = [self.fetchedResultsController
        objectAtIndex:indexPath];
    RecipeCell *recipeCell = (RecipeCell *)cell;

    // Setup the Detail and Text labels with the recipe's values
    recipeCell.titleLabel.text = recipe.title;
    recipeCell.subtitleLabel.text = [recipe servingsString];
}
```

This method gets the recipe corresponding to the received index path, and casts the cell it receives to an instance of RecipeCell. Last but not least, the cell's labels are set to the recipe's title and number of servings.

Run the project and look at the results:



Now that you understand reusing a table view cell from a NIB, let's see how it would work if you wanted to create your table view cell via code.

Create a new `UITableViewCell` subclass named **RecipeCodeCell** and replace the contents of **RecipeCodeCell.h** with the following:

```
#define RecipeCodeCellReuseIdentifier    @"RecipeCodeCell"
#define RecipeCodeCellSegue           @"RecipeCodeCellSegue"

@interface RecipeCodeCell : UITableViewCell

@property (strong, nonatomic) UILabel *nameLabel;
@property (strong, nonatomic) UILabel *servingsLabel;

@end
```

There are two `#define` statements here, one for the segue that's already in place in the storyboard, and one for the cell's reuse identifier. Inside the class interface you can see the same two labels as in the `RecipeCell` class, but with some different names and without the `IBOutlet`.

Since the cell is not going to be used in a nib or storyboard, there's no need to make the `IBOutlet`s. Open **RecipeCodeCell.m** and add the following method:

```
- (NSString *)reuseIdentifier
{
    return RecipeCodeCellReuseIdentifier;
}
```

This sets the reuse identifier for the cell to the string declared in the header – nothing complicated. The biggest portion of code is the custom `initWithStyle:reuseIdentifier:` method, which will replace the existing boilerplate code:

```
- (id)initWithStyle:(UITableViewCellStyle)style  
reuseIdentifier:(NSString *)reuseIdentifier  
{  
    if (self = [super initWithStyle:style  
                      reuseIdentifier:reuseIdentifier])  
    {  
        self.accessoryType =  
            UITableViewAccessoryDisclosureIndicator;  
  
        self.servingsLabel = [[UILabel alloc] init];  
        self.servingsLabel.font =  
            [UIFont fontWithName:@"Noteworthy-Bold" size:13];  
        self.nameLabel = [[UILabel alloc] init];  
        self.nameLabel.font =  
            [UIFont fontWithName:@"Noteworthy-Bold" size:12];  
  
        self.servingsLabel.frame = CGRectMake(10, 21, 270, 21);  
        self.nameLabel.frame = CGRectMake(10, 2, 270, 21);  
  
        [self.contentView addSubview:self.servingsLabel];  
        [self.contentView addSubview:self.nameLabel];  
    }  
  
    return self;  
}
```

The code sets the cell's accessory type, creates two labels and adds them as subviews. It's doing pretty much the same thing as the cell inside the nib, except that this time, the setup is done via code.

Your cell is ready, so put it to use in **RecipeListViewController.m**! Change the `#import` statement for `RecipeCell` to `RecipeCodeCell`:

```
#import "RecipeCodeCell.h"
```

Find `configureCell:atIndexPath:` and replace it with the following code:

```
- (void)configureCell:(UITableViewCell *)cell  
atIndexPath:(NSIndexPath *)indexPath  
{  
    // Get the Recipe for the corresponding row index
```

```
Recipe *recipe = [self.fetchedResultsController  
    objectAtIndex:indexPath];  
  
//RecipeCell *recipeCell = (RecipeCell *)cell;  
RecipeCodeCell *recipeCell = (RecipeCodeCell *)cell;  
  
// Setup the Detail and Text labels with the recipe's values  
recipeCell.servingsLabel.text = [recipe servingsString];  
recipeCell.nameLabel.text = recipe.title;  
}
```

The above is the same code as if you were using the nib cell, except that the cell received as a parameter is cast to `RecipeCodeCell` instead of `RecipeCell` and the labels use the new names that you just declared in `RecipeCodeCell.h`.

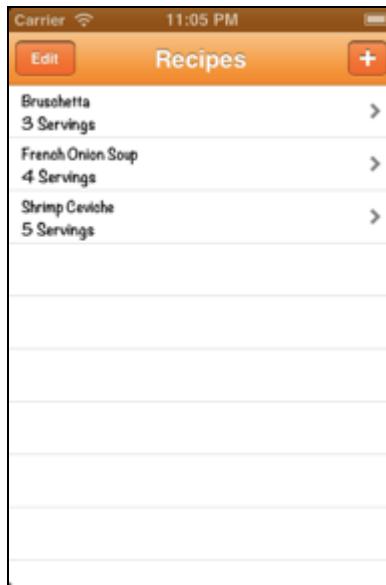
Move over to `viewDidLoad` and replace the previous lines for registering the cell's nib for reuse with the following line (the code you need to remove is commented out in the sample below):

```
//UINib *recipeCellNib = [UINib nibWithNibName:@"RecipeCell"  
    bundle:[NSBundle mainBundle]];  
//[self.tableView registerNib:recipeCellNib  
//    forCellReuseIdentifier:RecipeCellReuseIdentifier];  
[self.tableView registerClass:[RecipeCodeCell class]  
    forCellReuseIdentifier:RecipeCodeCellReuseIdentifier];
```

Finally, make a small change in `tableView:cellForRowAtIndexPath:` – the new version should look as follows:

```
- (UITableViewCell *)tableView:(UITableView *)tableView  
cellForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    // Create or dequeue a table cell and configure it for the  
    given index path  
    //UITableViewCell *cell = [tableView  
    dequeueReusableCellWithIdentifier:RecipeCellReuseIdentifier  
    forIndexPath:indexPath];  
    UITableViewCell *cell = [tableView  
    dequeueReusableCellWithIdentifier:RecipeCodeCellReuseIdentifier  
    forIndexPath:indexPath];  
  
    [self configureCell:cell atIndexPath:indexPath];  
  
    return cell;  
}
```

Instead of calling the `dequeueReusableCell` method with the nib's reuse identifier, you now call it with the programmatic cell's identifier. Run the program once more:



The results are similar, but this time the cells are being created using your programmatic cell (the difference is irrelevant to the user; it's all under the hood).

Pat yourself on the back – now you know how to reuse table view cells whether they are contained in NIBs or created programmatically. As for cells created via Storyboards, you should already know how to do that from iOS 5 by Tutorials. ☺

Displaying the details

Before wrapping up the list of recipes, how about adding support for displaying the recipe's detail view when a cell is tapped? The starter project includes a manual segue with the Push style that's been given a reuse identifier (you have a `#define` for it inside `RecipeCodeCell.h`).

Add the implementation for `tableView:didSelectRowAtIndexPath:` to **RecipeListViewController.m**:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [self performSegueWithIdentifier:RecipeCodeCellSegue
        sender:self];
}
```

This performs the segue previously set up in the storyboard.

If you build and run the project and tap on a cell, you'll see that the detail view now appears. But you'll also notice that no recipe details are actually being displayed! Well, right now it would be impossible for the detail view to show anything, because it isn't receiving a Recipe object for which to show the details.

In order to address this, you can use `prepareForSegue:sender:` to set up any data necessary before performing the segue. Find the existing implementation for the method and replace it with the following code:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender
{
    if ([[segue identifier]
        isEqualToString:RecipeCodeSegue]])
    {
        NSIndexPath *indexPath = [self.tableView
            indexPathForSelectedRow];
        Recipe *recipe = [[self fetchedResultsController]
            objectAtIndex:indexPath];

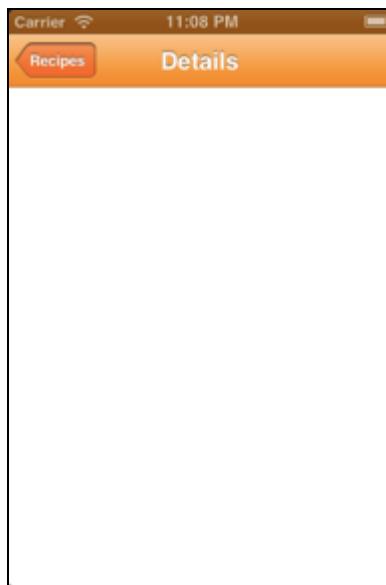
        RecipeDetailViewController *recipeDetailViewController =
            segue.destinationViewController;

        recipeDetailViewController.recipe = recipe;
        recipeDetailViewController.managedObjectContext =
            self.fetchedResultsController.managedObjectContext;
    }
}
```

If the identifier matches the segue between the recipe list and detail controllers, then the index path for the selected row is acquired and used to retrieve the appropriate recipe.

With the Recipe object fetched and ready, the segue's destination view controller is cast to `RecipeDetailViewController`, and the recipe data is passed into it, in addition to the managed object context (so changes to the recipe's detail can be saved and persisted).

Build and run your project again and select a row in the table:



Hmm... still empty. Oops! That's because the detail view is not set up to display any information at the moment. Sorry about that. ☺ But we'll get to that next!

The dinner is in the details

Open your storyboard and drag a scroll view onto the `RecipeDetailViewController` scene. Make it as large as the view itself so that the content can be resized when the keyboard is shown – something you'll learn about later in the chapter.

Now put the following elements, with their corresponding attributes, inside the scroll view:

Text Field:

- Give it a Bold System Font with 14 points for size.
- Change the placeholder text to "Recipe Title".
- Change the Border Style to no border.
- Position it at 20 points in X and 7 points in Y.
- Make it 280 points wide by 25 points high.

Button:

- Change the button type to Custom.
- Change the title to "Servings".
- Give it a System Font with 12 points for size.
- Set Alignment (under Control) to left-aligned.
- Position it at 20 points in X and 50 points in Y.

- Make the button 280 points wide by 23 points high.

Text View:

- Give it a System Font with 12 points for size.
- Position it at 0 points in X and 246 points in Y.
- Make the text view 320 points wide by 170 points high.

Now drag a bar button item from the Object Library onto the right side of the navigation bar. In the Attributes inspector change the Identifier to Action.

This next part is a bit tricky, so pay close attention. Drag a container view from the Object Library onto your recipe detail view. Give it a width of 320 points and a height of 165 points, and position it at 0 points in X and 80 points in Y.

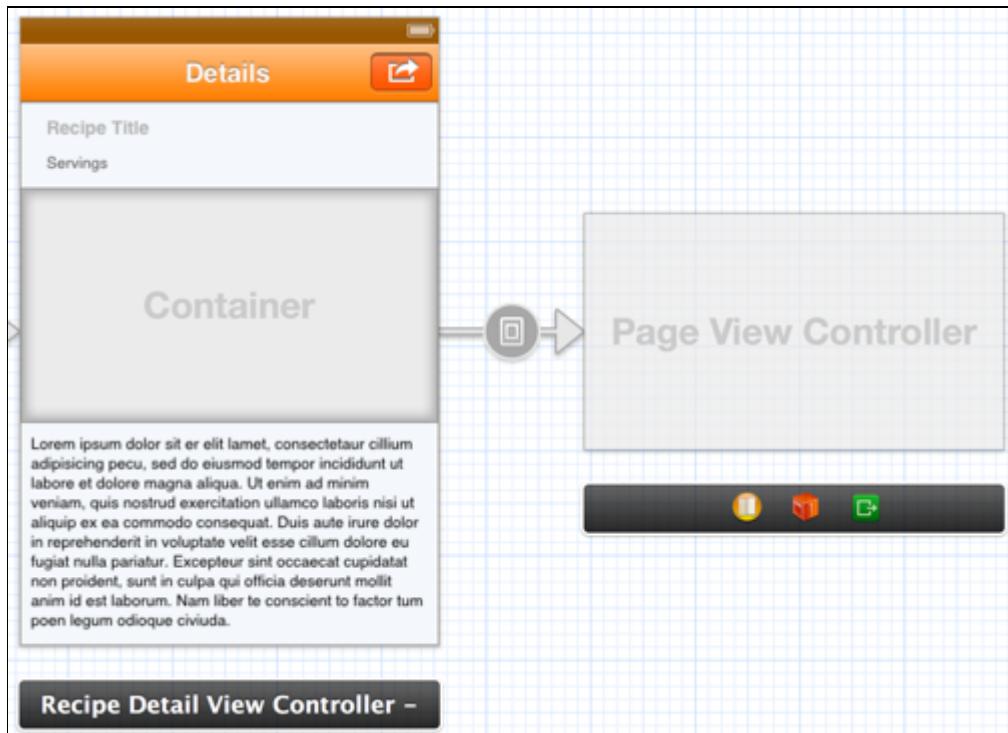
Notice how, by dragging out a container view, you automatically get an embed segue with a view controller.

Note: Container views and embed segues are new in iOS 6. We'll touch on them only briefly here, so to learn more check out Chapter 21, "What's New With Storyboards."

This container view will display a page view controller. So the default segue embedded into the container view is not going to be useful. Select the embedded view controller for the container view and delete it. In its place, drag a page view controller from the Object Library.

Right-click the container view and from the popup window that is shown, drag from `viewDidLoad` to the new page view controller (or, simply Control-drag from the container view to the page view controller). Select **Embed** (the only option available) from the popup menu.

You should now have something similar to the following UI layout for the Recipe Detail View Controller:



Notice how the page view controller is automatically resized to match the container view's size. Pretty convenient, eh?

You need to tweak one final setting for the page view controller, so select it in the storyboard and open the Attributes inspector. Make sure Navigation is set to Horizontal and that the Transition Style is set to Scroll.

Another new feature of iOS 6 is support for a scrollable page view controller. In iOS 5, when `UIPageViewController` was introduced, the only transition style was the page flip. Now in iOS 6, you can choose a scroll transition style that lets users swipe through view controllers.

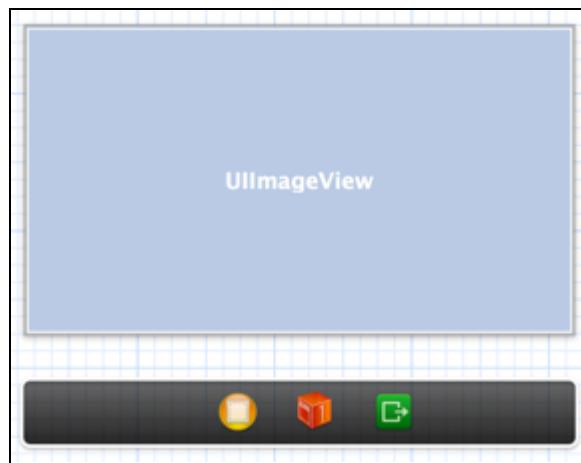
It even comes with a page indicator and a lot of functionality that you would have had to build yourself prior to iOS 6. For RecipesKit, you'll use the scroll style for the page view controller to show the pictures you've added for your recipes.

Now you need to add a view controller for the individual photos to be shown inside the contained page view controller. A `UIPageViewController` works with child view controllers, thanks to the view controller containment API introduced in iOS 5. When it's time to show a recipe detail, you will create view controllers for each image the recipe has in its images entity.

Drag out a view controller, open the Attributes inspector and set the Size to Freeform and the Status Bar to None.

Select the view inside the view controller, navigate to the Size inspector and set the width to 320 points and height to 180 points.

Next, add an image view as a subview of this view controller, make it the same size as the parent view, and in the Attributes inspector, change the Mode to Aspect Fit. Here is the finished view controller:



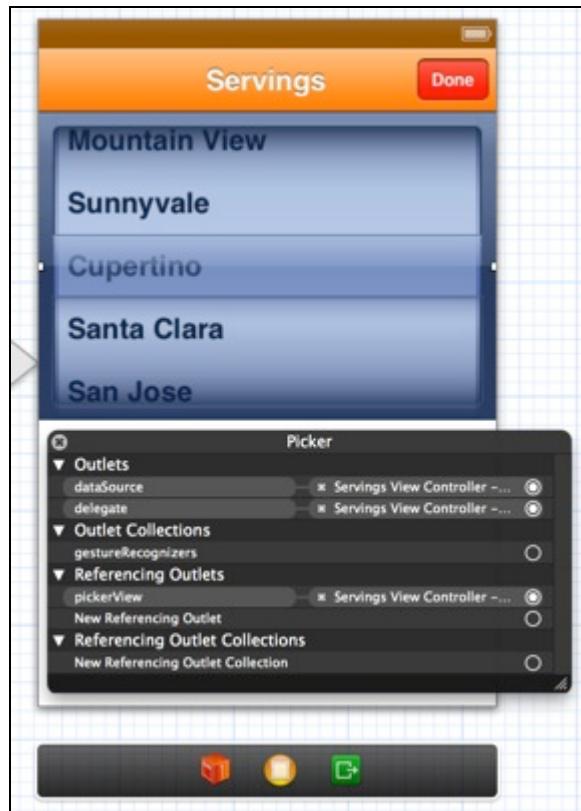
Let's move on. You need a way for the user to change the number of servings for the current recipe. Remember the button you added to the Recipe Detail View? This button, when tapped, should present a view controller with a servings picker view. However, there is no such view controller yet.

Drag a view controller from the Object Library, and then add a navigation bar at the top. Double-click the center of the navigation bar and enter "Servings" for the title. Then set the tint color of the navigation bar to Tangerine.

Now drag a bar button item from the Object Library and place it on the right side of the navigation bar. In the Attributes inspector, change the Identifier for the button to Done.

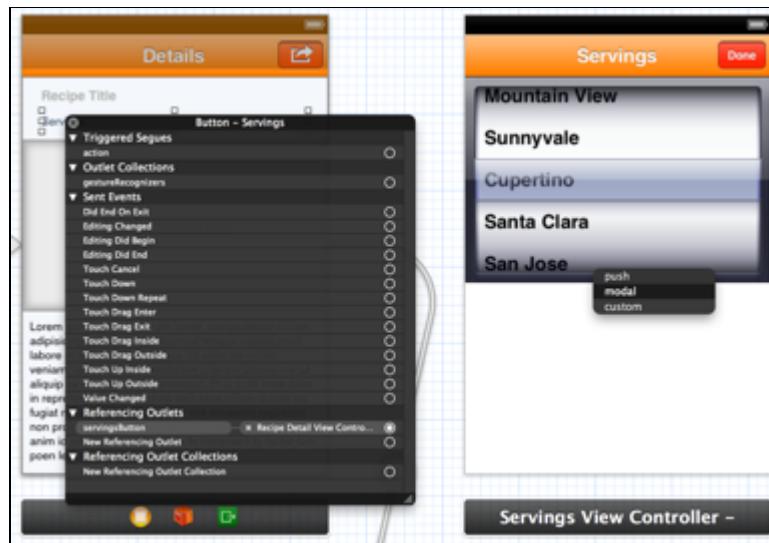
Next, drag out a picker view and place it right below the navigation bar.

A picker view has a data source and a delegate, so right-click on the picker and connect both of those outlets to the view controller that it's in.

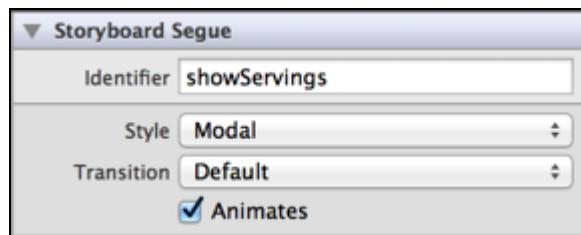


There is currently no way for you to get to the Servings Picker View Controller. So go over to the Recipe Detail View Controller and control-drag from the button to the Servings Picker View Controller.

When you let go, you will have a few options for the presentation style – select Modal.



Finally, select the segue between the Recipe Details View Controller and the Servings Picker View Controller, and in the Attributes inspector set "showServings" as the Identifier.



Right now, if you run the app and tap the Servings button, the app will crash because the picker's delegate and data source methods have not yet been implemented. Also, there are no outlets in the Recipe Details View Controller to enable the desired functionality.

It's time to add this in code and learn about some brand new stuff in iOS 6!

But before getting down to the code, there are two important changes in iOS 6 that you need to know about:

- Deprecation of `viewDidUnload` and `viewWillUnload`
- A new interface orientation API via `supportedInterfaceOrientations` and `preferredInterfaceOrientationForPresentation`

Let's take each of these in turn.

viewDidUnload Deprecated

Prior to iOS 6 when your app would get a memory warning, the OS would cycle through your view controllers and look for any that have a main view that is not currently visible. If it found any view controllers with non-visible views, it would unload the view for that view controller (effectively setting it to nil) and call `viewWillUnload`/`viewDidUnload` on your view controller when this happens. The next time the view controller became visible, the OS would automatically call `viewDidLoad`/`viewWillAppear` to reconstruct view views.

As of iOS 6, the OS no longer automatically unloads/reloads your views like this. This is because Apple has been improving things behind the scenes, and this no longer results in the memory savings that it used to. Furthermore, Apple has found that this process was a common cause of programming errors, and it would just be better to keep the view for a view controller always loaded unless a programmer *really* knows what they're doing.

So in iOS 6, `viewDidUnload` and `viewWillUnload` are deprecated. This isn't to say that your existing apps will crash or have trouble executing, it's just that these methods won't be called anymore.

Most of the time, you can get away with just deleting the code you used to have there and you'll be OK. But what if you really want to unload your view upon memory warnings to get the memory savings?

Well you can still do that, you just have to instead implement any of the following three to clean up your views and controllers:

- `viewDidDisappear`
- `viewWillDisappear`
- `didReceiveMemoryWarning`

The reason for this is that the methods above will yield the same results as the deprecated methods if you just change the way you use them. `viewDidDisappear` and `viewWillDisappear` can be used to cleanup any resources that can easily be reloaded in `viewWillAppear` and `viewDidAppear`. `didReceiveMemoryWarning` can replace `viewDidUnload` in order to perform clean up when your view is unloaded from memory due to insufficient memory.

There is one very important check to make before cleaning up any items in `didReceiveMemoryWarning`:

```
if ([self.view window] == nil)
{
    // ...
}
```

If the window of the view controller's view is `nil`, then that means the view is no longer in the hierarchy and is no longer being actively displayed. So, here it is safe to set the view to `nil`, as well as any that you want to clear in order to release memory. Just make sure you reload those items in `viewDidLoad` again, or your app will probably crash!

You'll try this out in a few minutes – but first let's discuss another change in iOS 6.

The new orientation methods

The other big change in UIKit relevant to this section has to do with orientation support.

Up until iOS 6, you set your application's supported orientations in `Info.plist` or the project summary. Then, for each individual controller, you overrode `shouldAutorotateToInterfaceOrientation:`, checked the received orientation and returned `YES` or `NO` depending on whether your controller supported the orientation the device was switching to.

In iOS 6, things become much simpler. Most of the time, you don't have to do anything, as view controllers are intended to be able to adapt to the size given to them. However, there are two methods you can override in your view controllers to customize the behavior of the app:

- `supportedInterfaceOrientations`
- `preferredInterfaceOrientationForPresentation`

`supportedInterfaceOrientations` returns an `NSUInteger` bitmask with the supported orientations of your view controller. What's really cool and awesome about this new method is that it's only called on the root view controller or the topmost controller that fills the window in your view hierarchy – rather than having to check with each view controller in the hierarchy like before.

If the bitmask returned by the controller matches the orientations supported by your application (inside `Info.plist` or the project summary), then the view controller rotates along with its child view controllers.

`preferredInterfaceOrientationForPresentation` simply lets you specify your controller's preferred orientation when being presented in full screen. If you don't want to override this method, then your view controller will have the same orientation as the status bar.

Out with the old, in with the new

It's time to put this new knowledge to use in your existing controllers, starting with memory clean up.

Let's see how you could clear out your view upon a memory warning in iOS 6 – just as used to happen automatically in iOS 5 and prior. Remember – this is an optional thing if the memory savings matter to your app.

Open `RecipeListViewController.m` and replace the existing `didReceiveMemoryWarning` with the following:

```
- (void) didReceiveMemoryWarning
{
    if ([self.view window] == nil)
    {
        self.view = nil;
    }

    [super didReceiveMemoryWarning];
}
```

As you saw before, the view's window is checked for `nil` before cleaning up any sub-views. Given that the recipe list's view is simply a table view, that's the only element that is set to `nil` and cleaned up inside `didReceiveMemoryWarning`.

Note: In the iOS Simulator, you can go to **Hardware\Simulate Memory Warning** to test `didReceiveMemoryWarning` and check if your views are loaded/unloaded properly from memory.

Now add the method for supported orientations:

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

Given that RecipesKit is an iPhone app, it will support all orientations except portrait upside down. One thing to double-check is that you use the new interface orientation masks. They are not the same as the previous orientation values used with `shouldAutorotateToInterfaceOrientation`. Here are all of the new masks:

```
UIInterfaceOrientationMaskPortrait
UIInterfaceOrientationMaskLandscapeLeft
UIInterfaceOrientationMaskLandscapeRight
UIInterfaceOrientationMaskPortraitUpsideDown
UIInterfaceOrientationMaskLandscape
UIInterfaceOrientationMaskAll
UIInterfaceOrientationMaskAllButUpsideDown
```

Note: If you ever have problems with the orientation of your view controllers, make sure that, if implementing `supportedInterfaceOrientations`, you are using the new masks.

Note that `shouldAutorotateToInterfaceOrientation` is now deprecated in iOS 6 and will no longer be called. You should use the new `supportedInterfaceOrientations` method instead.

There is one caveat to using the new orientation methods. While they are cleaner and better than the previous way of supporting different orientations, they will not work in anything lower than iOS 6.

To work around this, you would need to add a preprocessor directive similar to the following to check the iOS version and support orientation handling accordingly:

```
#if __IPHONE_OS_VERSION_MAX_ALLOWED >= __IPHONE_6_0
- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskAllButUpsideDown;
}

#else

- (BOOL)shouldAutorotateToInterfaceOrientation:
```

```
(UIInterfaceOrientation)interfaceOrientation  
{  
    return YES;  
}  
  
#endif
```

Run your application, change the device orientation to landscape and look at the results. If everything works fine, you should see something like this:



Fantastic work! Believe it or not, these new memory management and orientation changes are some of the most important in iOS 6. They aren't flashy changes that users will notice, but they will become standard moving forward.

The orientation handling methods are of particular importance. I'm sure you remember having to check every view controller in your hierarchy just because a view controller wasn't rotating to the correct orientation (despite indicating that it does). That's a headache of the past, and I'm sure you will enjoy the new orientation methods as much as I have.

View controller soup, with buttons

Now that you've learned the new ways of handling orientation changes and memory cleanup, let's continue with RecipesKit and add some functionality to the Recipe Details View Controller, and all of the controllers and views associated with it.

There are two view controller subclasses you'll need to add: one for the photos controller and one for the servings controller.

Add a new Objective-C class to your project, make sure it's a `UIViewController` subclass and name it **PhotosViewController**.

Repeat these steps to create the other view controller, but this time name it **ServingsViewController**.

Open **ServingsViewController.h** and add this line of code to the class interface:

```
@property (weak, nonatomic) IBOutlet UIPickerView *pickerView;
```

Then switch to **ServingsViewController.m** and replace the class extension at the top of the file with this new version, which implements the picker view's data source and delegate protocols:

```
@interface servingsViewController () <UIPickerViewDataSource,  
UIPickerViewDelegate>  
  
@end
```

Implementing a protocol in a class extension is perfectly legal, and it keeps your header file and class interface clean.

Next, add the data source methods for the picker view:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)  
pickerView  
{  
    return 1;  
}  
  
- (NSInteger)pickerView:(UIPickerView *)pickerView  
numberOfRowsInComponent:(NSInteger)component  
{  
    return 10;  
}
```

The picker will have only one column, called components. This column will have ten items so users can make a recipe with up to ten servings.

There is just one method you need to implement for the picker's delegate:

```
- (NSAttributedString *)pickerView:(UIPickerView *)pickerView  
attributedTitleForRow:(NSInteger)row  
forComponent:(NSInteger)component  
{  
    return [[NSAttributedString alloc] initWithString:[NSString  
        stringWithFormat:@"%d", row + 1]];  
}
```

pickerView:attributedTitleForRow: returns an NSAttributedString with the title you want for the given row of the picker. This is the title that will be shown in the picker, and since the servings go from 1-10 (unlike the picker row index, which goes from 0-9), the code adds one to the row to get the serving value for each row.

Last, but not least, are the methods for memory clean up and orientation support (remember to replace `didReceiveMemoryWarning`, since that method already exists in the code):

```
- (void) didReceiveMemoryWarning
{
    if ([self.view window] == nil)
    {
        _pickerView = nil;
        self.view = nil;
    }

    [super didReceiveMemoryWarning];
}

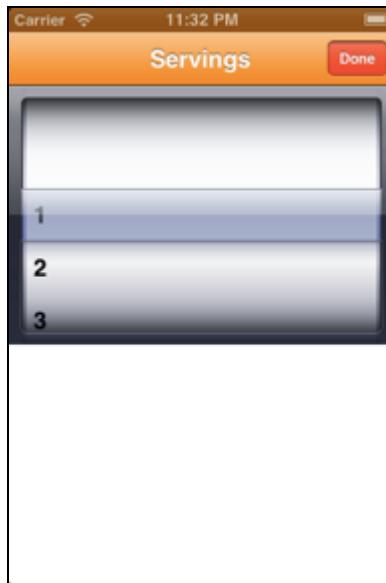
- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskAllButUpsideDown;
}
```

`didReceiveMemoryWarning` sets the picker view and the controller's view to `nil`. Those will get reloaded when the controller is needed, so this is perfectly OK. `supportedInterfaceOrientations` once again returns the bitmask that supports all orientations except for portrait upside down.

It's time to hook `ServingsViewController` up to the corresponding controller in your storyboard. Open **MainStoryboard.storyboard** and select the view controller with the picker view, then open the Identity inspector and set the class to `ServingsViewController`.

Right-click the view controller (you can use the bar below the view to do this) and drag from the `pickerView` outlet to the picker view on the storyboard.

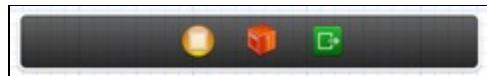
Run the app, go to the Details view, and tap the `Servings` button to look at your modal servings controller with its own picker view.



There is one problem – tapping the Done button does not dismiss the picker!

iOS 6 includes a new feature called unwind segues that is covered in detail in Chapter 21, “What’s New With Storyboards.” In short, unwind segues are like regular segues you use to push or present a new controller, except that they get called when dismissing or popping a controller – hence the name.

Note: Unwind segues can be set up and connected using the little black icon (the rightmost one) at the bottom of a view controller inside Interface Builder:



Unwind segues are called from the controller that you want to receive the notice about the dismissal. In this case, the Recipe Detail View Controller needs to declare an `IBAction` that will get called from the Servings Picker View Controller when the user taps Done.

To implement this, open `RecipeDetailsViewController.m` and add the following method:

```
- (IBAction)doneTapped:(UIStoryboardSegue *)segue
{
    ServingsViewController *servingsViewController =
        (ServingsViewController *)segue.sourceViewController;
    NSNumber *servings = @(([servingsViewController.pickerView
        selectedRowInComponent:0]) + 1);

    self.recipe.servings = servings;
```

```
AppDelegate *appDelegate = (AppDelegate *)[[UIApplication
    sharedApplication] delegate];
[appDelegate saveContext];
}
```

A `UIStoryboardSegue` has `destinationViewController` and `sourceViewController` properties to indicate what controllers the transition is going/coming from.

The `sourceViewController` is cast to an instance of `ServingsViewController` (since you are going from the Servings View Controller, it's the source) and the servings are retrieved from the picker view.

With the servings handy, the Detail View Controller's `Recipe` object is updated with the new value, and the app delegate's `saveContext` method is called in order to save the changes via Core Data.

You need to import two headers in order for the code you just added to work properly:

```
#import "AppDelegate.h"
#import "ServingsViewController.h"
```

Finally, you need to connect the unwind segue inside the storyboard. Switch to **MainStoryboard.storyboard**, navigate to the Servings View Controller, right-click the Done button and drag from the action outlet to the black icon at the bottom right of the view controller's bar. Then from the popup menu, select `doneTapped:`.



Your segue is done, and the Servings View Controller can now be properly dismissed with the changes immediately saved.

Run the project again, select a recipe, and tap the Servings button. Now when you tap the Done button to dismiss the view, things should work as expected.

Another view controller done! Now it's `PhotosViewController`'s turn.

Open **PhotosViewController.h** and add the following properties:

```
@property (strong, nonatomic) UIImage *image;
@property (assign, nonatomic) NSUInteger index;
```

`image` corresponds to the picture of the recipe that this particular controller will show. `index` is simply an `NSUInteger` that will help identify the index of the page view corresponding to each instance of `PhotosViewController`.

In `PhotosViewController.m`, add a `UIImageView` property to the class extension:

```
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
```

No one outside of `PhotosViewController` needs to know about the image view, so you keep it private by declaring it inside the class extension.

Next, add the following code, replacing existing methods as needed:

```
- (void)didReceiveMemoryWarning
{
    if ([self.view window] == nil)
    {
        _image = nil;
        _imageView = nil;
        self.view = nil;
    }

    [super didReceiveMemoryWarning];
}

- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskAllButUpsideDown;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    [self.imageView setImage:_image];

    self.view.backgroundColor = [UIColor clearColor];
}
```

`didReceiveMemoryWarning` cleans up the image, image view and controller view variables. This is nothing new or different from what you've been doing with other controllers.

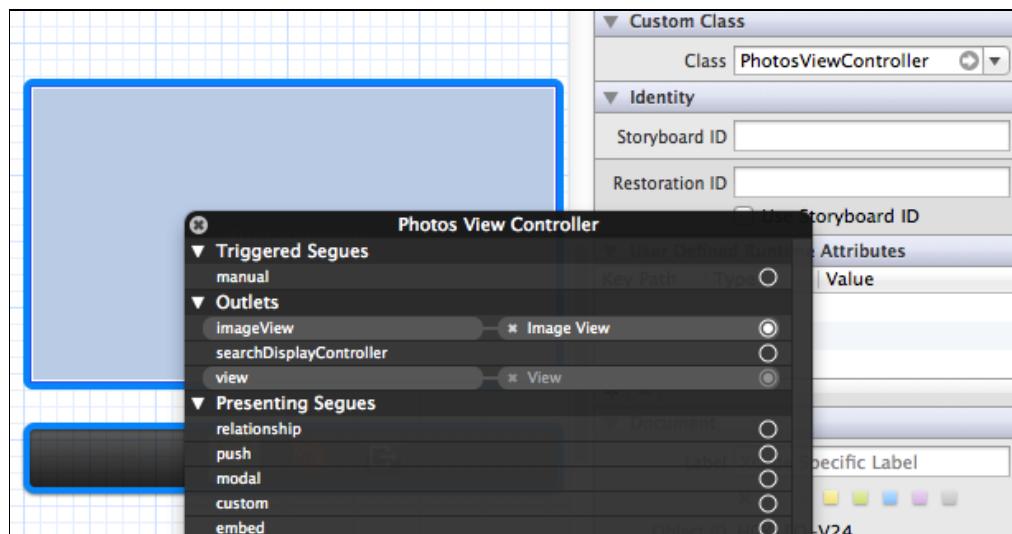
`supportedInterfaceOrientations` once again returns the bitmask to indicate that it can rotate to any orientation except portrait upside down.

`viewDidLoad` sets the image view's `image` property (the property will be set prior to showing the Photos View Controller) and makes the view's background color fully transparent.

And that's it for `PhotosViewController`!

It's time to connect the image view outlet inside the storyboard. Locate the Photos View Controller in Interface Builder and, with the controller selected, set the class to `PhotosViewController` inside the Identity inspector.

Right-click the image view and drag from its referencing outlet to the `viewController` in order to connect it to the `UIImageView` property.



Another view controller is done and ready for you to use!

Build and run the app to sure that everything compiles and works properly, but keep in mind there is no way to test the Photos View Controller yet – there's still some work to be done before images can be displayed.

Open `RecipeDetailsViewController.m`, and in the class extension add the following properties:

```
@property (strong, nonatomic) UIBarButtonItem *actionButton;
@property (strong, nonatomic) UIBarButtonItem *cameraButton;
@property (strong, nonatomic) UIBarButtonItem *doneButton;
@property (weak, nonatomic) IBOutlet UITextView *notesTextView;
@property (weak, nonatomic) IBOutlet UIScrollView *scrollView;
@property (weak, nonatomic) IBOutlet UIButton *servingsButton;
@property (weak, nonatomic) IBOutlet UITextField *titleTextField;
```

There are outlets for the title text field, the notes text view, the Servings button and the scroll view.

Now add the following method and method stubs:

```
- (void)actionTapped
{
}

- (void)localDoneTapped
{
    if (self.notesTextView.isFirstResponder)
    {
        [self.notesTextView resignFirstResponder];
    }
    else if (self.titleTextField.isFirstResponder)
    {
        [self.titleTextField resignFirstResponder];
    }

    self.recipe.title = self.titleTextField.text;
    self.recipe.notes = self.notesTextView.text;
}

- (void)cameraTapped
{
```

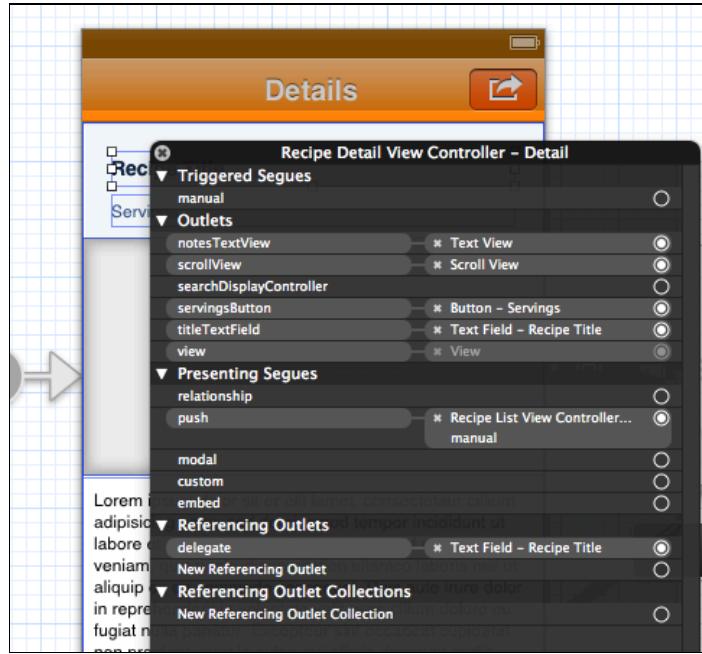
`actionTapped` will bring up an activity view controller that will allow your users to share recipes and perform other actions on them.

`cameraTapped` will show a photo picker so your users can select pictures from their photo album and add them to a recipe.

`localDoneTapped` is for the Done bar button item (added programmatically in `viewDidLoad`) that lets users dismiss the keyboard after editing a recipe's title or notes.

This method checks if the notes text view or the title text field are first responders and asks the relevant control to resign first responder (so the keyboard is dismissed). After doing this, the title and properties of the recipe object are updated as expected.

In your storyboard, connect the outlets to the corresponding items inside the Recipe Detail View Controller. Also, make the view controller the title text field's delegate.



Back in **RecipeDetailViewController.m**, add the `UITextFieldDelegate` protocol in the class extension:

```
@interface RecipeDetailViewController () <UITextFieldDelegate>
```

Then add the following text field delegate method:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [self.titleTextField resignFirstResponder];

    return YES;
}
```

The detail view controller will only support the portrait orientation. The content looks better in portrait. So enforce that:

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

Since there's no way to add more than one bar button item per side to a navigation bar, you need to add the buttons for the view programmatically. `viewDidLoad` is the perfect place to do so – replace it with the following code:

```
- (void)viewDidLoad
{
```

```
[super viewDidLoad];

self.actionButton = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemAction
    target:self action:@selector(actionTapped)];
self.cameraButton = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemCamera
    target:self action:@selector(cameraTapped)];
self.doneButton = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemDone
    target:self action:@selector(localDoneTapped)];

self.navigationItem.rightBarButtonItem =
@+[self.cameraButton, self.actionButton];
}
```

All three buttons are created via code and are set to call the methods that you added previously.

When the view is first loaded, only the camera and action buttons need to be visible, so those are displayed to the right of the navigation bar.

Right now, if your user selects a recipe from the list, no details are shown or updated. In order to change this, you're going to override `viewWillAppear:` and `viewWillDisappear:`. Add the code for those as shown below:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    self.notesTextView.text = self.recipe.notes;
    [self.servingsButton setTitle:[self.recipe servingsString]
        forState:UIControlStateNormal];
    [self.servingsButton setTitle:[self.recipe servingsString]
        forState:UIControlStateHighlighted];
    self.titleTextField.text = self.recipe.title;
}

- (void)viewWillDisappear:(BOOL)animated
{
    [self localDoneTapped];

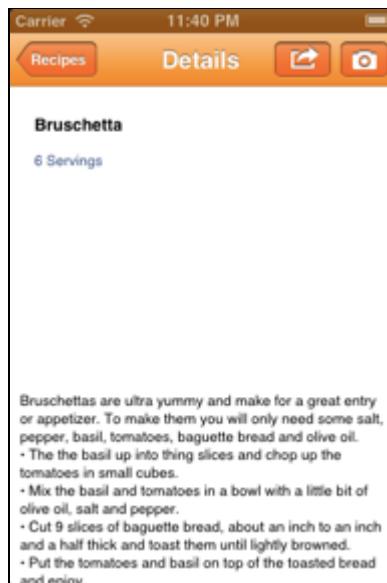
    AppDelegate *appDelegate = (AppDelegate * )[[UIApplication
        sharedApplication] delegate];
    [appDelegate saveContext];
```

```
[super viewWillAppear:animated];  
}
```

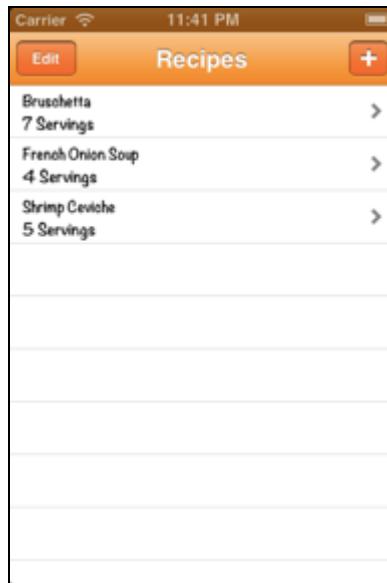
`viewWillAppear:` sets the recipe's notes, title and servings to the appropriate UI elements.

`viewWillDisappear:` calls `localDoneTapped` first, so the keyboard is dismissed (if it's shown) and the title and notes are updated, then the app delegate's `saveContext` is called to save any changes the user may have made to the recipe.

You've made a fair number of changes to the code, so build and run the project, give things a try, and make sure everything's working as expected:



Some things are working, but some are still missing. You can update the recipe's number of servings, title and notes and have those persisted as expected. If you made changes to the recipe and navigated back to the list of recipes, then you will have noticed that the list too is reflecting the new title and servings:



What's not fully working is the page view controller with the photos, the Done button that's supposed to appear when editing the title or the notes, the camera button and the action button.

Let's tackle the buttons first, beginning with the action button.

When users taps the action button (from the navigation bar), they will be presented with an activity view controller that will let them share a recipe with friends using Facebook, Twitter, or Sina Weibo. The activity view controller will also allow users to print a recipe, among other options that may vary based on a user's setup.

In **RecipeDetailViewController.m**, add the following import:

```
#import "Image.h"
```

Next, complete `actionTapped` as follows:

```
- (void)actionTapped
{
    NSString *titleString = [NSString stringWithFormat:@"I just
        made a delicious %@ recipe using RecipesKit",
        self.recipe.title];

    UIImage *activityImage;

    if (self.recipe.images.count > 0)
    {
        Image *image = [self.recipe.images anyObject];
        activityImage = image.image;
    }
}
```

```
NSArray *items = @[@[titleString];

if (activityImage)
{
    items = @[@[titleString, activityImage];
}

UIActivityViewController *activityViewController =
[[UIActivityViewController alloc]
initWithActivityItems:items applicationActivities:nil];

[self presentViewController:activityViewController
animated:YES completion:nil];
}
```

The method creates a custom title string to pass to the activity and retrieves any image (if available) from the recipe's image set. These items are then added to an array and an activity view controller is presented with the title and image for the recipe.

Run the project one more time and tap the action button. Try sharing the recipe using a social network or via email.



It works!

You want to show the Done button when the user is editing the title or notes. An easy way to know when the user is editing a field is to register for some keyboard notifications. Add the following code at the end of `viewWillAppear:`:

```
[[NSNotificationCenter defaultCenter] addObserver:self
```

```
    selector:@selector(keyboardWillHide:)
    name:UIKeyboardWillHideNotification object:nil];
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(keyboardWillShow:)
    name:UIKeyboardWillShowNotification object:nil];
```

Now `RecipeDetailViewController` will receive notifications when the keyboard is about to show and when it's about to hide. Of course, if you register a controller for notifications, then it's appropriate to also unregister when the notifications are no longer needed. Add this to the top of `viewWillDisappear:`:

```
[[NSNotificationCenter defaultCenter] removeObserver:self];
```

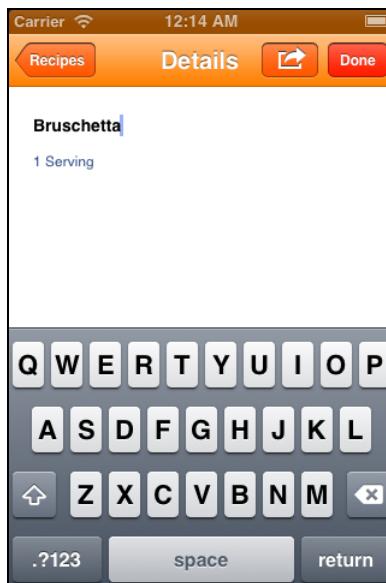
The methods that receive the keyboard notifications don't exist yet. It's time to add those to the view controller in order to avoid getting a crash at runtime.

```
- (void)keyboardWillHide:(NSNotification *)userInfo
{
    self.navigationItem.rightBarButtonItem =
    @[self.cameraButton, self.actionButton];
}

- (void)keyboardWillShow:(NSNotification *)userInfo
{
    self.navigationItem.rightBarButtonItem = @[self.doneButton,
    self.actionButton];
}
```

There's very little code going on here, and all these methods do is update the right buttons on the navigation bar, depending on whether or not the user is editing the title or the notes.

Run the project and navigate to a recipe's detail view. Tap the title or notes to begin editing, and notice how the camera button disappears and is replaced by the Done button. When the keyboard is dismissed, the camera button appears again on the navigation bar.



A scrollable photo buffet

A few key elements are still missing from the detail view. They are:

- Adding photos to a recipe using the image picker;
- Showing the photos in the page view controller using the new scroll style;
- Adapting the scroll view's content height so that the notes and the entire recipe view are visible when the keyboard is shown.

Let's start with the page view controller, since you need it to display any photos added with the image picker.

Embed segues, like the one used on the recipe detail screen for the page view controller, are like regular segues in that you can still use `prepareForSegue:` to handle any data exchange or custom logic before the controller is presented.

In **RecipeDetailViewController.m**, add support for the page view controller data source protocol:

```
@interface RecipeDetailViewController () <UITextFieldDelegate,  
UIPageViewControllerDataSource>
```

Then, add a property to hold the page view controller to the class extension:

```
@property (weak, nonatomic) UIPageViewController  
*pageViewController;
```

The property is declared as weak because it's part of the view hierarchy. This means it's going to be retained as long as the Recipe Detail View Controller is in view. No need for you to retain it twice.

You can set up the page view controller and its child controllers inside `prepareForSegue:`. Add the method for this as shown below:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender
{
    if ([segue.identifier
        isEqualToString:@"pageViewController"])
    {
        if (self.recipe.images.count > 0)
        {
            self.pageViewController =
                segue.destinationViewController;
            self.pageViewController.dataSource = self;

            PhotosViewController *photosViewController =
                [self.storyboard
                    instantiateViewControllerWithIdentifier:
                    @"PhotosViewController"];
            photosViewController.index = 0;

            Image *image = [[self.recipe.images allObjects]
                            objectAtIndex:0];

            photosViewController.image = image.image;

            [self.pageViewController
                setViewControllers:@[photosViewController]
                direction:
                UIPageViewControllerNavigationDirectionForward
                animated:YES
                completion:nil];
        }
    }
}
```

And add an import to the top of the file:

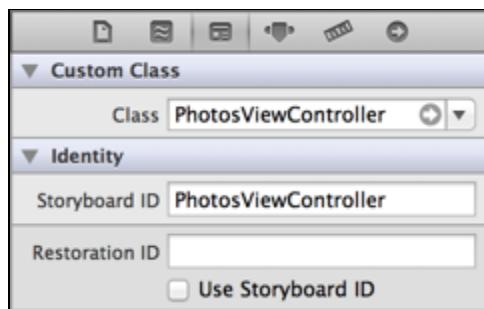
```
#import "PhotosViewController.h"
```

OK, time to go over `prepareForSegue:`!

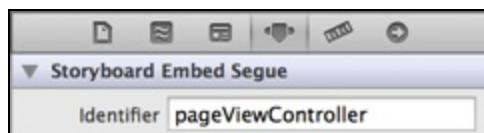
First you verify that the segue identifier corresponds to the page view controller's identifier (you'll set this up in the storyboard shortly). If it does, then you check for the number of images in the recipe. There's no need to create an instance of `PhotosViewController` if the recipe doesn't have any images.

If, however, there are pictures for the recipe, then the destination view controller is assigned to the `pageViewController` property and its data source is set to `RecipeDetailViewController`. An instance of `PhotosViewController` is created from the storyboard, the `image` property is set up and `PhotosViewController` is added to the hierarchy of controllers of `pageViewController`.

It's time to give some identifiers to the embed segue and Photos View Controller. Open the storyboard, locate the Photos View Controller, select it and go to the Identity inspector. Make sure the Class is set to **PhotosViewController**, and the Storyboard ID is too:



Select the embed segue from the Recipe Detail View Controller to the Page View Controller, open the Attributes inspector and set **pageViewController** as the Identifier.



The only thing that remains is adding the appropriate methods for the page view controller's data source.

Back in **RecipeDetailViewController.m**, add the following code:

```
- (UIViewController *)pageViewController:  
    (UIPageViewController *)pageViewController  
    viewControllerAfterViewController:(UIViewController *)  
    viewController  
{  
    PhotosViewController *previousViewController =  
        (PhotosViewController *)viewController;  
  
    NSUInteger pageCount = self.recipe.images.count;  
  
    pageCount--;  
  
    if (previousViewController.index == pageCount)  
    {
```

```
        return nil;
    }

    PhotosViewController *photosViewController =
        [self.storyboard
            instantiateViewControllerWithIdentifier:
                @"PhotosViewController"];
    photosViewController.index =
        previousViewController.index + 1;
    photosViewController.image =
        [[[self.recipe.images allObjects] objectAtIndex:
            photosViewController.index] image];

    return photosViewController;
}

- (UIViewController *)pageViewController:
    (UIPageViewController *)pageViewController
viewControllerBeforeViewController:
    (UIViewController *)viewController
{
    PhotosViewController *previousViewController =
        (PhotosViewController *)viewController;

    if (previousViewController.index == 0)
    {
        return nil;
    }

    PhotosViewController *photosViewController =
        [self.storyboard instantiateViewControllerWithIdentifier:
            @"PhotosViewController"];
    photosViewController.index =
        previousViewController.index - 1;
    photosViewController.image =
        [[[self.recipe.images allObjects]
            objectAtIndex:photosViewController.index] image];

    return photosViewController;
}
```

These two methods are called when a user swipes left and right inside the page view controller. Their function is to retrieve the corresponding controller that comes before or after the controller from which the user swiped.

Have a look at `pageViewController:viewControllerAfterViewController:` first. Since both methods are similar in functionality, analyzing one will help you understand the other.

The first thing the method does is cast the received view controller to an instance of `PhotosViewController`. The number of photos is retrieved from the images set of the `recipe` property, and the index of the `PhotosViewController` received is checked against the total number of images for the recipe.

This is why the `index` property inside `PhotosViewController` is useful – it lets you easily know the index of the controller and make the necessary checks before presenting it to the page view controller.

If the current `PhotosViewController` corresponds to the last image of the recipe's image set, then `nil` is returned since there are no further images to display. Otherwise, a new `PhotosViewController` instance is created from the storyboard, its index is assigned, the photo is retrieved and stored in the controller's property, and the new view controller is returned at the end of the method.

`pageViewController:viewControllerBeforeViewController:` does exactly the same thing, except that it gets called when a user swipes to a previous picture. The comparisons and checks are obviously the opposite of the ones in `pageViewController:viewControllerAfterViewController:`, but the logic is the same.

There are just two more (very short) methods you have to add in order to finish implementing the page view controller data source:

```
- (NSInteger)presentationCountForPageViewController:
    (UIPageViewController *)pageViewController
{
    return self.recipe.images.count;
}

- (NSInteger)presentationIndexForPageViewController:
    (UIPageViewController *)pageViewController
{
    PhotosViewController *photosViewController =
        [pageViewController.viewControllers lastObject];

    return photosViewController.index;
}
```

`presentationCountForPageViewController:` needs to return the total number of pages (controllers) that the page view controller has. Returning the count of the recipe's image set does this.

`presentationIndexForPageViewController:` is used to determine the current index of the page view controller. It retrieves the last object of the page view controller's

`viewControllers` property, casts it to `PhotosViewController` and gets its `index` property.

That `index` property you added to `PhotosViewController` is very handy, isn't it?

One last thing to do is make the page view controller's background transparent. Add the following line at the bottom of `viewDidLoad`:

```
[self.pageViewController.view setBackgroundColor:  
[UIColor clearColor]];
```

You are all done with the page view controller! Build the project to make sure everything's working. Build and run, and check out the delicious pictures!



- Start by making some chicken or beef stock (about 2 liters) with a bay leaf in the pot.
- Proceed to chop up about 4 white onions, 1 purple onion and a long onion.
- Bring a frying pan up to heat (medium-high) with some olive oil and start sautéing the onions for 30-40 minutes.
- With the onions nice and brown go ahead and reduce the heat to medium and add a spoon of sugar to the frying pan (this will make the onions caramelize). Stir the onions until caramelized (about 30 minutes).
- Add some garlic and let that sauté with the onions for a

However, a user can't add images to the recipe yet. For this, you need the system's image picker controller.

Still life with food

To implement the image picker controller, begin by adding the following property declaration to the class extension inside `RecipeDetailViewController.m`:

```
@property (strong, nonatomic) UIImagePickerController  
*imagePickerController;
```

Then add the following code to the end of `viewDidLoad`:

```
self.imagePickerController =  
[[UIImagePickerController alloc] init];  
self.imagePickerController.allowsEditing = YES;
```

```
self.imagePickerController.delegate =
(id <UIImagePickerControllerDelegate,
UINavigationControllerDelegate>)self;
self.imagePickerController.sourceType =
UIImagePickerControllerSourceTypePhotoLibrary;
```

This code creates a new instance of `UIImagePickerController` and stores it inside the `imagePickerController` property. It sets some attributes of the image picker, like the delegate, the source type and whether or not a user can edit an image when selecting it.

When the user finishes selecting an image, then the image picker's `imagePickerController:didFinishPickingMediaWithInfo:` delegate method gets called. This is the code you need to add in order to receive that image, process it and store it in the recipe:

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    // 1
    UIImage *selectedImage = ((UIImage *)[info
        objectForKey:UIImagePickerControllerEditedImage]);

    // 2
    float actualHeight = selectedImage.size.height;
    float actualWidth = selectedImage.size.width;
    float imgRatio = actualWidth/actualHeight;
    float maxRatio = 320.0/180;

    if (imgRatio != maxRatio)
    {
        if (imgRatio < maxRatio)
        {
            imgRatio = 480.0 / actualHeight;
            actualWidth = imgRatio * actualWidth;
            actualHeight = 480.0;
        }
        else
        {
            imgRatio = 320.0 / actualWidth;
            actualHeight = imgRatio * actualHeight;
            actualWidth = 320.0;
        }
    }

    // 3
```

```
CGRect rect = CGRectMake(0.0, 0.0, actualWidth,
    actualHeight);
UIGraphicsBeginImageContext(rect.size);
[selectedImage drawInRect:rect];
UIImage *croppedImage =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();

// 4
Image *image = [NSEntityDescription
    insertNewObjectForEntityForName:@"Image"
    inManagedObjectContext:self.managedObjectContext];
image.image = croppedImage;

[self.recipe addImagesObject:image];

// 5
NSError *error;
[self.managedObjectContext save:&error];

if (error)
{
    NSLog(@"Append Data Save Error = %@", [error
        localizedDescription]);
}

// 6
[self dismissViewControllerAnimated:NO completion:nil];
[self.navigationController
    popToRootViewControllerAnimated:YES];
}
```

That's a fair bit of code, but here's a breakdown of what it does:

1. The edited image is retrieved.
2. The picture's ratio is calculated to determine the appropriate width and height.
3. A new `Image` entity is inserted into the managed object context.
4. The `Image` entity you just created is added to the recipe's images set.
5. The context is saved (so that the changes are saved via Core Data).
6. The image picker is dismissed and the user is taken back to the recipe list.

The image picker is ready; it just needs to be called when the user taps the camera button. The camera button will present the user with an action sheet that will allow them to either delete all pictures from the recipe or select a new one from the

photo album. Replace the empty implementation for `cameraTapped` with the one shown below:

```
- (void)cameraTapped
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@""
        delegate:self
        cancelButtonTitle:@"Cancel"
        destructiveButtonTitle:@"Delete All Images"
        otherButtonTitles:@"Select Image", nil];
    [actionSheet
        showFromBarButtonItem:
            self.navigationItem.rightBarButtonItem animated:YES];
}
```

This is simply a regular action sheet with the appropriate buttons. Notice that `RecipeDetailViewController` is now the delegate of the action sheet. So implement that protocol in the class extension:

```
@interface RecipeDetailViewController () <UIActionSheetDelegate,
UIPageViewControllerDataSource, UITextFieldDelegate>
```

And add the necessary delegate method:

```
- (void)actionSheet:(UIActionSheet *)actionSheet
    clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 0)
    {
        [self.recipe removeImages:self.recipe.images];

        [self.navigationController
            popToRootViewControllerAnimated:YES];
    }
    else if (buttonIndex == 1)
    {
        [self presentViewController:self.imagePickerController
            animated:YES completion:nil];
    }
}
```

If the user taps the Delete button (index 0), then all images for the recipe are removed and the user is taken back to the recipe list. If the images button is tapped (index 1) then the image picker controller is presented to the user.

Everything's ready for the user to add or remove images from the recipe. But one last thing worth doing is properly implementing `didReceiveMemoryWarning` so you can be a good iOS citizen who cleans up memory when possible:

```
- (void) didReceiveMemoryWarning
{
    if ([self.view window] == nil)
    {
        _actionButton = nil;
        _cameraButton = nil;
        _doneButton = nil;
        _imagePickerController = nil;
        _notesTextView = nil;
        _pageViewController = nil;
        _scrollView = nil;
        _servingsButton = nil;
        _titleTextField = nil;
        self.view = nil;
    }

    [super didReceiveMemoryWarning];
}
```

Awesome! Build and run your app, open a recipe and try adding a few pictures, then swipe through them and delete them. Everything should be working perfectly!

Finishing touches

The new scroll style of the page view controller is very, very handy. It even includes the page control indicating the index of the current page. No custom scroll view programming had to be done to achieve this awesome effect.

Note: You might not see the page control unless you change the background of the container view to a color other than white. In fact, you'll need to change the background color for several views to get the uniform colored look you see above – consider it an additional exercise. ☺

But one thing remains somewhat dysfunctional. Tap on the notes text view and try editing its contents – the keyboard comes up and obstructs your view! You can still edit the text, but you have to do it blind.

This is why the scroll view was put in place, but there's a bit of work to be done in order to adjust its frame and content size to allow for proper editing of the notes.

Update `keyboardWillShow` and `keyboardWillHide` as follows:

```
- (void)keyboardWillHide:(NSNotification *)userInfo
{
    self.navigationItem.rightBarButtonItem =
    @[self.cameraButton, self.actionButton];

    self.scrollView.contentSize =
    CGSizeMake(self.view.frame.size.width,
               self.view.frame.size.height);
    self.scrollView.frame = self.view.bounds;
}

- (void)keyboardWillShow:(NSNotification *)userInfo
{
    self.navigationItem.rightBarButtonItem =
    @[self.doneButton,
      self.actionButton];

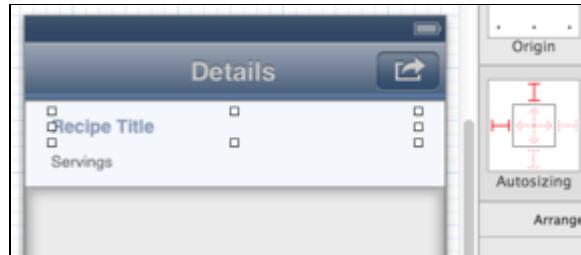
    CGRect keyboardFrame = [[userInfo.userInfo
        objectForKey:UIKeyboardFrameEndUserInfoKey] CGRectValue];
    self.scrollView.contentSize =
    CGSizeMake(self.view.frame.size.width,
               self.view.frame.size.height);
    self.scrollView.frame = CGRectMake(0, 0,
                                      self.view.frame.size.width, self.view.frame.size.height -
                                      keyboardFrame.size.height);
}
```

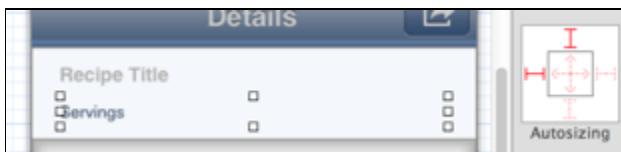
When the keyboard is about to appear, its frame is retrieved and used to resize the scroll view's frame, as well as the scroll view's content size. When the keyboard is about to hide, it resizes the scroll view and frame to the full size of the view.

You also need to make some changes to the storyboard's UI elements so they remain in position when the scroll view resizes. Open

MainStoryboard.storyboard and navigate to the Recipe Detail View Controller.

Here are the springs and struts configurations for the title and button (if your storyboard doesn't match, modify accordingly):





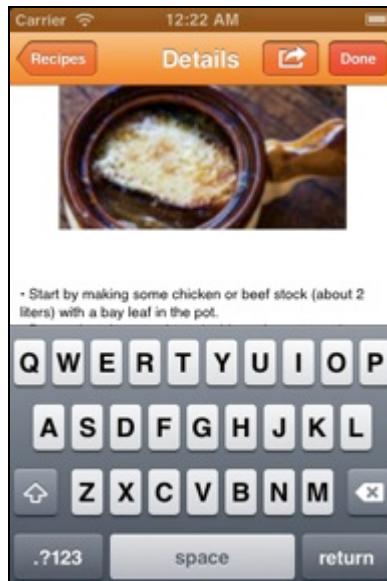
Now for the container view:



And finally the text view:



Great! Once again, build and run the project, open a recipe and tap the title or notes. The scroll view should resize as expected, all elements should remain in view and when you tap Done, everything should go back to the way it was before the keyboard was presented.



Try rotating the device while viewing a recipe's details... What? It flips to landscape orientation! Why is this happening if the Recipe Detail View Controller's `supportedInterfaceOrientations` specifies only the portrait orientation?

Remember when explaining the new orientation mechanics I mentioned how the app will only ask the root view controller or top-most full screen controller for its orientation? That's the trouble.

In this case, the root controller of the view hierarchy is the navigation controller. Whatever orientation the navigation controller supports is what its children will support. In order to overcome this, you could do one of two things:

- Subclass `UINavigationController`.
- Override `application:supportedInterfaceOrientationsForWindow:` inside the app delegate.

Which of the two is best? It depends. In this case, you're going to use the second choice and learn a new method for the updated orientation mechanics in iOS 6. It's also quicker, simpler, and has less overhead than overriding an entire class for a single method.

Go to **AppDelegate.m** and add the following method:

```
- (NSUInteger)application:(UIApplication *)application
    supportedInterfaceOrientationsForWindow:(UIWindow *)window
{
    NSUInteger orientations = UIInterfaceOrientationMaskAll;

    if (self.window.rootViewController)
    {
        UIViewController* presented =
    }
```

```
[[[UINavigationController *)
    self.window.rootViewController viewControllers]
lastObject];
orientations = [presented
    supportedInterfaceOrientations];
}

return orientations;
}
```

This returns the supported interface orientations for the top-most controller in the view. It's a handy way of allowing different orientations within your navigation controller's hierarchy, and will let you show the recipe list in portrait and landscape orientations, but only portrait orientation for the recipe details.

Run the app one more time and try to rotate the device when looking at a recipe's details. Notice how this view doesn't change from the portrait orientation, but the list view controller does.

Pull to refresh

There is one last thing you need before you can call the project done: pull-to-refresh functionality in your recipe list. You can use this new functionality to update your app via a remote RSS feed, Tweets, some JSON content, etc.

In order to keep things simple, though, RecipesKit is going to invert the sort order of the recipes when the user pulls to refresh the table view.

A refresh control is programmatically created and added to a table view controller. To add one to your project, go to **RecipeListViewController.m** and add the following lines of code at the end of `viewDidLoad`:

```
UIRefreshControl *refreshControl = [[UIRefreshControl alloc]
init];
[refreshControl addTarget:self
action:@selector(refreshControlValueChanged)
forControlEvents:UIControlEventValueChanged];
self.refreshControl = refreshControl;
```

You create the refresh control and add an action for the `UIControlEventValueChanged` event. This means that when the user pulls and initiates a refresh action, `refreshControlValueChanged` will be called. After you've set up the refresh control, you assign it to the table view controller's `refreshControl` property.

You haven't yet implemented the method that the refresh control is set to trigger when pulled. And there's a slight hitch – because the pull-to-refresh action is not actually fetching any data from a remote server, the action will be nearly instantaneous. So a user might not even notice it happening.

In order to provide a more realistic pull-to-refresh experience, you'll make it so that `refreshControlValueChanged` will call `refreshControlSortTriggered` after a short delay. Add a new property named `sortAscending` to the class extension that will be used to know the sort order of the recipe list.

```
@property (nonatomic) BOOL sortAscending;
```

Now, add the methods:

```
- (void)refreshControlSortTriggered
{
    self.sortAscending = !self.sortAscending;
    _fetchedResultsController = nil;
    [self.tableView reloadData];

    [self.refreshControl endRefreshing];
}

- (void)refreshControlValueChanged
{
    [self performSelector:@selector(refreshControlSortTriggered)
        withObject:nil afterDelay:0.85];
}
```

`refreshControlValueChanged` calls `refreshControlSortTriggered` after a delay of 0.85 seconds. `refreshControlSortTriggered` inverts the sort order, sets the existing fetched results controller to `nil` (so the data is reloaded in the new sort order) and the table view is reloaded. Lastly, the refresh control's `endRefreshing` method is called so it returns to its normal state.

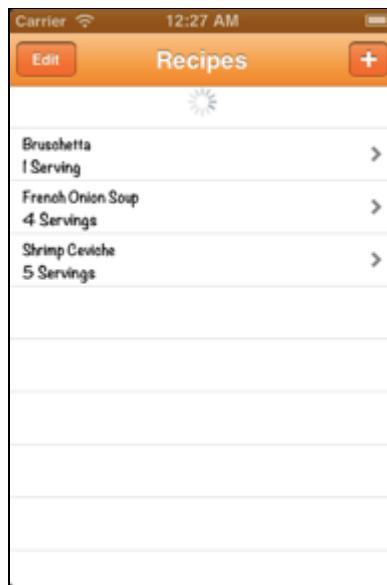
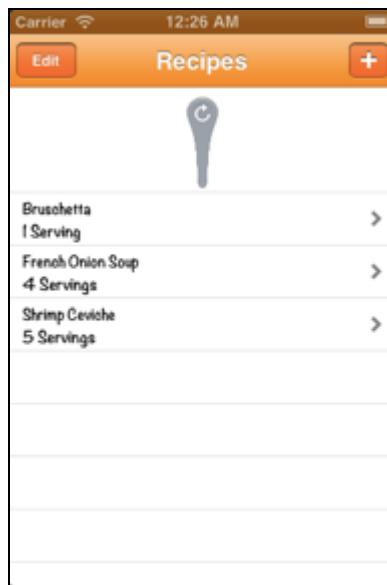
In order for this to work, there's one final thing you need to do: update the `fetchedResultsController` custom getter to use the `sortAscending` property for sorting.

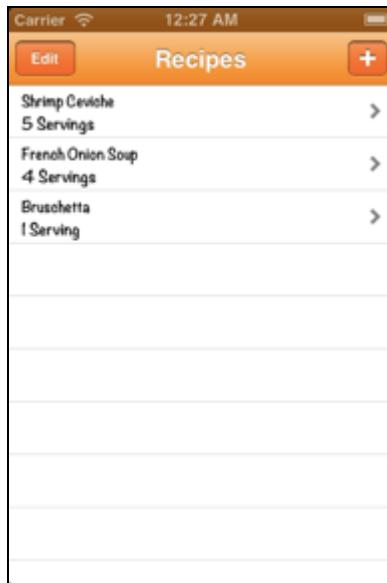
Find the `fetchedResultsController` getter and change the line that creates the sort descriptor to the following:

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
initWithKey:@"title" ascending:self.sortAscending];
```

Now, the recipe list's fetched results controller will use `sortAscending` to determine the sort order.

Run your app, add a few recipes with different titles and pull the table view to invert the sort order. Below is a sequence of screenshots of the refresh control in action:

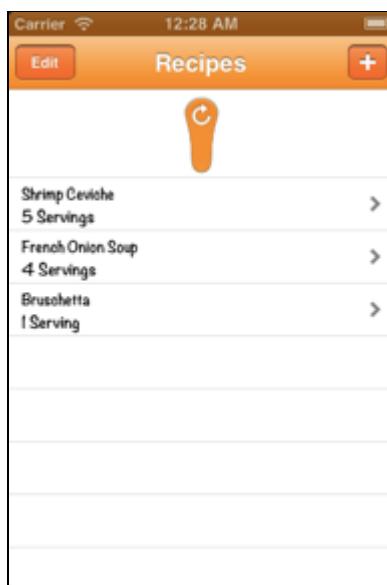




As an added bonus, how about tinting the refresh control? At the bottom of `viewDidLoad`, in **RecipeListViewController.m**, add this line:

```
self.refreshControl.tintColor = [UIColor orangeColor];
```

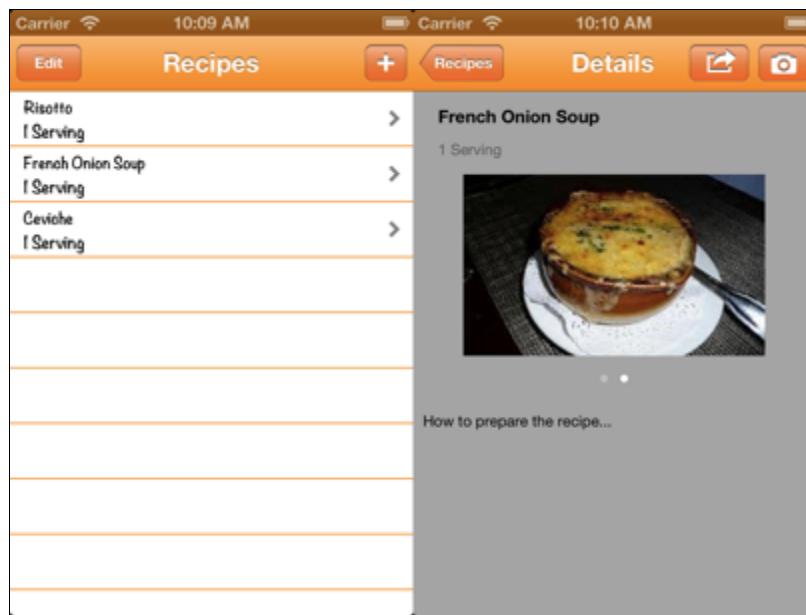
Then run the app once more and have a look:



It matches, how pretty! 😊

Congratulations, you've made it all the way to the end of the chapter! Give yourself a high-five, because you are now in possession of valuable new knowledge that will aid you in the creation of even cooler apps in the future!

Bonus: The final source includes a bit of extra design added, as shown below:



Where to go from here?

Congratulations – you have learned a ton of new concepts, tools and functionality introduced into Cocoa Touch in iOS 6!

Some of the concepts you learned in this chapter are:

- Status bar color tinting
- New memory management and cleanup methods
- Updates and changes to interface orientation code
- Methods for registering a cell for reuse from a nib or code
- `UIPageViewController` scroll style
- `UITableViewController` pull-to-refresh control
- Unwind segues inside Interface Builder
- View controller containment inside Storyboards
- Using the activity view controller

The possibilities are endless as to what you can do with these new tools. You no longer have to look for open source or third-party controls to add pull-to-refresh functionality to your table views. You can work with table view cells with even more power and using an approach that you're comfortable with, whether it's nibs or code.

Unwind segues and view controller containment inside Storyboards should help you speed up your workflow and reduce the amount of code you have to write. And I could go on!

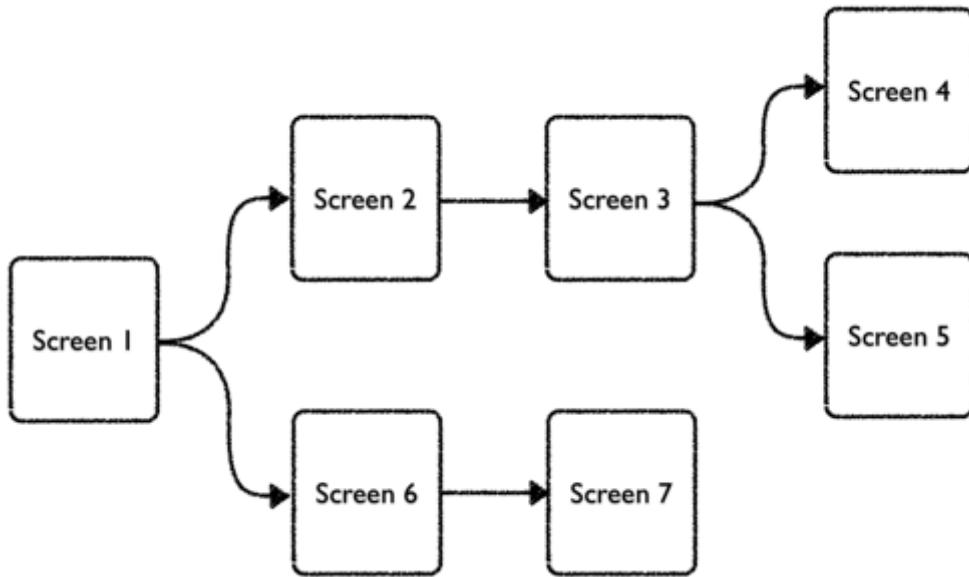
Thanks for sticking around and making it to the end of the chapter. ☺

Chapter 21: What's New with Storyboards

By Matthijs Hollemans

Storyboards were introduced with iOS 5 to simplify the creation of user interfaces. Before storyboards were introduced, you had to create the UI of each view controller in a separate nib file (or XIB if you prefer the iOS naming) and present new view controllers programmatically. With your screen designs spread over many different files, it could be hard to keep track of the workflow of your app.

A storyboard contains the designs of all your view controllers and the connections between them. One of the biggest advantages of using a storyboard is that it shows the entire application flow at a glance:



To navigate from one view controller in the storyboard to another, you use a *segue*, represented by the arrows in the picture above. Segues can be connected to buttons, table view cells, and other control elements. When a user taps the control, the segue transitions to the new view controller with an animation.

Segues work quite well. However, the arrows only go one way. The segues from iOS 5 can only present a new view controller. They can't dismiss it, nor do they

allow you to jump back to a previous screen in the navigation flow; that still has to be done programmatically. iOS 6 provides a solution for this with the new *unwind segues*.

Another omission in iOS 5's storyboards was support for view controller containment: that is, embedding one view controller within another. Even though iOS 5 came with new APIs for placing view controllers inside other view controllers (such as the `addChildViewController:` method), it was not possible to use this feature via the Storyboard Editor. Except for the standard navigation and tab bar containers, storyboards did not support custom containers.

iOS 6 introduces a new Container View element that finally allows you to embed view controllers within other view controllers in storyboards.

In this chapter, you will learn about these two new features: using the Container View for embedding one view controller inside another, and using unwind segues to do a "reverse" segue.

Note: By far the biggest change to storyboards – as well as nibs – is Autolayout, but that's such a huge feature that we've given it two chapters of its own! See Chapter 3, "Beginning Autolayout" and Chapter 4, "Intermediate Autolayout".

There have also been some important changes to the use of view controllers in general, such as the way autorotation is handled. You can read more about these changes in Chapter 20, "What's New with Cocoa Touch."

Tip: If you are completely new to storyboards, then you can find in-depth coverage in our book *iOS 5 by Tutorials*. A couple of preview chapters from the book also appear on raywenderlich.com:

- <http://www.raywenderlich.com/5138/beginning-storyboards-in-ios-5-part-1>
- <http://www.raywenderlich.com/5191/beginning-storyboards-in-ios-5-part-2>

The GumStats app

Let's say you're the proud owner of a state-of-the-art chewing gum vending machine. But this being the 21st century, you'd like to make an app that allows you to check in real-time how many gumballs you've sold. The vending machine is hooked up to the Internet and sends a message to a web service every time someone buys a piece of gum. Your app, GumStats, downloads these statistics and shows them on your iPhone:



This chapter shows you how to build the UI for such an app using the new storyboard features of iOS 6.

To save you some time, I have already prepared a starter app that you can find with this chapter's resources. It's a fairly simple project, based on Xcode's Master-Detail app template.

The first screen of the app contains a list of dates and the total number of sales for each date. Tapping on a date opens a new screen with a graph for that day. The Edit button reveals the complete list of data points for that day. Each data point represents an hour's worth of sales. Finally, you can change a specific data point by typing into a text field.

The storyboard for the initial version of the app looks like this:



In storyboard terminology, every screen is usually known as a scene. And every scene is usually represented by a view controller and its views. The GumStats app contains six scenes in total: four view controllers and two `UINavigationController`s.

Play with the app for a bit to get a feel for how it works, and browse through the Xcode project to see how it's been implemented. It's a pretty simple app and the code is straightforward, especially if you've used storyboards before.

The data is provided by `Record` objects. The `DaysViewController`, which is the initial screen that lists the dates, contains an `NSArray` of `Record` objects.

If this was a real app – and you really had a chewing gum vending machine – you would fill these `Record` objects with actual data you received from the web service.

But since that topic is beyond the scope of this book, we will have to make do with made-up data.

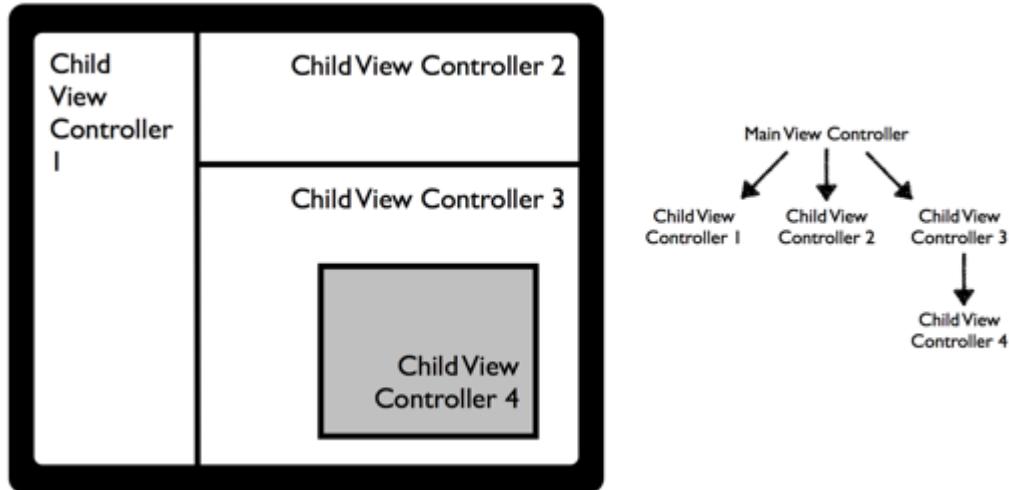
Embedding view controllers

In this section, you'll learn how to embed the contents of one view controller inside another using the new Container View element. When you're done, the app will combine the list of dates and the graph into a single screen, like this:



View controller containment is what happens when you put more than one view controller on the same screen. Apple has provided a few standard containers for some time now, such as the `UINavigationController` and the `UITabBarController`, but as of iOS 5 it is possible to write your own view controller containers. Unfortunately, the Storyboard Editor did not support this feature, so you had to write your own containment code by hand.

Embedding view controllers is especially useful on the iPad. The iPad's screen is big enough that it practically encourages you to subdivide it into several distinct parts. It's much better practice to put the logic for each part into its own view controller, rather than lumping it all inside one big, ungainly view controller that becomes harder and harder to maintain as more functionality is added.



On the iPhone there is less of a need for this, because of the smaller screen. Each view controller usually takes over the entire screen. But it's still nice to be able to split up the functionality of your screens among several view controllers.

Note: Before iOS 5, if you wanted to embed one view controller within another you had to manage everything yourself, including forwarding rotation, layout events to the child view controller and other such housekeeping tasks. Because you can't control what UIKit does behind the scenes, this wasn't always easy to get right.

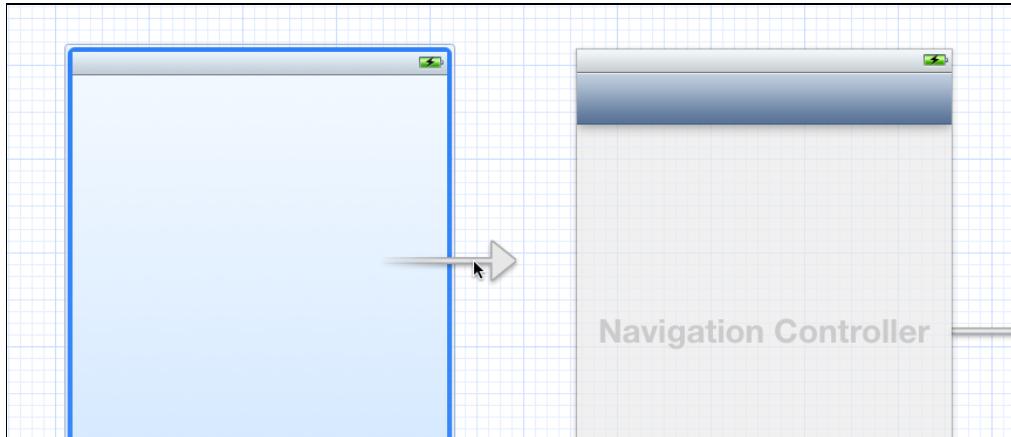
The view controller containment APIs that were added in iOS 5 made it a lot easier to create your own containers, but that still required you to write a fair amount of code. The new Container View element in iOS 6 takes away some of that work, but it's not a complete replacement for the API. You will still need to create more complex containers via code.

You can find an introduction to the view controller container APIs in *iOS 5 by Tutorials*.

Enough theory, let's see some action!

Open **MainStoryboard.storyboard** and drag a new View Controller object from the Object Library onto the canvas. Put it to the left of the first scene, which is a navigation controller.

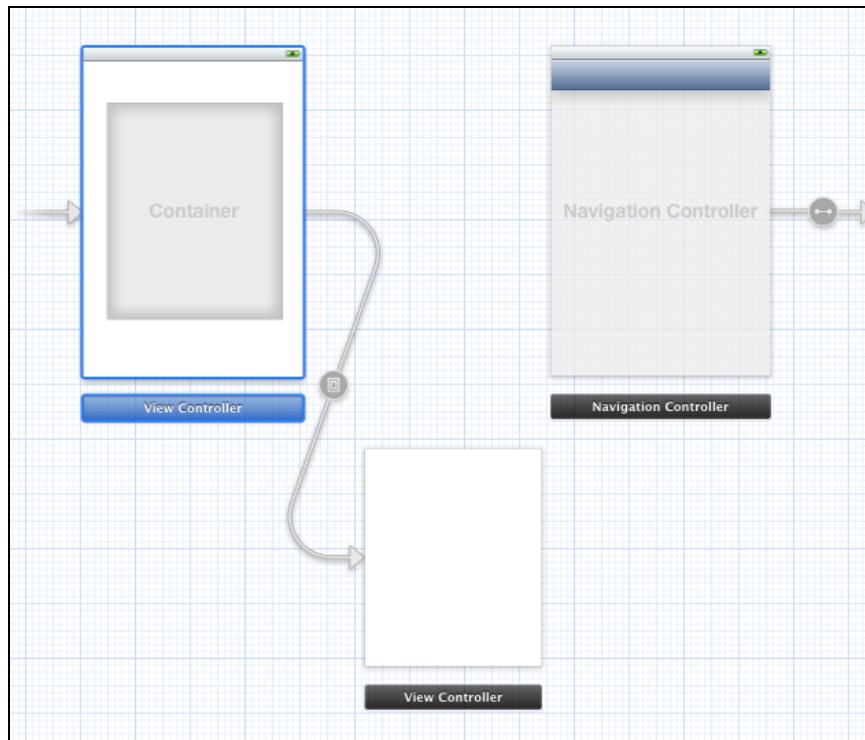
The newly-added view controller must become the initial view controller, so that it will be the first scene the app loads. Currently the navigation controller is the initial view controller, as indicated by the arrow that points to it. Click on the arrow and drag it to the new view controller to change this.



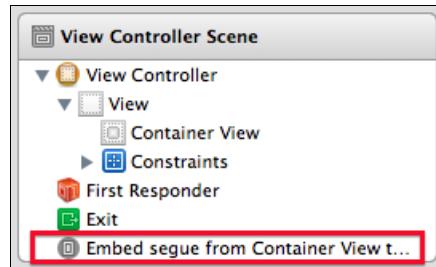
Locate the Container View object in the Object Library. It looks like this:



Make sure you are fully zoomed in (you cannot modify view controllers in the Storyboard editor while you are zoomed out), and drag a container view onto the new view controller. The container view comes with a new view controller of its own that is attached by a new type of segue, the *embed segue*. The storyboard should now look something like this:



There is a new gray container block inside the original view controller. It is connected to a second view controller using the new embed segue. You can also see this segue in the Document Outline on the left:

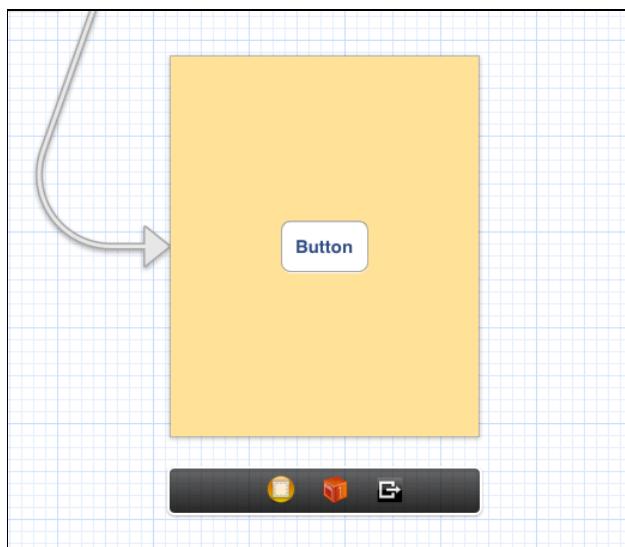


Note: If you haven't tried the new Auto Layout feature yet, then you might wonder what the Constraints item is for. By default, all new storyboards and nibs have Auto Layout enabled. With this new feature the relationships between the various views in your view controllers are described using these constraints. For more information about Auto Layout, see Chapters 3 and 4, "Beginning and Intermediate Auto Layout".

Notice that this new view controller at the other end of the umbilical cord is smaller than a regular view controller. In fact, it has the same dimensions as the container view. If you resize the container view, the child view controller will change sizes accordingly. Try it out.

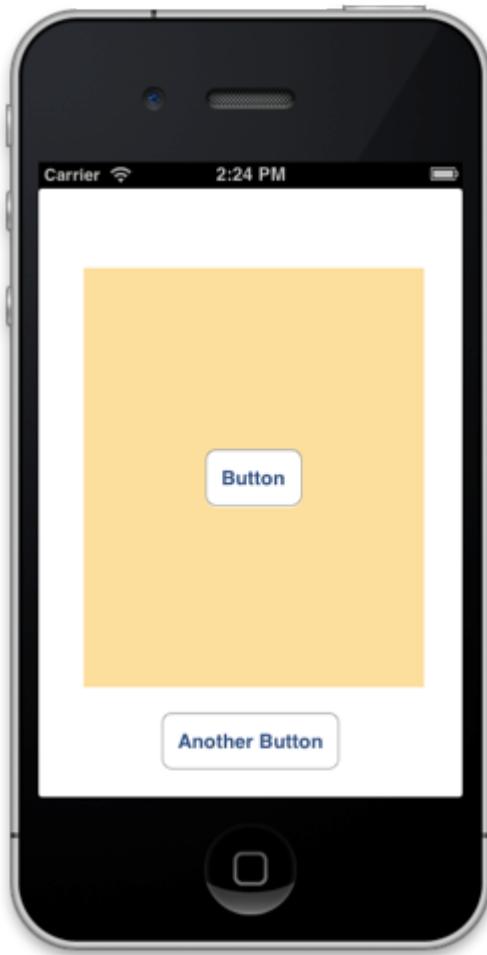
When you run the app, the container view will display whatever is in the attached view controller. Right now there is not much to see – everything is white – so let's put something inside this embedded view controller to demonstrate that this works.

Set the background color of the embedded view controller to something other than white and drop a `UIButton` in there. Something like this:



Also put some controls such as buttons and labels into the initial view controller.

Now when you run the app, it looks like this:



The space that was taken up by the container view in the storyboard is magically replaced by the contents of the embedded view controller.

The button in the center lives in the embedded view controller, and if you tap it, the action gets handled by that embedded view controller. The button at the bottom, however, lives in the main view controller, and its actions will be handled there.

Note: The container view in the parent view isn't actually a real view. It's just a placeholder element in the Storyboard Editor that shows you where the embedded view controller will be placed and how big it will be. This is why you can't drop any subviews, such as buttons or labels, directly onto the container view. Instead, you're supposed to put them inside the connected child view controller.

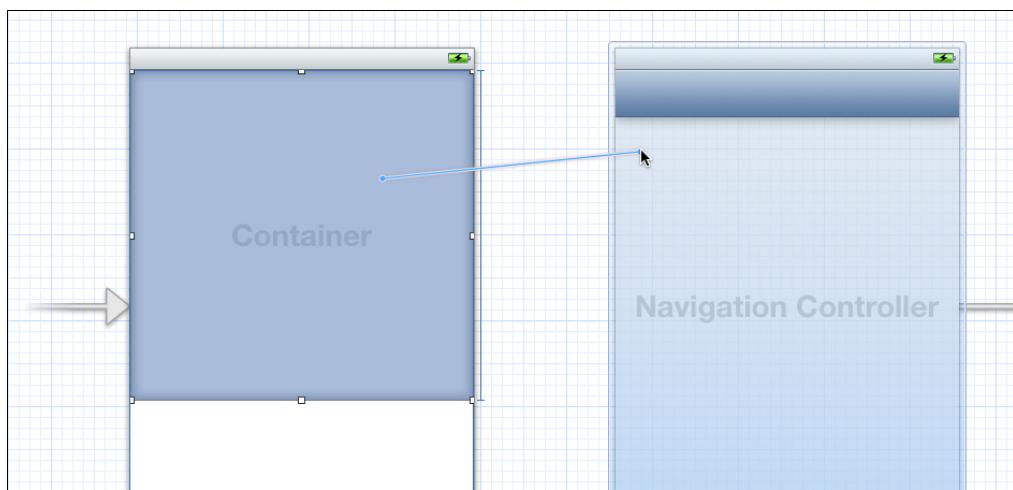
Adding a container view is basically all you have to do to make the contents of one view controller appear inside another. UIKit will take care of forwarding all rotation and layout events to this embedded view controller. Pretty simple, huh?

Two for the price of one

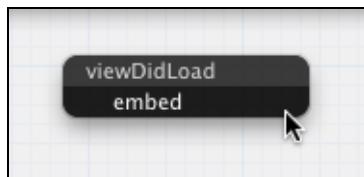
Now you'll change the GumStats app to place both the table of dates and the graph inside the main screen. These two plots of screen real estate will continue to be controlled by their own view controllers. The only difference will be that now, instead of having two separate screens in the app for the dates and the graph, they'll be combined into a single overview.

Remove the embedded view controller (the one with the colored background) from the storyboard by selecting its view controller and deleting it. The embed segue should disappear, but the Container View stays. (Also remove any buttons or other controls you added to the parent view.)

Resize the Container View to 320 points wide, 308 points high, and align it with the top of the view controller. Now Ctrl-drag from the Container View to the Navigation Controller on the right:

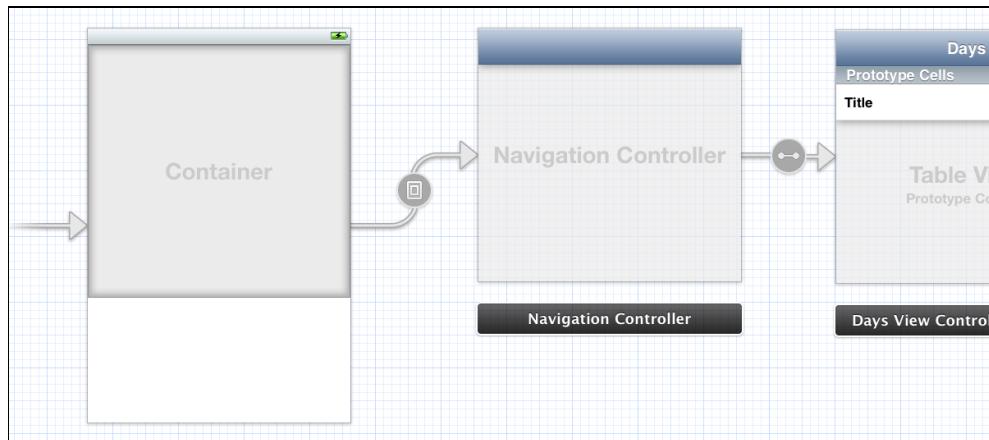


From the small popup that appears, choose "embed":

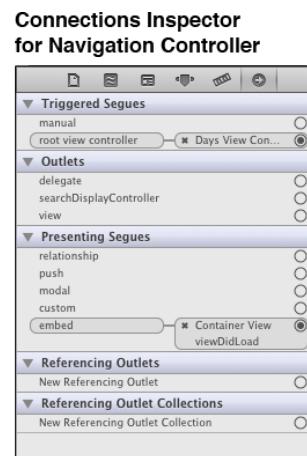
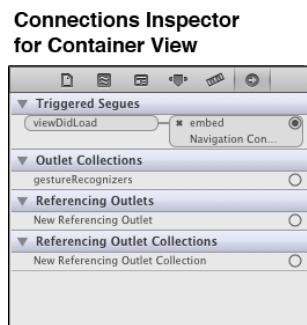


Now the editor creates a new embed segue between the Container View on the left and the Navigation Controller on the right. Because the size of the Container View dictates the size of the embedded view controller, the Navigation Controller and everything that is attached to it will also resize. (Except for the Edit View Controller all the way to the right – that scene is presented modally and therefore is not part of the navigation chain.)

The storyboard now looks as follows:



You can also see this new embed segue in the Connections Inspectors for the Container View (under "Triggered Segues") and the Navigation Controller (under "Presenting Segues"):



Note: An embed segue shows up as being connected to the `viewDidLoad` method, but that's just a Storyboard Editor convention – you can't hook it up to a different method.

Run the app. It still works pretty much the same way it did before, except that now less of the initial table view is visible (only 6 rows). What you're seeing here is that the table view controller is successfully embedded inside the new main view controller. The rows at the bottom aren't visible because the Container View simply doesn't extend that far.

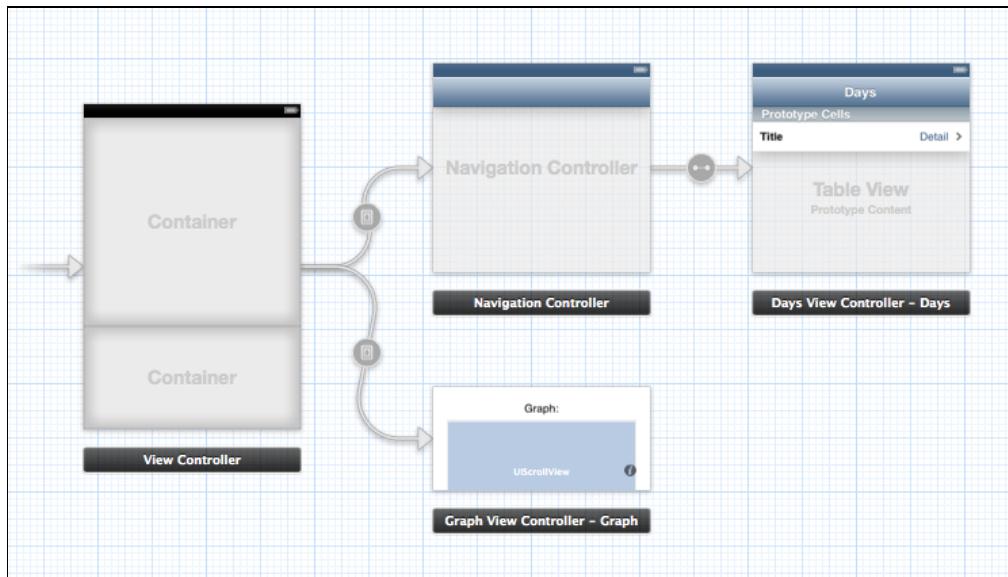


Now when you tap a date, no graph shows up. That's due to the Autolayout constraints. Embedding the navigation controller has resized the Graph View Controller as well, and this has made the graph view invisible. No matter, because you are going to take the Graph View Controller out of the navigation flow anyway and place it on the main screen.

Select the push segue between the Days View Controller and the Graph View Controller (by clicking on it) and delete it. Because the Graph View Controller is no longer linked to the embedded navigation controller and the Container View, it restores to its original size.

Also remove the push segue that goes from the Graph View Controller to the Measurements View Controller. (Xcode will flash a warning that scenes have become unreachable. Don't worry about that; you'll fix it later.)

Drag a new Container View into the main view controller, below the other one. Make it 320 points wide and 152 points high so that it takes up the remaining space. This Container View also comes with its own embedded view controller – delete it. Then move the Graph View Controller to its place and create a new embed segue to that (by Ctrl-dragging as before). The leftmost portion of the storyboard now looks like this:

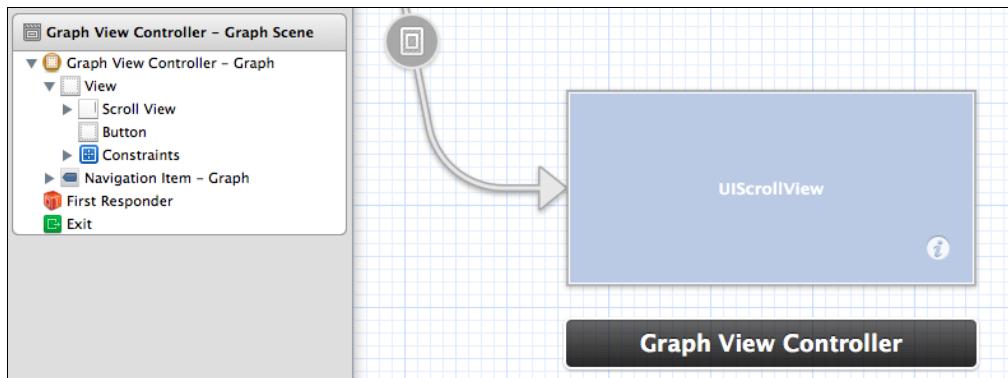


Unfortunately, the content of the Graph View Controller is all squashed. Let's fix that.

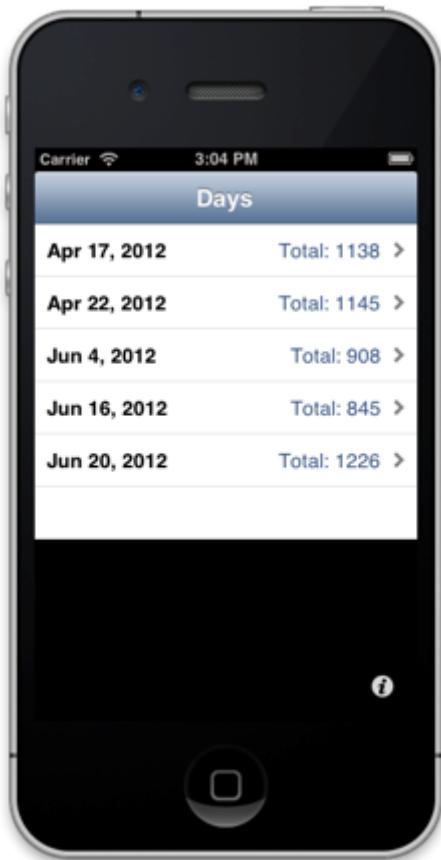
This scene contains three subviews: a label that says "Graph:", a scroll view that contains the actual graph (because the graph view is wider than the screen), and an info button at the bottom.

Because this view controller needs more room to work with, remove the label altogether.

Resize the `UIScrollView` to 320 by 152 points and place it at coordinates 0, 0. The info button can stay where it is, but it will look better if you change its type from Info Dark to Info Light (in the Attributes Inspector).



Run the app again. It now looks like this:



Believe it or not, you have successfully combined the contents of two view controllers on the same screen! It's just that the graph is black because you haven't told the Graph View Controller to display anything yet.

Note: If you try out the app on the "iPhone (Retina 4-inch)" simulator or an iPhone 5, the graph view may not appear properly at the bottom of the screen.

This happens because the Auto Layout constraints between the two Container Views haven't been set up properly yet. Without going into too much depth on Auto Layout, this is how the constraints should be set up:

- * The top Container View has a Vertical Space constraint to the bottom of the window of 152 points.
- * There is a Vertical Space constraint of 0 points between the two Container Views, so that the top one keeps the bottom one in place.
- * The bottom Container View should have no Height constraint, or a Vertical Space to the top of the window.

With these constraints in place, the table view from the top Container View can take up the extra space on the iPhone 5. To learn more about Auto Layout, see chapters 3 and 4.

Previously, sending the data to display on the Graph View Controller happened with a push segue from the Days View Controller that was triggered by a tap in the table view.

If you've worked with storyboards before, you know that when a segue happens, the `prepareForSegue:` method is called on the "source" view controller (the one that performs the segue), which allows it to configure the "destination" view controller (the one that is opened).

This is what `prepareForSegue:` from **DaysViewController.m** currently does:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                  sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"ShowGraph"])
    {
        NSIndexPath *indexPath = [self.tableView
                                  indexPathForCell:sender];

        GraphViewController *controller =
            segue.destinationViewController;

        Record *record = _records[indexPath.row];
        controller.record = record;
    }
}
```

It gets the `Record` object that corresponds to the table view cell that the user tapped, and puts it into the `record` property of the `GraphViewController` object.

However, this push segue no longer exists – you just deleted it from the storyboard – and therefore this method no longer gets called.

Because `DaysViewController` and `GraphViewController` now share a single screen, the `DaysViewController` should no longer open the `GraphViewController` in a new screen when you tap one of its rows. Instead it should tell the existing `GraphViewController` instance to update its graph. That means you can no longer use a segue here.

Remove the `prepareForSegue:` method from **DaysViewController.m**.

Connecting the two child view controllers

Because `DaysViewController` needs to talk directly to `GraphViewController`, it needs to have a new property that points at the `GraphViewController` instance.

Change `DaysViewController.h` to:

```
@class GraphViewController;

@interface DaysViewController : UITableViewController

@property (nonatomic, strong) GraphViewController
    *graphViewController;

@end
```

Add the following to the bottom of `DaysViewController.m`:

```
#pragma mark - UITableViewDelegate

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    Record *record = _records[indexPath.row];
    self.graphViewController.record = record;
}
```

When the user taps a row from the table view, `DaysViewController` now looks up the corresponding `Record` and passes it along to `GraphViewController`.

That's simple enough, but where do you actually set that `GraphViewController` property? How does `DaysViewController` find out who `GraphViewController` is?

If you were thinking, "I'll simply make it an `IBOutlet` property and hook up everything in the Storyboard Editor," then you are going to be disappointed. At the time of writing this chapter, there is no way to do this in the Storyboard Editor. (Try it out if you don't believe me!)

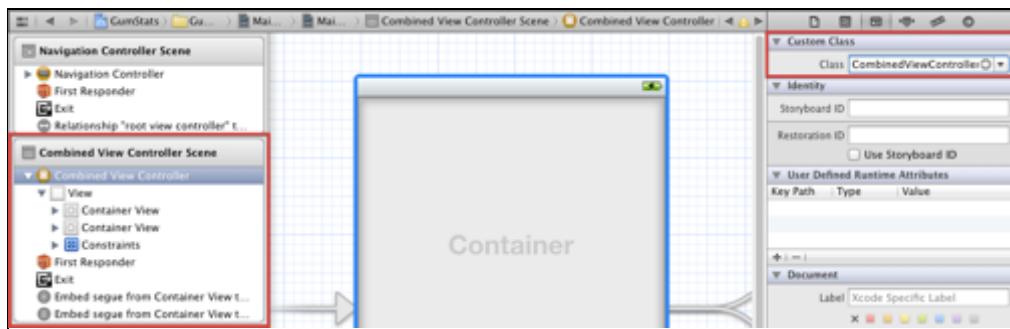
Note: If you Ctrl-drag from one view controller to the other in the storyboard, all it lets you do is create segues between them, and that's exactly what you don't want to do here. It is unfortunately impossible with the current version of Xcode to make connections that are not segues between different view controllers.

However, there is a solution to this problem. Embed segues are still segues, and therefore the `prepareForSegue:` method is called when they load. In order to take advantage of that, you first have to make a new `UIViewController` subclass.

From the File menu, choose **New\File...**. Pick the Objective-C class template and fill in the next screen as follows:

- Class: `combinedViewController`
- Subclass of: `UIViewController`
- Targeted for iPad: no
- With XIB for user interface: no

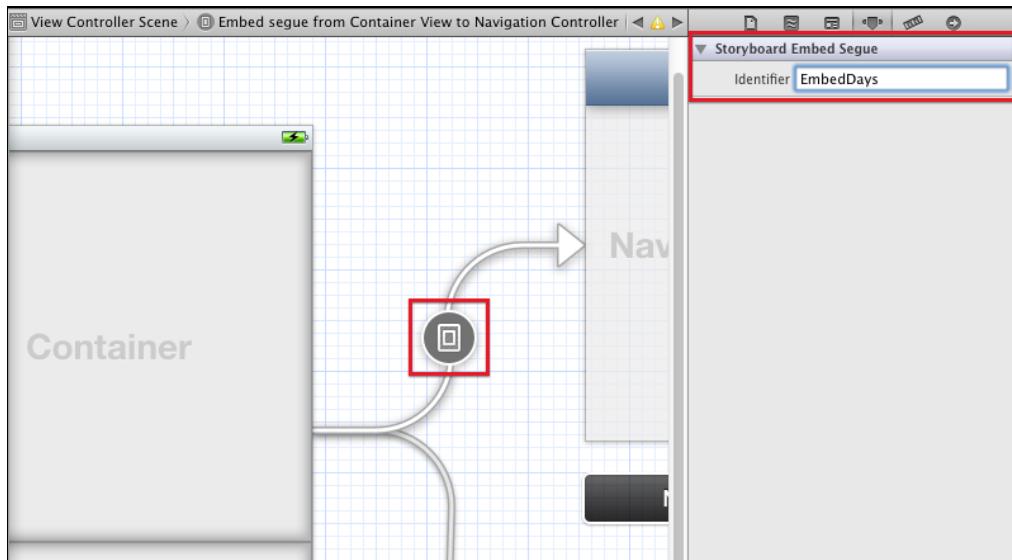
In the storyboard, select the initial view controller and change its Custom Class field (in the Identity Inspector) to `CombinedViewController`:



From now on, when the app loads that view controller, it actually creates an instance of `CombinedViewController` rather than just a plain `UIViewController`.

You can add the `prepareForSegue:` method to `CombinedViewController` in order to intercept the embed segues. For that to work, though, you must first give identifiers to these segues, so that you can tell them apart. After all, this view controller has two Container Views, each with its own segue. It will be handy to know which is which.

Select the embed segue that belongs to the top Container View. You can do this from the Document Outline or by clicking the icon that sits on the arrow. In the Attributes Inspector, type "EmbedDays" into the Identifier field:



Repeat this for the other embed segue. Name it "EmbedGraph."

Now that you have given the embed segues identifiers, you can give the initial view controller a `prepareForSegue:` method to do something with them.

Add the following to the top of **CombinedViewController.m** (replacing the existing empty class extension):

```
#import "DaysViewController.h"
#import "GraphViewController.h"

@interface CombinedViewController ()
@property (nonatomic, weak) DaysViewController
    *daysViewController;
@property (nonatomic, weak) GraphViewController
    *graphViewController;
@end
```

This defines two new properties that `CombinedViewController` will use to keep track of the two view controllers that it embeds. Remember, you're doing this because the Storyboard Editor will not let you directly connect the view controllers to outlets.

Add the `prepareForSegue:` method to the end of **CombinedViewController.m** (before the final `@end`):

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"EmbedDays"])
    {
```

```
UINavigationController *navigationController =
    segue.destinationViewController;
self.daysViewController = (DaysViewController *)
    navigationController.topViewController;
}
else if ([segue.identifier isEqualToString:@"EmbedGraph"])
{
    self.graphViewController =
        segue.destinationViewController;
}
}
```

First the code checks which segue is happening. Because the view controller has two embed segues, `prepareForSegue:` will be called twice, once for the “EmbedDays” segue and once for the “EmbedGraph” segue.

Note: Because you added the “EmbedDays” segue first, it will likely be called first, but it’s probably a good idea to not depend on any particular performance order for the two segues. If UIKit changes how it works behind the scenes, then any assumptions you make may no longer hold true.

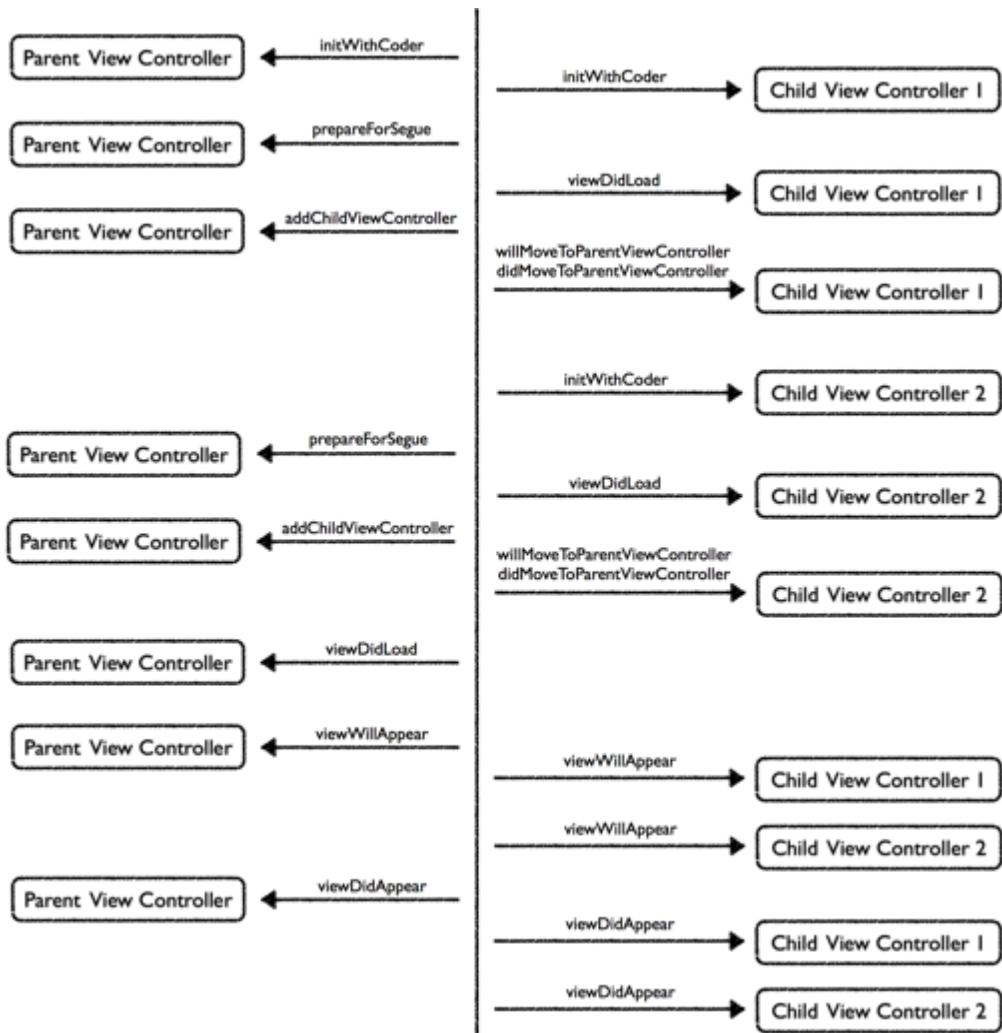
What this `prepareForSegue:` does is pretty simple: it looks at `UIStoryboardSegue`’s `destinationViewController` property to find `DaysViewController` and `GraphViewController`, respectively. Because the “EmbedDays” segue doesn’t directly connect to `DaysViewController`, but to a `UINavigationController`, it has to do a little more work for that one.

Great, you have filled in the `daysViewController` and `graphViewController` pointers. Now what? Well, you still need to tell the Days view controller about its Graph counterpart, so somewhere you will also need to do:

```
self.daysViewController.graphViewController =
    self.graphViewController;
```

Unfortunately, `prepareForSegue:` is not a good place for this. If the “EmbedDays” segue is performed first, the `graphViewController` property will still be `nil`.

To solve this conundrum, let’s take a look at the sequence of events that happens when you embed a view controller inside another:



The very first thing that happens is that UIKit calls `initWithCoder:` to initialize the parent view controller, `CombinedViewController`. (Not `initWithNibName: bundle:`, because the view controller is being loaded from a storyboard.)

Immediately afterward, the first embedded view controller – `DaysViewController` – is allocated and its `initWithCoder:` gets called in order to load it from the storyboard as well.

Directly after `initWithCoder:`, `prepareForSegue:` is called. This happens early on in the lifetime of the parent view controller, long before it receives the `viewDidLoad` message. That means you can connect `DaysViewController` to `GraphViewController` in `viewDidLoad`.

Change the `viewDidLoad` method in **CombinedViewController.m** to:

```

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

```
self.daysViewController.graphViewController =
    self.graphViewController;
}
```

You're guaranteed that both these properties are populated by now, so the above will work.

Run the app and tap on the date rows. Hmm, still nothing much happens. You'd expect the graph view to show the data for the selected date, but it stays black.

Here's what's missing: previously the `record` property of `GraphViewController` was set in the segue coming from `DaysViewController` (the segue that you deleted earlier). That property was guaranteed to be set before `GraphViewController`'s view was loaded, and so the Graph View Controller took advantage of that by simply redrawing the graph in its `viewDidLoad`.

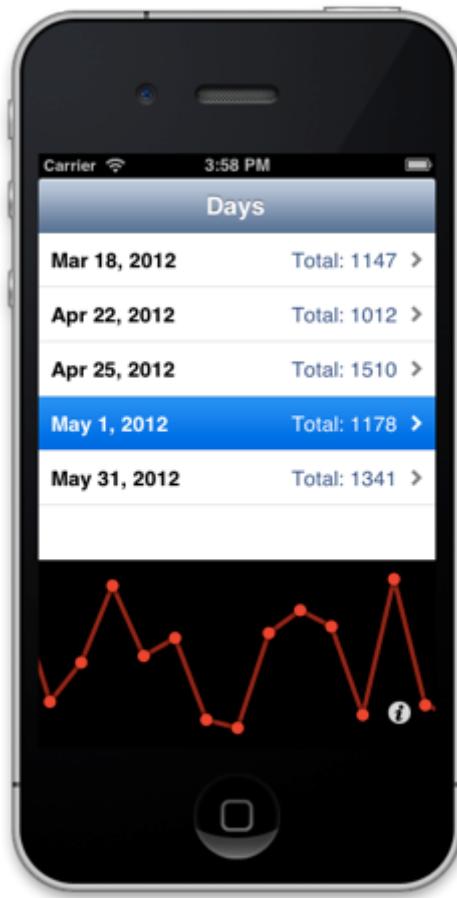
Here, however, `GraphViewController`'s `viewDidLoad` happened a long time before you tapped the date row, so you need another way to trigger the graph redraw. The easiest solution is to provide your own setter for the `record` property.

Add the following method to **GraphViewController.m**:

```
- (void)setRecord:(Record *)newRecord
{
    if (_record != newRecord)
    {
        _record = newRecord;
        [self redrawGraph];
    }
}
```

The `redrawGraph` method (which already exists), takes the list of values from `self.record` and assigns them to the `GraphView`, which then redraws itself.

Run the app again, and now tapping a row does show the graph for that date:



Tap the Info button in the bottom-right corner to toggle how the graph draws the lines (with or without the circles).

Why bother?

You might be wondering, "Why go through all this trouble? Why not simply take the `GraphViewController` component out of the `GraphViewController` and the `UITableView` out of the `DaysViewController` and combine them inside a single view controller, rather than embedding two different controllers that you then have to set up to talk to each other?"

Here's why: the `DaysViewController`, which is a subclass of `UIViewController`, is very good at dealing with table-based data. The `GraphViewController`, on the other hand, is very good at dealing with graphs. This is a clean *separation of concerns* – each view controller does what it is best at. There is nothing stopping you from combining all this logic into a single `UIViewController`, but then the code becomes harder to read and the concerns get muddled up.

`GraphViewController`, for example, has a toggle button for switching between how the graph draws the lines. The action method for that button, `toggleGraph:`, lives in `GraphViewController.m`. `DaysViewController` and `CombinedViewController` do not

have to concern themselves with this particular feature – it is something that belongs specifically to the graph and has nothing to do with the table view. In a situation such as this, it is cleaner to split up the logic into two separate view controllers so that each deals with their own specific problem domain.

Another advantage of this approach is reuse and flexibility. As you just saw, because the graph and list of dates were handled by two different view controllers, it was very easy to reorganize the screens into a completely different look. It also makes it easier to reuse view controllers in other projects, or when you port your app to a device with a bigger screen - like the iPad!

Of course, there are limits to how far you can – and should – take this. On the iPhone you probably wouldn't embed more than two view controllers into a single one, due to screen size constraints. On the iPad you can get away with a bit more because the screen is bigger. A good rule of thumb is: if the layout becomes too complex, then divide it up.

Tip: As you're probably quite aware, `UITableViewController` is a very handy class for dealing with tables. It offers extra functionality that you don't get when you just put a `UITableView` inside a regular `UIViewController`, such as automatic clearing of any selection when the table is displayed, flashing the table view's scroll indicators when it first appears, and more.

One of the downsides of `UITableViewController` is that its table view is also its main view, and therefore must occupy the entire content area. You can't easily add views above or below the table view (except in the header and footer area).

Embedded view controllers allow you to work around this problem, exactly the way you've done in this chapter. Create a new `UIViewController` that holds the additional views, and place the `UITableViewController` inside a Container View.

Initializing the child view controllers

The `prepareForSegue:` in `CombinedViewController` is also a good place to put data into the properties of the embedded view controllers. For example, it might be a good idea to move the data model – the array of `Record` objects – one level higher in the view controller hierarchy, out of `DaysViewController` and into `CombinedViewController`.

Make the following changes to `CombinedViewController.m`. Add an import:

```
#import "Record.h"
```

Add an instance variable to hold the array:

```
@implementation CombinedViewController
{
    NSArray *_records;
}
```

Now cut the `initWithCoder:` and `makeFakeRecord:` methods out of **DaysViewController.m** and paste them into **CombinedViewController.m**. You can get rid of the `initWithNibName:bundle:` method in **CombinedViewController.m** that was put there by the Xcode template, as it is never used in this project.

Remove the `_records` instance variable from **DaysViewController.m**, but put this property in its place in **DaysViewController.h**:

```
@property (nonatomic, strong) NSArray *records;
```

If you want to follow proper coding practices, you really should replace any occurrences of `_records` in **DaysViewController.m** with `self.records`. As we discussed in Chapter 2, “Programming in Modern Objective-C”, if you make a property for a variable it’s good practice to access it through the property rather than accessing it directly through its instance variable.

If you now run the app, the table view is empty. That makes sense. You moved the data model to `CombinedViewController`, but never told `DaysViewController` about it. In **CombinedViewController.m**, change `prepareForSegue:` to:

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

        self.daysViewController = (DaysViewController *)
            navigationController.topViewController;

        self.daysViewController.records = _records;
    }
    else if . . .
```

After it gets a pointer to `DaysViewController`, this method now passes it the data object. That works because `prepareForSegue:` is called after `initWithCoder:`, so at that point the `_records` array is already allocated and filled in.

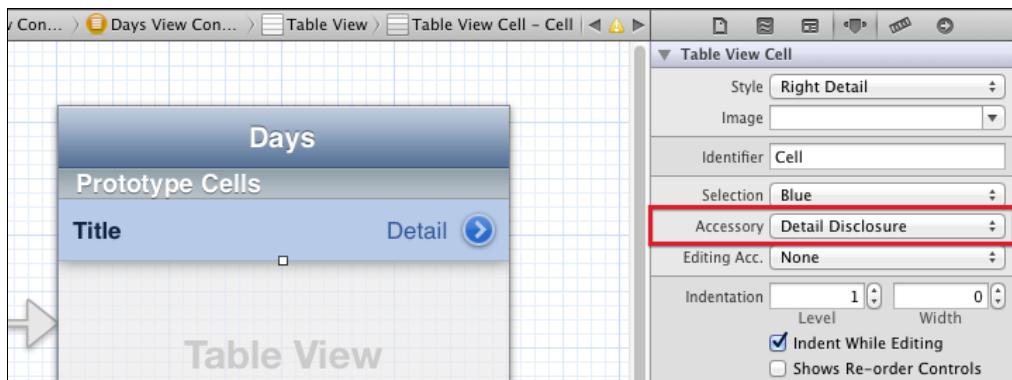
Run and build the app at this point to confirm that the data displays correctly in the table view again.

Putting the Edit screen back into action

So far, you've shuffled the various screens around a bit, but in the process severed the connection to the `MeasurementsViewController` and the Edit screen. Let's hook these screens back up again using a segue.

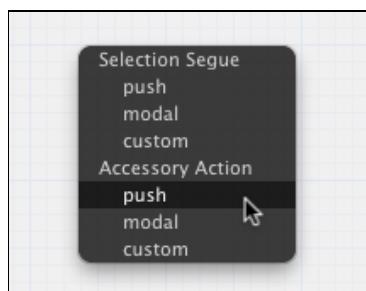
Since tapping a date row on the main screen already has a function – it shows the graph for that date – you need to find another way to perform the segue. Usually when a table cell needs to perform more than one task, the detail disclosure button is used to activate the secondary function.

In the Storyboard Editor, change the accessory of the prototype cell in the Days View Controller to Detail Disclosure. This changes the > chevron into a blue button:



In iOS 5, it wasn't possible to make a segue directly from a detail disclosure button. I'm happy to say that iOS 6 removes this limitation.

Ctrl-drag from the table view cell (anywhere in the cell is fine) to the Measurements View Controller. The popup that appears has two sections, "Selection Segue" and "Accessory Action." Choose "push" from the Accessory Action section. This hooks up the segue to the accessory button.



Change the segue identifier to "ShowRecord." In `DaysViewController.m`, add an import:

```
#import "MeasurementsViewController.h"
```

And implement `prepareForSegue:` to pass the `Record` object from the tapped row to the new view controller:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue  
    sender:(id)sender  
{  
    if ([segue.identifier isEqualToString:@"ShowRecord"])  
    {  
        NSIndexPath *indexPath = [self.tableView  
                                indexPathForCell:sender];  
  
        MeasurementsViewController *controller =  
            segue.destinationViewController;  
  
        Record *record = _records[indexPath.row];  
        controller.record = record;  
    }  
}
```

Run the app. If everything is hooked up properly, then tapping the blue disclosure detail button will open the “Measurements” screen that lists the Record for that day. You can even tap a specific value to edit it, and both the table and the graph will redraw properly when you return.



Improving the navigation

To be honest, I find the navigation a bit weird. The graph stays visible when you move to the Measurements screen. Whether or not that's something you want to happen depends on the app that you're writing, but for GumStats I don't think it's appropriate.

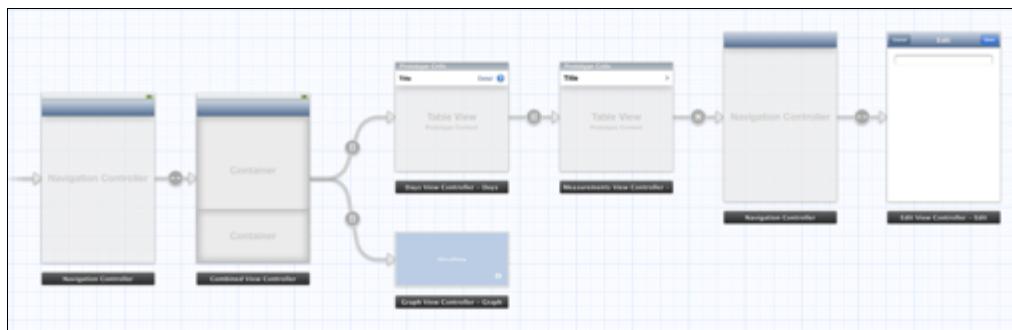
What happens is that the app is still on the `CombinedViewController`, even though it is showing the list of measurements. It has pushed a new view controller, the one that represents the Measurements screen, onto the navigation stack. But because the navigation controller is embedded inside a Container View in the main screen, only that part of the screen changes.

You're going to take the navigation controller out of the `CombinedViewController` and do it the other way around: push the `CombinedViewController` as a whole onto the navigation stack instead.

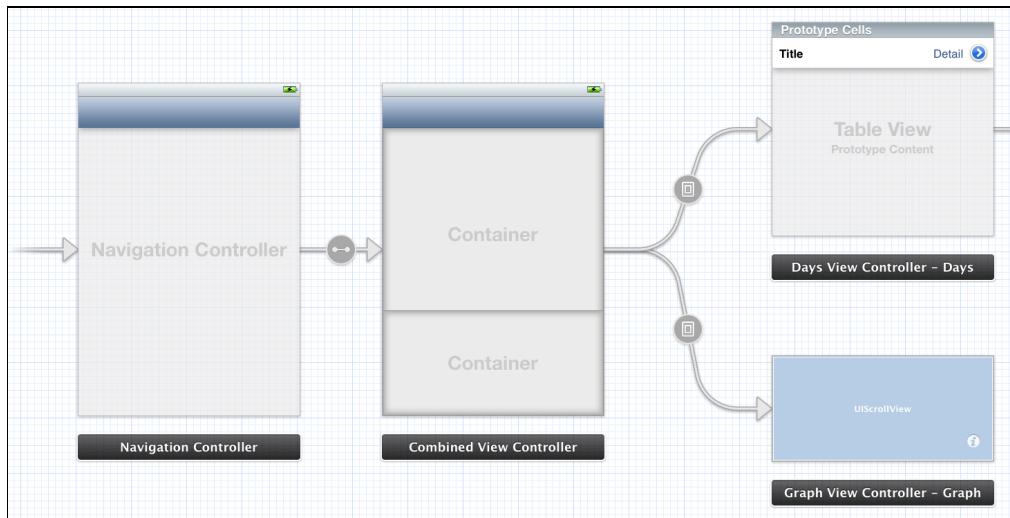
In the Storyboard Editor, select the initial view controller and choose **Editor\Embed In\Navigation Controller** from the Xcode menu. This adds a new navigation controller in front of everything else, and makes it the new initial scene.

Remove the navigation controller that sits between `CombinedViewController` and `DaysViewController`. This breaks the uppermost embed segue, so make a new one and name it "EmbedDays" like before.

The new storyboard now looks like this:



Here's a close-up of the left-hand portion:



Notice that `DaysViewController` no longer has a navigation bar showing. That's because it is no longer part of the navigation hierarchy; instead, it lives inside another view controller, `CombinedViewController`. Change the title of the `Combined View Controller` to "Days."

If you run the app now, it will crash with an exception similar to:

```
[DaysViewController topViewController]: unrecognized selector sent to instance 0x719eb20
```

The culprit is `prepareForSegue:` in **CombinedViewController.m**. It still assumes that the Days View Controller sits inside a `UINavigationController` of its own, which it no longer does. Therefore, change the method to:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"EmbedDays"])
    {
        self.daysViewController =
            segue.destinationViewController;
        self.daysViewController.records = _records;
    }
    else if ([segue.identifier isEqualToString:@"EmbedGraph"])
    {
        self.graphViewController =
            segue.destinationViewController;
    }
}
```

Run the app again. This time, it shouldn't crash. Tap on an accessory button and the Measurements screen appears full-screen again:



The segue from `DaysViewController` to `MeasurementsViewController` is a push segue, so the navigation controller replaces the current view controller on the navigation stack – the complete `CombinedViewController` – with the new one. That means both the Days and Graph View Controllers slide off the screen, because their views are part of the `CombinedViewController`. As you see, triggering a push segue works even from within an embedded view controller.

There's still one small issue: the back button in the navigation bar now says "Days" because it uses the title from the previous view controller on the stack. Originally, this back button showed the date that the user had tapped, but since `DaysViewController` isn't directly on the navigation stack anymore, that no longer happens. However, you can fake it.

The "ShowRecord" segue is still being triggered from `DaysViewController`, so it is that object's `prepareForSegue:` that gets called – after all, that's how you put the Record object into the new `MeasurementsViewController`. Add the following code to the method in `DaysViewController.m`:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue  
                      sender:(id)sender  
{
```

```
if ([segue.identifier isEqualToString:@"ShowRecord"])
{
    . . .

    UIBarButtonItem *backButton = [[UIBarButtonItem alloc]
        initWithTitle:[record dateForDisplay]
        style:UIBarButtonItemStylePlain
        target:nil
        action:nil];

    self.parentViewController.navigationItem
        .backBarButtonItem = backButton;
}
```

It's possible to create your own back button. You need to set this on the view controller that is being hidden, i.e. `CombinedViewController`. This view controller can be accessed from `DaysViewController` through the `self.parentViewController` property. Here you create a new `UIBarButtonItem` whose title is the chosen date, and set it on `CombinedViewController`.

Run the app again and verify that the back button now contains the chosen date:



Note: `GraphViewController` also still has a `prepareForSegue:` method that handles the "ShowRecord" segue. This was used way back in the beginning by the Edit bar button. You can remove this method from the code and the view controller's Navigation Item from the storyboard.

Handling low-memory situations

CombinedViewController declares two properties for its embedded view controllers, daysViewController and graphViewController. Maybe you wondered why you made these properties weak instead of strong? This has to do with handling low-memory warnings.

Apple used to recommend that developers implement the `viewDidUnload` method to release any references to `IBOutlets` as well as any cached data in the event of a low memory warning. As of iOS 6, Apple recommends the opposite. You no longer need to implement `viewDidUnload` – this method is deprecated and will not be called anymore.

The reason this is no longer called is that iOS has become better about automatically releasing the memory associated with views when they are no longer on screen. At this point, setting views to `nil` only gives you a relatively small amount of memory savings, which is not worth the effort – especially since this type of code is a notorious source of application bugs.

That doesn't mean it is now impossible to unload your views when available memory gets tight if you want to – just handle it via the `didReceiveMemoryWarning` callback. Typically, that method would look like this (add this to **CombinedViewController.m**, replacing the existing implementation):

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

    if ([self isViewLoaded] && self.view.window == nil)
    {
        NSLog(@"UNLOADING");
        self.view = nil;
    }
}
```

If the view is no longer attached to a window (think “no longer visible on the screen”), it is safe to set `self.view` to `nil` in order to unload the view. Simple enough, right?

Here's the kicker: if you decide to unload the view in `didReceiveMemoryWarning` in your own apps – and remember, Apple recommends against it – then you need to be aware of the following: when the view becomes visible again, your embedded view controllers are also reloaded!

That may come as a surprise. Normally when you set `self.view` to `nil`, only the view disappears, while the view controller itself stays in memory. Not so for embedded view controllers. ☹ When the main view gets reloaded from the

storyboard, the embedded view controllers are loaded again and there's nothing you can do to stop it.

You can verify this by sprinkling some `NSLog()` statements throughout your code. There is already one in `didReceiveMemoryWarning`. Add the following method to all the view controllers in the project:

```
- (void) dealloc
{
    NSLog(@"dealloc %@", self);
}
```

That will print a message to the debug output pane when the view controller is deallocated. Also add the following method to all view controllers (in the case of `CombinedViewController`, which already has an `initWithCoder:` implementation, just add the `NSLog` line after the call to `super`):

```
- (id) initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        NSLog(@"initWithCoder %@", self);
    }
    return self;
}
```

This prints a message when the view controller is created.

Run the app. The console should show something like this:

```
GumStats [...] initWithCoder <CombinedViewController: 0x91abc70>
GumStats [...] initWithCoder <DaysViewController: 0x71860f0>
GumStats [...] initWithCoder <GraphViewController: 0x718a100>
```

That was to be expected. First `CombinedViewController` is created, and then it loads the embedded view controllers.

If you now trigger a low-memory warning from the simulator menu (under **Hardware\Simulate Memory Warning**), the output pane should just say:

```
GumStats [...] Received memory warning.
```

It does not print the "UNLOADING" message because the current view cannot be unloaded – after all, the user is still interacting with it. So far, so good.

First tap on a date row to show the graph. Then tap the disclosure button to open the Measurements screen. The output pane says:

```
GumStats [...] initWithCoder <MeasurementsViewController: 0x7495a40>
```

That also makes perfect sense. Now once again fake a memory warning from the simulator menu. The output pane says:

```
GumStats[...] Received memory warning.  
GumStats[...] UNLOADING
```

OK, that means `CombinedViewController`'s view is now a goner. Tap the back button to return to the main screen. The output pane says:

```
GumStats[...] initWithCoder <DaysViewController: 0x7686ce0>  
GumStats[...] initWithCoder <GraphViewController: 0x7496a20>  
GumStats[...] dealloc <MeasurementsViewController: 0x7495a40>
```

Notice that the graph view is all black again and doesn't show any lines. That happens because this is not the same `GraphView` instance as before; it is a completely new object. The "initWithCoder" lines in the output pane confirm that.

You may wonder where the "dealloc" messages are for the Days and Graph View Controllers. As it turns out, these view controllers aren't automatically deallocated at all. The `CombinedViewController` still holds onto them. So here you have a paradoxical instance of the handling of a low-memory situation actually causing a memory leak!



To resolve the above, you'll have to do something like this in your `didReceiveMemoryWarning` implementation for **CombinedViewController.m**:

```
- (void) didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
  
    if ([self isViewLoaded] && self.view.window == nil)  
    {  
        self.view = nil;  
  
        [self.daysViewController  
         willMoveToParentViewController:nil];
```

```
[self.daysViewController removeFromParentViewController];  
  
[self.graphViewController  
    willMoveToParentViewController:nil];  
[self.graphViewController removeFromParentViewController];  
}  
}
```

You have to manually use view controller containment methods to remove the embedded view controllers from their parent. With this change in place, the “dealloc” `NSLog()` messages will appear in the output pane when you trigger the low-memory warning from the Measurements screen.

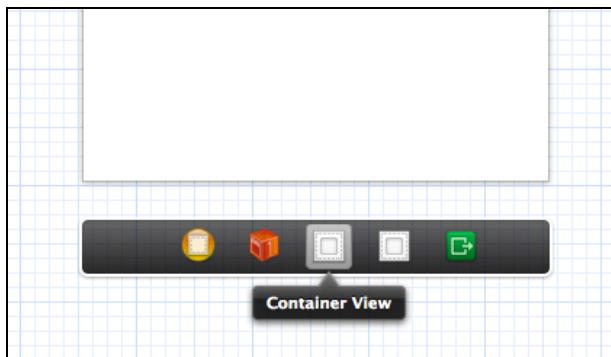
Note: Be aware of this – if you unload a view that has embedded view controllers in it, then those view controllers are destroyed as well, and reloaded when the main view reloads. With that in mind, you probably don’t want to unload that main view in low-memory situations after all!

Limitations of Container View

The Container View is a handy feature that lets you quickly embed one or more view controllers into another. However, it also has its limitations.

You probably shouldn't consider the Container View and embed segues a visual replacement for the `addChildViewController:` API. They solve the problem of dividing up the screen into different sections that are each handled by their own view controller. No more, no less. For more advanced custom containers, such as a tab-bar or navigation controller replacement, you're better off writing code.

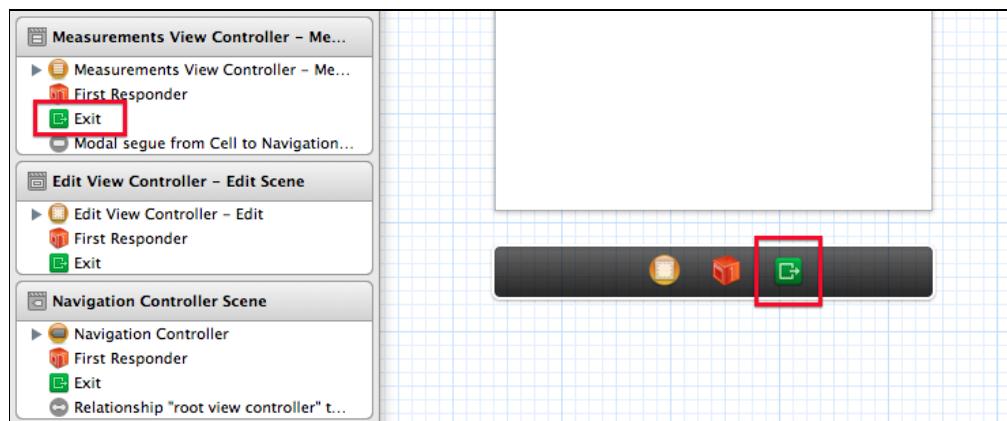
Tip: It is possible to add one or more Container Views directly to the Dock area below the scene, so that they aren't actually visible in the main view:



Such an embedded view controller works just like the ones that you drop into the view. It is still loaded along with the parent view controller, and `prepareForSegue:` is called on it. But the embedded controller won't become visible until you manually add its view into the view hierarchy.

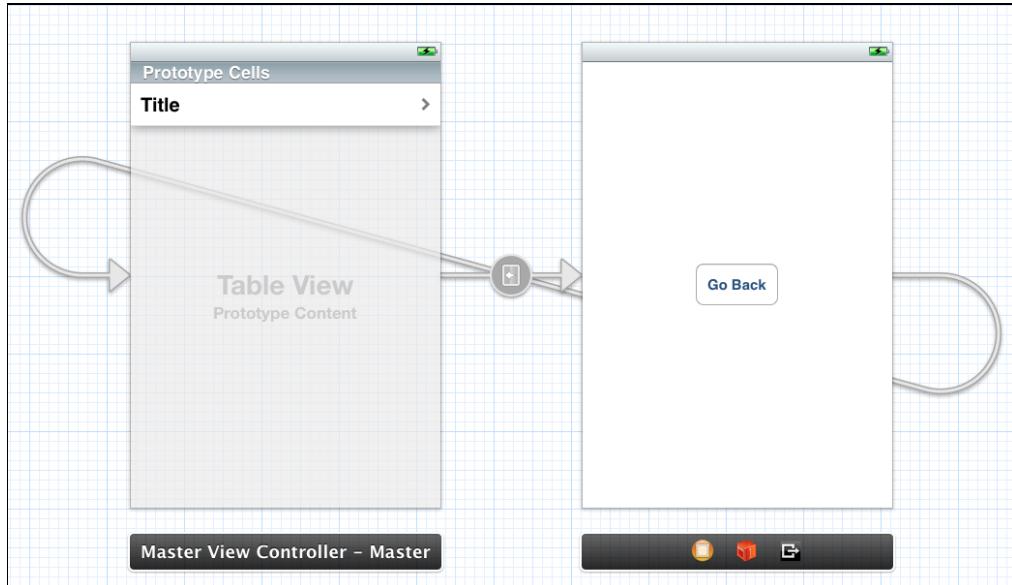
Time to unwind

Speaking of the Dock area below a scene, you might have noticed that in iOS 6 there's a new icon there – that we refer to by the very official technical term “the Exit thingy.” However, if you try playing around with it you'll notice it doesn't seem to do anything. You cannot make connections to it and it has no attributes – what gives?



The Exit icon is for creating *unwind segues*, which let you go back to a screen that you came from.

Sometimes people new to storyboards try to make a segue back to the previous screen, which seems like a reasonable thing to do. Except that it doesn't quite work that way... A segue will always create a new instance of the view controller that it points to, and eventually you will run out of memory. So don't do something like this:



Up until now, typical solution for returning from a view controller to the previous one was to use a delegate protocol. That is exactly what happens in `EditViewController` in the GumStats app. It declares an `EditViewControllerDelegate` protocol with two required methods:

```
@protocol EditViewControllerDelegate <NSObject>

- (void)editViewControllerDidCancel:(EditViewController *)controller;

- (void)editViewController:(EditViewController *)controller
didChangeValue:(int)newValue;

@end
```

`EditViewController` object itself has a property for this delegate:

```
@property (nonatomic, weak) id <EditViewControllerDelegate>
delegate;
```

Inside the action methods that are hooked up to its Cancel and Done buttons, the view controller notifies the delegate:

```
- (IBAction)cancel:(id)sender
{
    [self.delegate editViewControllerDidCancel:self];
}

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

```
{  
    int newValue = [self.textField.text intValue];  
    [self.delegate editViewController:self  
        didChangeValue:newValue];  
}
```

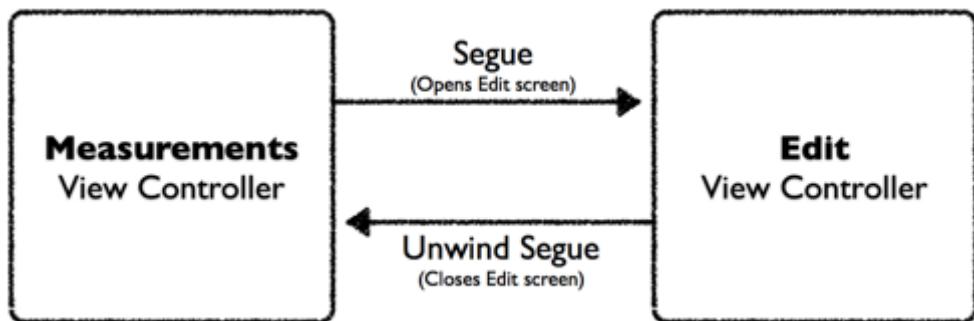
The delegate in this case is MeasurementsViewController, because that is the screen that launches EditViewController. It implements these delegate methods as follows:

```
#pragma mark - EditViewControllerDelegate  
  
- (void)editViewControllerDidCancel:(EditViewController *)  
    controller  
{  
    [self dismissViewControllerAnimated:YES completion:nil];  
}  
  
- (void)editViewController:(EditViewController *)controller  
    didChangeValue:(int)newValue  
{  
    NSIndexPath *indexPath = [self.tableView  
        indexPathForSelectedRow];  
  
    [self.record replaceValue:newValue atIndex:indexPath.row];  
    [self.tableView reloadData];  
  
    [self dismissViewControllerAnimated:YES completion:nil];  
}
```

In both cases, it dismisses the Edit View Controller. In the `editViewController:didChangeValue:` delegate method, MeasurementsViewController also changes the data model and updates the screen with the new value provided by the user.

This approach works pretty well – using delegates in iOS code is a time-honored tradition – but it's also a lot of work. And you have to repeat it over and over for every view controller that you want to have send data back when it's closed.

As of iOS 6, your life gets a lot easier. Now you can replace such delegate protocols with an unwind segue. Think of this as a segue that goes the other way:



Unfortunately, using unwind segues is a little less obvious than using regular “forward” segues. You cannot simply Ctrl-drag between the two view controllers. Instead, you have to add a special *unwind action method* to the code before the Exit icon becomes active, so you can hook it up.

Goodbye delegate, hello unwind segue

Remove the `cancel:` and `done:` methods from **EditViewController.m**. These are `IBActions` that are connected to the Cancel and Done bar buttons at the top of the Edit screen. You won’t need them anymore.

Also remove the declaration of the `EditViewControllerDelegate` protocol from **EditViewController.h**, as well as the delegate property. It should now look like this:

```

@interface EditViewController : UIViewController

@property (nonatomic, assign) int value;

@end
  
```

Because **MeasurementsViewController** still uses this protocol, you also have to make some changes there. Remove the `<EditViewControllerDelegate>` bit from the class extension in **MeasurementsViewController.m**, and in `prepareForSegue:` remove the line:

```
controller.delegate = self;
```

Now the app should build again, but obviously the Edit screen will no longer close when you tap the Cancel or Done buttons. (And the app will in fact crash when you try.)

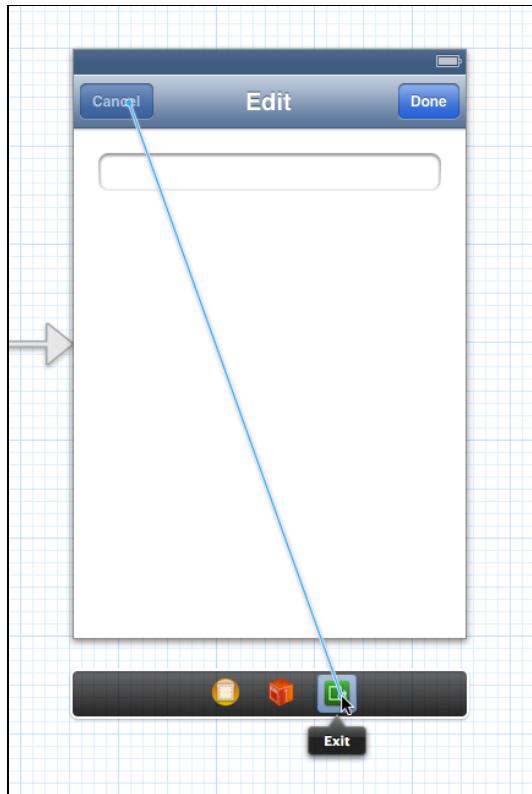
Still in **MeasurementsViewController.m**, add the following code:

```

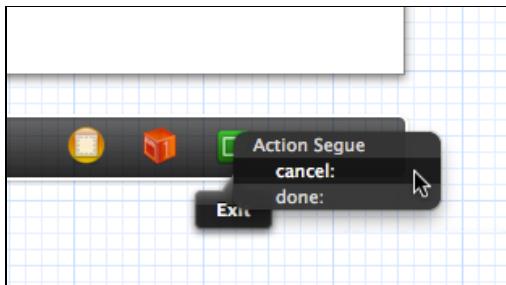
- (IBAction)cancel:(UIStoryboardSegue *)segue
{
}
  
```

```
- (IBAction)done:(UIStoryboardSegue *)segue
{
}
```

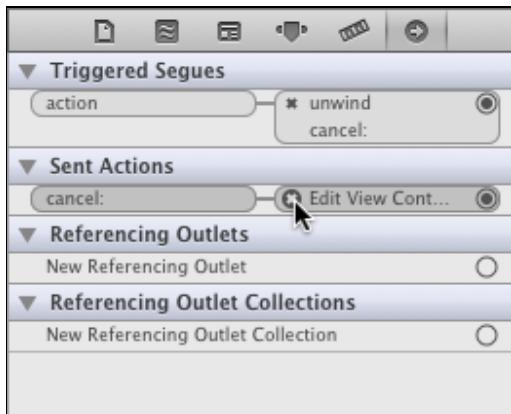
Go back to the storyboard and Ctrl-drag from the Edit View Controller's Cancel button to the Exit icon in the dock below the scene:



When you let go of the mouse button, a small popup appears:



Choose "cancel:" to connect the button. In the Connections Inspector for the Cancel button, under Triggered Segues, you can see that the button is now connected to the unwind action named "cancel":



Important: Previously the Cancel button was connected to the regular `IBAction cancel: on EditViewController`. This connection is shown under “Sent Actions.” Because you have removed this method from the code, you should also break this connection by clicking the little X button.

Repeat the same procedure for the Done button to connect it to the “done:” unwind action on the Exit icon. (And remember to remove the old “Sent Actions” connection as above for the “Cancel” button.)

Run the app again and navigate to the Edit screen. If you now tap the Cancel or Done buttons, the screen closes as before even though you didn’t write any code to call `dismissViewControllerAnimated:completion:` on `EditViewController`. The unwind segue does that for you automatically – nice, eh?

Let’s recap what you did here.

You first removed the delegate stuff because you no longer need it. The Cancel and Done buttons used to be connected to regular `IBAction` methods on `EditViewController`, but all those methods did was call the delegate. Since there is no longer a delegate, you also disconnected these actions and removed their methods.

The only things you added were the unwind action methods in `MeasurementsViewController`. An unwind action method always has the special form:

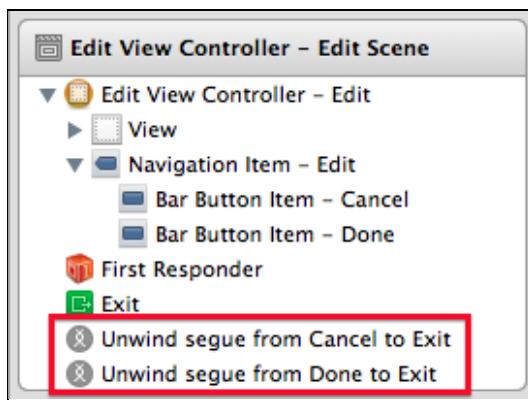
```
- (IBAction)methodName:(UIStoryboardSegue *)segue
{
}
```

It is still marked as `IBAction`, so the Storyboard Editor can find this method in your code, but it always takes a `UIStoryboardSegue` object as its only parameter.

The Exit icon for each scene doesn’t do anything until you have defined at least one unwind action method somewhere. Once you have one or more unwind actions

defined, the Action Segue popup lists all the available unwind action methods, wherever they may appear in your view controllers.

With the connection of the Cancel and Done buttons to these special unwind action methods, the unwinding mechanism is able to do its job. Unfortunately, at the time of writing this chapter there is no arrow in the storyboard that visually represents such unwind segues, so they are a little less obvious to use than regular segues. They do appear in the Document Outline on the left of the storyboard window, though:



These segues have attributes in the Attributes Inspector (an identifier and the name of the unwind action method it is connected to), but show nothing in the Connections Inspector. The Exit icon itself does not have any attributes, but it does show the Connections that are made to it:



Note: At this point you may be wondering why you had to move the `done:` and `cancel:` methods into the previous view controller in the chain. Wouldn't it be easier if `EditViewController` could handle the unwind actions itself? The problem is that this no longer gives you a way to send data back to that previous view controller (which is the topic of the next section).

It is true that not all view controllers need to pass data back to their callers. Maybe you have a modal screen that directly makes changes to your Core Data store and it just needs to dismiss itself when the user taps Cancel or Done. In that case there is no data to send back.

However, such an approach doesn't seem to work well with unwind segues. The "source" view controller, in other words the one that triggers the unwind segue, is not supposed to handle its own unwind actions. Instead, you can

simply hook up your controls to a local action method and dismiss/pop yourself from there.

You only need to use an unwind segue when you want some sort of communication to happen between the two view controllers involved.

Passing data back to the caller

Maybe you've noticed that there is now something missing. If you modify a value in the Edit screen, the new value isn't passed back to `MeasurementsViewController`. That makes sense, because you removed the code that does this. There used to be a special delegate method for it, but that no longer exists and the new `done:` unwind action doesn't do anything yet.

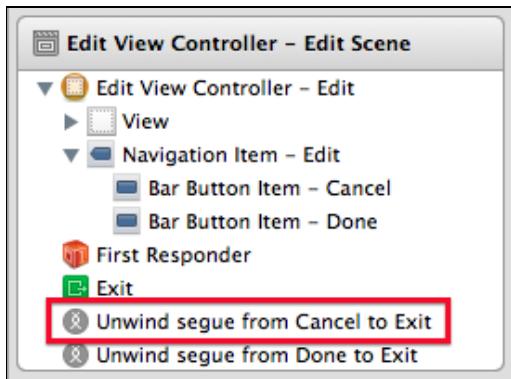
In order to know what to do here, it's important to understand the sequence of events that happens when the user taps the Cancel or Done buttons.

- 1. Which view controller is the destination?** First, UIKit determines which view controller will be the destination of the unwind segue. This is the view controller that has the unwind actions. You may think that this is obvious – after all, you added those actions to `MeasurementsViewController`, so that should be the destination of the segue – but it can get more complicated than that. More about this later.
- 2. Call `prepareForSegue: on the source`.** Once the destination is found, UIKit calls `prepareForSegue:` on the source view controller. The "source" is the view controller that is about to close, which in your case is `EditViewController`. This is the same `prepareForSegue:` that gets called for regular segues and embed segues. Here you can do whatever you need to do to prepare the data you want to send back. Typically you would put it inside a property that's visible to other objects.
- 3. Call the unwind action on the destination.** The unwind action is called on the destination view controller. Here you'd look at that property on the source and do what you need to do with it.
- 4. Perform the segue.** Finally, the actual segue is performed. This just animates the source view controller out of view and the destination controller into view.

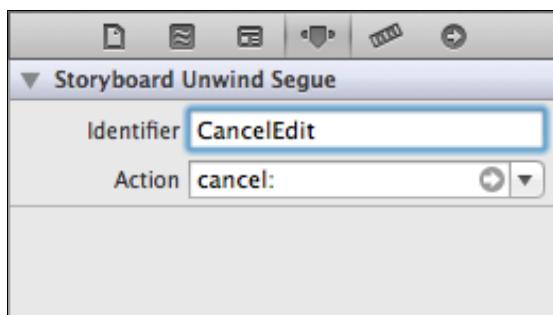
This means you need to write two more pieces of code, `prepareForSegue:` on the Edit View Controller, and the `done:` unwind action on the Measurements View Controller.

Before you get to that, it's a good idea to give unique identifiers to the unwind segues so you can tell them apart. Usually you want to do something different for the Cancel segue than for the Done segue, and giving them identifiers lets you distinguish between them.

In the Storyboard Editor, select the "Unwind Segue from Cancel to Exit" item from the Document Outline on the left:



In the Attributes Inspector, give it the identifier "CancelEdit." Do likewise for the other unwind segue, but name it "DoneEdit."



Now add the `prepareForSegue:` method to **EditViewController.m**:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"DoneEdit"])
    {
        self.value = [self.textField.text intValue];
    }
}
```

This takes the text that is currently in the text field and places it in the `self.value` property. That was step one.

Now replace the `done:` action in **MeasurementsViewController.m** with:

```
- (IBAction)done:(UIStoryboardSegue *)segue
{
    EditViewController *controller = segue.sourceViewController;

    NSIndexPath *indexPath = [self.tableView
        indexPathForSelectedRow];

    [self.record replaceValue:controller.value
```

```
atIndex:indexPath.row];  
  
[self.tableView reloadData];  
}
```

Remember, this method is invoked after `prepareForSegue:`, but before the segue animation actually happens. It can look at the segue's `sourceViewController` property to determine where the segue is coming from. In this case, it's `EditViewController`, of course, and the method looks at its `value` property to get the value the user entered.

Note: The other segue properties you can look at are `identifier` and `destinationViewController`. You don't use them here, but if you have one unwind action handling multiple segues, it can be useful to distinguish between them by examining the value of `segue.identifier`.

Now if you run the app again, the Edit screen works as before, but with a lot less code. Cool!



Let's think about the code you didn't have to write. You no longer have to make your own delegate protocol, make the first view controller implement that protocol, and hook up the two. You don't need to call `dismissViewControllerAnimated:completion:`, or `popViewControllerAnimated:` if you're in a navigation stack – all of that is taken care of for you by the unwinding mechanism.

The only thing you need to worry about now is moving data back to the previous view controller and handling it in the unwind action method.

Note: You will probably use unwind segues most with modally-presented view controllers, like you did above. For controllers that are pushed on a navigation stack, there is less of a need for them because the back button will already take care of going back to the previous screen. (It does not use unwind segues for that, by the way.)

If you need to jump back more than one level in the navigation stack, then it does make sense to create an unwind segue. You'll learn more about unwinding across multiple view controllers later on.

Cancelling segues

It used to be that once a segue got started, there was no way to stop it. iOS 6 adds a new feature that lets you determine, just before a segue happens, whether it should really continue. Segues can now change their minds!

You can use this feature to perform input validation in the Edit screen. Maybe you've noticed that you can currently type all kinds of invalid data into the text field (especially on the simulator, where you can type regular text into the box even though only the number pad is visible).

Because the code does `[self.textField.text intValue]`, anything that is not a number gets converted to the value 0. But what about numbers that are too large? The data model assumes the maximum value is 100, but there is nothing stopping the user from typing in larger values. (Is there a Guinness record for number of gumballs purchased from a single machine in one hour?) You can guard against this and pop up an alert view when the user types in an invalid value.

The following method can prevent a segue from being triggered:

```
- (BOOL)shouldPerformSegueWithIdentifier:(NSString *)identifier  
                                sender:(id)sender
```

UIKit attempts to call this method before `prepareForSegue:`. If it returns `NO`, the segue is cancelled. Try it out. Add this to **EditViewController.m**:

```
- (BOOL)shouldPerformSegueWithIdentifier:(NSString *)identifier  
                                sender:(id)sender  
{  
    return NO;  
}
```

Now run the app. Neither the Cancel nor Done button will appear to function anymore. Of course, that's a little too extreme, so change the method to:

```
- (BOOL)shouldPerformSegueWithIdentifier:(NSString *)identifier  
                                sender:(id)sender  
{  
    if ([identifier isEqualToString:@"DoneEdit"])  
    {  
        if ([self.textField.text length] > 0)  
        {  
            int value = [self.textField.text intValue];  
        }  
    }  
}
```

```
        if (value >= 0 && value <= 100)
            return YES;
    }

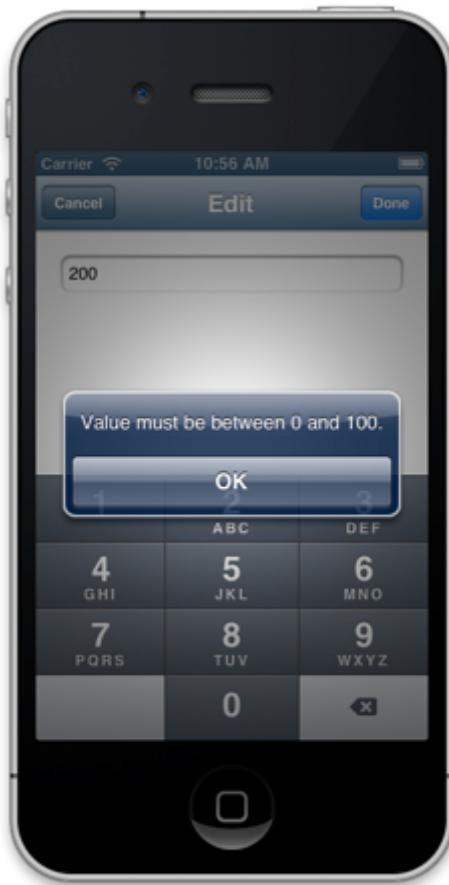
    [[[UIAlertView alloc]
        initWithTitle:@"nil"
        message:@"Value must be between 0 and 100."
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil]
    show];

    return NO;
}
return YES;
}
```

First it looks at the segue's identifier. The validation checks are only done for the "DoneEdit" segue. If the user taps the Cancel button, you don't care what they typed.

The current validation logic simply checks that the value is between 0 and 100. If it isn't, the app shows a `UIAlertView` and returns `NO` from the method to prevent the segue from happening.

Try it out. Type a value that is larger than 100 and tap Done. You should now get an alert:



Note: The `shouldPerformSegueWithIdentifier:sender:` method isn't just for unwind segues, it works for any kind of segue.

Unwinding as far as you like

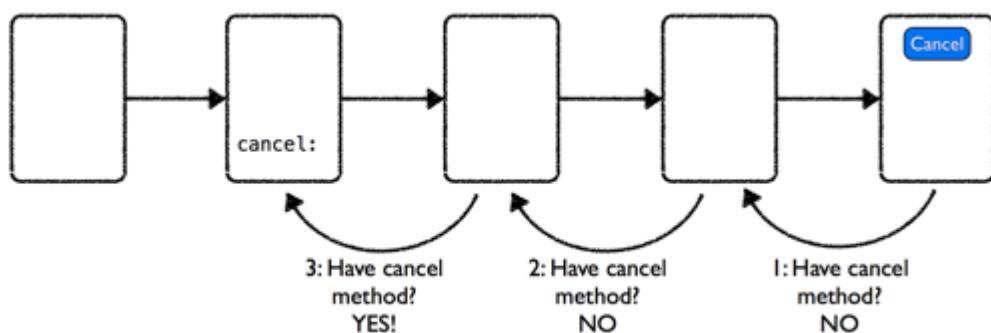
The unwind segues you've seen so far just went back to the previous view controller, but it is also possible to jump further back. It's not a requirement that the unwind action methods live in the parent of the view controller that you're trying to close. You can jump back to any point on the chain of view controllers.

For fun – but not necessarily good user interface design – make the Cancel button jump back all the way to the first screen. To pull this off, you don't need to make any new connections in the Storyboard Editor. All you need to do is move the `cancel: unwind` action from `MeasurementsViewController` to `CombinedViewController`.

Cut the `cancel:` method out of **MeasurementsViewController.m** and paste it in **CombinedViewController.m**. Now run the app again. Go to the Edit screen and tap the Cancel button. You no longer end up back at the Measurements screen, but on the main screen of the app.

You might think that's odd because you never changed anything in the Storyboard Editor to make this happen. Here's the trick: when you connect a button or other control to the Exit icon, you don't actually connect it to a specific view controller but only to the name of the unwind action method, in this case, "cancel:". During runtime, when the user taps that button, UIKit will go through all the active view controllers to look for a method with that name that takes a `UIStoryboardSegue` parameter.

It does this in a specific order, usually from top to bottom, so the most-recently-used view controller is asked first. In your app that is `MeasurementsViewController`. But since that object no longer has the `cancel:` method, UIKit will ask the previous controller on the stack and then the one before that, and so on, until it finds a view controller that will handle the action method.



Now put the `cancel:` method back into `MeasurementsViewController`, but also leave it in `CombinedViewController`. Tapping the Cancel button now brings you back to the Measurements screen again. UIKit decided to use `MeasurementsViewController`'s unwind method rather than `CombinedViewController`'s, because it is "closer" to the view controller that is closing.

Note: If no matching unwind method is found, then the segue simply doesn't happen. The app won't crash or give any error messages. You can even connect to unwind actions that are not in your view controller chain, although that has no effect whatsoever. For some reason, the Storyboard Editor also lets you Ctrl-drag from controls in one view controller to the Exit icon in another view controller. None of this matters because the action methods are only looked up during runtime, and if no match is found the segue is simply ignored.

Manually unwinding segues

You may already know that it's possible to have a regular "forward" segue that isn't connected to a control but to the view controller itself, that you can trigger manually using `performSegueWithIdentifier:`. The same thing is possible for unwind segues.

Let's demonstrate this on the Edit screen. You'll add a Delete button that first pops up an action sheet. If the user confirms the action, an unwind segue is performed that asks the receiver – in this case `MeasurementsViewController` – to delete this particular value from the list.

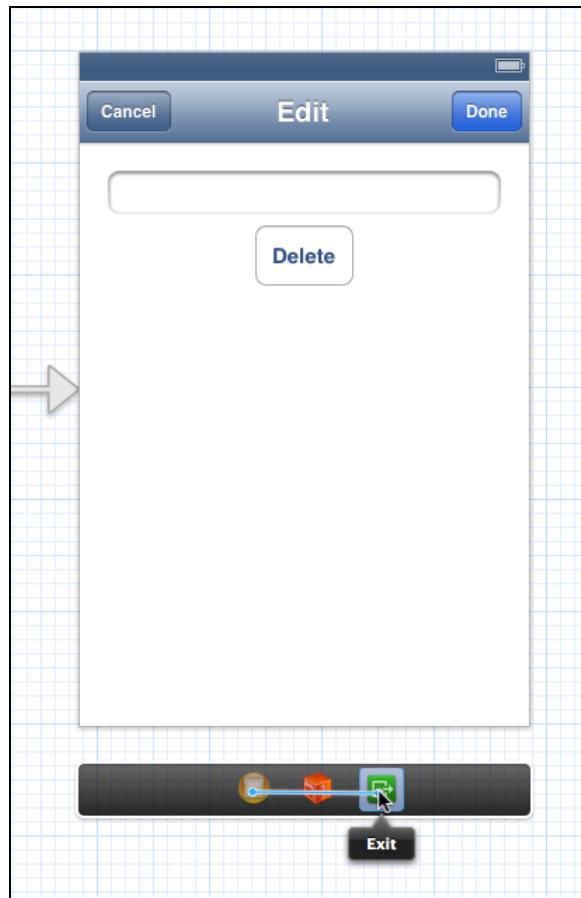
First add the unwind action method to **MeasurementsViewController.m**:

```
- (IBAction)deleteValue:(UIStoryboardSegue *)segue
{
    NSIndexPath *indexPath = [self.tableView
        indexPathForSelectedRow];

    [self.record deleteValueAtIndex:indexPath.row];
    [self.tableView deleteRowsAtIndexPaths:@[indexPath]
        withRowAnimation:UITableViewRowAnimationFade];
}
```

This is pretty simple. At this point the table view still knows which row is selected, so you can simply delete that row from both the data model and the table view. (If you need more information from the source view controller, such as which item to delete, then you can look at the segue's `identifier` or `sourceViewController` properties, the way you did before.)

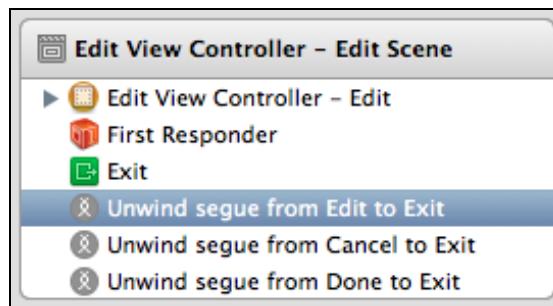
In the Storyboard Editor, add a new `UIButton` to the Edit scene and label it "Delete." Then Ctrl-drag from the view controller icon in the dock to the Exit icon. (Note that you're dragging from the view controller to the Exit icon, not from the Delete button you added.)



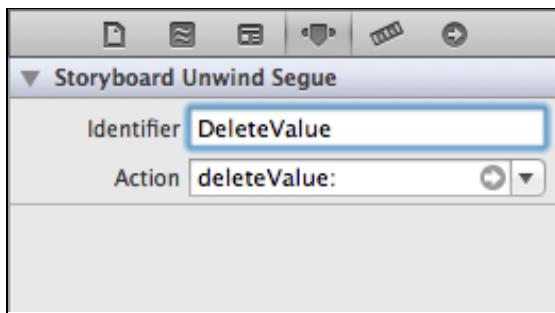
This creates a so-called “manual segue.” It’s called that because it isn’t hooked up to any user interface controls.

Manual Segue
cancel:
deleteValue:
done:

Choose the “deleteValue:” action. The new segue will appear in the Document Outline alongside the others as “Unwind Segue from Edit to Exit.” Here “Edit” refers to the name of the view controller, of course.



Select the segue and set its identifier (in the Attributes Inspector) to "DeleteValue":



Because this segue doesn't happen automatically, you have to write some code to trigger it. Add the following to **EditViewController.m**. First change the class extension to:

```
@interface EditViewController () <UIActionSheetDelegate>
@property (nonatomic, weak) IBOutlet UITextField *textField;
@property (nonatomic, weak) IBOutlet UIButton *deleteButton;
@end
```

This lets the compiler know this class can now act as a delegate for `UIActionSheet`. It also makes a property for the Delete button.

Then add these methods:

```
- (IBAction)delete:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Really delete?"
        delegate:self
        cancelButtonTitle:@"Cancel"
        destructiveButtonTitle:@"Delete"
        otherButtonTitles:nil];

    [actionSheet showFromRect:self.deleteButton.frame
        inView:self.view animated:YES];
}

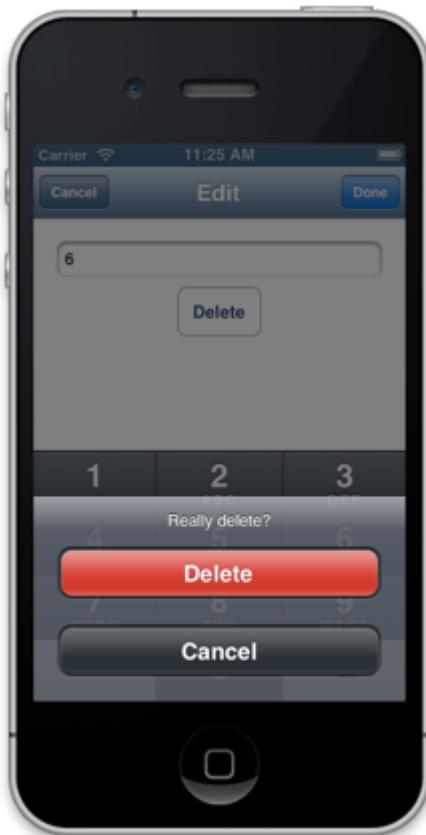
#pragma mark - UIActionSheetDelegate

- (void)actionSheet:(UIActionSheet *)actionSheet
    didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    if (buttonIndex != actionSheet.cancelButtonIndex)
    {
        [self performSegueWithIdentifier:@"DeleteValue"
            sender:nil];
    }
}
```

```
}
```

Go back to the storyboard once more and connect the Delete button to the new `deleteButton` outlet and the `delete:` action method. That should do it.

Now when you run the app, tapping Delete pops up the action sheet:

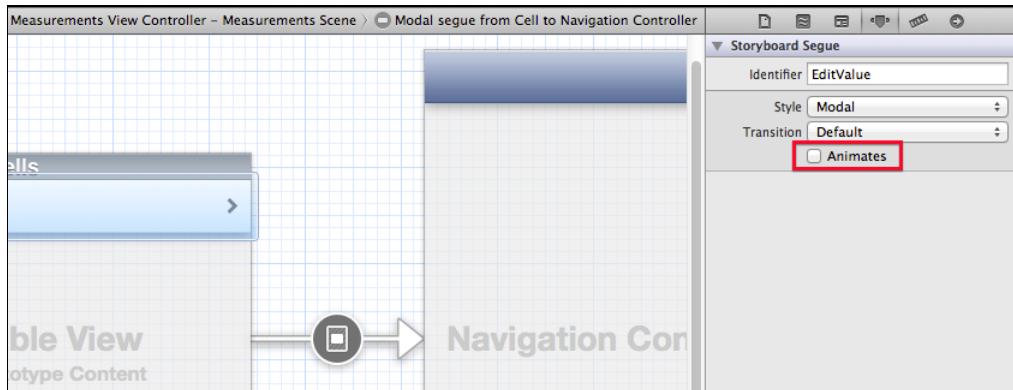


When the user confirms the deletion, the action sheet delegate method will call `performSegueWithIdentifier:@"DeleteValue"`, which will first call the unwind method `deleteValue: ON MeasurementsViewController`, and then trigger the segue to close the screen.

Seg it your way

Now the fun part. When you create a segue between two view controllers, you have the choice between push, modal and custom segues. “Push” only works in a navigation controller, “modal” shifts the new view controller up from the bottom of the screen, but “custom” lets you do your own animations. With a little bit of effort, it is also possible to use custom segues for unwinding.

Also, with iOS 6 it is now also possible to do a modal segue without the slide-up animation. Try this on the Edit screen. Select the segue between the Measurements screen and the Navigation Controller, and go to the Attributes Inspector.



Uncheck the Animates box. Now run the app again. Notice that there is no nice animation anymore when you tap on a row to edit it. The Edit screen is suddenly there (although you'll still see the keyboard animate into view).

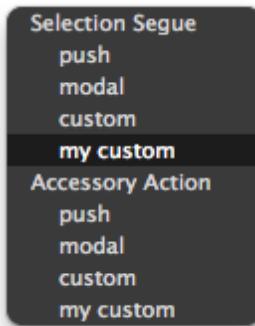
Also notice that when you close the screen, it slides off with an animation as usual. Turning off animation on a modal segue only works one way, not when unwinding.

Note: Wondering why this would ever be useful? Well, modal segues without animation are great for when you want to show a login screen as the first thing the user sees. You make a new view controller for that login screen and connect it to your main view controller with a modal segue that has animations turned off. Then in your main view controller's `viewWillAppear:`, you perform that segue manually. As soon as the app starts, the user immediately sees the login screen.

Now you'll try out a custom segue instead. To create one, you need a new class that extends `UIStoryboardSegue` and overrides its `perform` method. You'll begin by making the simplest possible custom segue – it will look just like the modal one without animation – and then you'll extend that to do a cool animation.

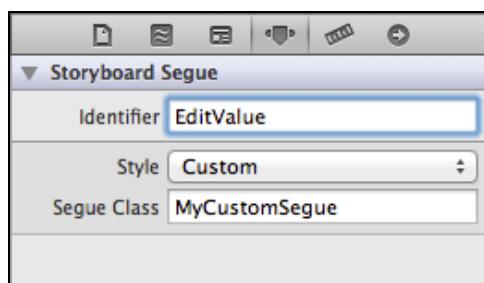
Add a new file to the project using the Objective-C class template. Name it **MyCustomSegue**, subclassing `UIStoryboardSegue`.

Go into the Storyboard Editor and remove the outgoing modal segue on the Measurements View Controller. Select the table view cell and Ctrl-drag to the navigation controller on the left to make a new segue. The following popup will appear:



This popup now contains the name of your custom segue class. Cool, huh? The Storyboard Editor picks this up automatically from the classes you have added to the project. Make sure to select the one from the “Selection Segue” section so that it’s triggered when the user taps the table view cell (and not the accessory).

The Attributes Inspector for the segue now looks like this:



Don't forget to enter "EditValue" for the identifier, or `prepareForSegue:` will no longer function.

Put the following in **MyCustomSegue.m**:

```
- (void)perform
{
    UIViewController *source = self.sourceViewController;
    UIViewController *destination =
        self.destinationViewController;

    [source presentViewController:destination animated:NO
                      completion:nil];
}
```

This is the simplest possible custom segue that you can write. It tells the source view controller to modally present the destination view controller without an animation.

Try it out. Run the app and open the Edit screen. It should appear suddenly, exactly the way it did when the segue was modal with the `Animates` attribute turned off. You know now that it works, which is good, but such abrupt transitions are very

disconcerting for the user. Now you'll replace it with something that is easier on the eye.

First, go into **EditViewController.m** and remove the following line from `viewDidLoad`:

```
[self.textField becomeFirstResponder];
```

This line caused the keyboard to animate into view, but that will conflict with the new animation from the segue, so it's better to turn it off.

Now replace the **MyCustomSegue.m** file in the project with the one from this chapter's resources. I won't explain all the code here since it is well commented, but I will quickly highlight what it does. As before, everything happens in the `perform` method. This method takes the views from the source and destination view controllers and animates them. However, there is a little trick involved.

The source view controller in this case is `MeasurementsViewController`, and the segue animation will slide it off the screen while `EditViewController` slides onto the screen. But it can't use the views of these two controllers directly. After all, they are both embedded in their own `UINavigationController`. If you were to animate these views, the navigation bar wouldn't animate along with them, because it belongs to the parent view controller.

What you need to do first is check if there is a `parentViewController` and then use the parent's view. The segue does that with the following method:

```
- (UIView *)findTopMostViewForViewController:
    (UIViewController *)viewController
{
    UIView *theView = viewController.view;
    UIViewController *parentViewController =
        viewController.parentViewController;

    while (parentViewController != nil)
    {
        theView = parentViewController.view;
        parentViewController =
            parentViewController.parentViewController;
    }
    return theView;
}
```

It is called like this from `perform`:

```
- (void)perform
{
    UIViewController *source = self.sourceViewController;
```

```
UIViewController *destination =
    self.destinationViewController;

UIView *sourceView = [self
    findTopMostViewForViewController:source];
UIView *destinationView = [self
    findTopMostViewForViewController:destination];

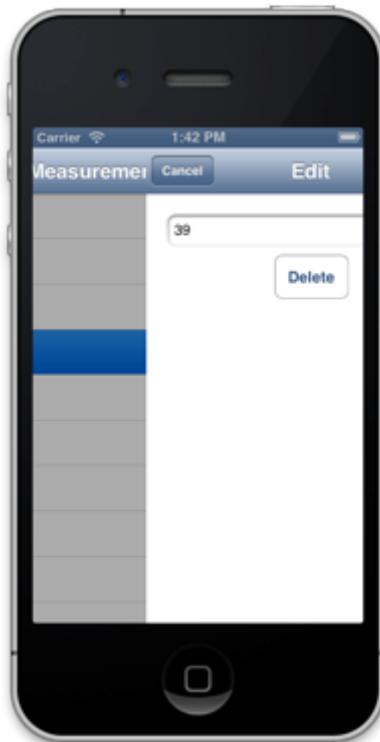
. . .

}
```

Then the animation uses the views from the `sourceView` and `destinationView` variables. The animation itself is simply a matter of positioning `sourceView` and `destinationView` on the screen, and then moving them to new coordinates.

To make the animation a bit more visually interesting, a completely black view is added on top of the source view. Initially it is totally transparent. As the animation progresses, the opacity is gradually increased so that it becomes darker and darker, which makes the source view appear to recede into the background.

The animation looks something like this:



If the animation goes too fast for you to see clearly, you can either increase the duration (it is currently 0.5 seconds) or enable slow animations from the simulator's

menu bar, under **Debug\Toggle Slow Animations**. You can also toggle slow animations by tapping the Shift key quickly three times in succession.

When the animation block completes, perform calls `presentViewController:animated:completion:` with the `animated` parameter set to `NO` in order to properly present the new view controller. Until that happens, all the segue has done is put the new view on the screen and move it around, but the new view controller itself is unaware of any of this happening. The destination view controller must always be properly presented for the app to function correctly.

Cool, you have your own segue animation! There's just one problem to fix, and that is the keyboard. You can't make the text field become the first responder in `viewDidLoad` anymore. Because the segue accesses the destination view controller's `view` property, `viewDidLoad` will be called just before the segue animation starts. This appears to conflict with the keyboard animation, and actually manages to crash the app under certain circumstances.

Here is a workaround for `EditViewController`:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    if (self.presentingViewController != nil)
    {
        [self.textField becomeFirstResponder];
    }
}
```

`viewDidAppear:` is actually called twice for `EditviewController`. Because the segue temporarily puts the view into the source view for the animation, UIKit calls `viewDidAppear:` to let the view controller know about this. But when the segue calls `presentViewController` after the animation completes, UIKit calls `viewDidAppear:` again.

It is not safe to show the keyboard on the first call to `viewDidAppear:`, but it is fine the second time around, because at that point the segue has finished its animation. You can check for this by looking at the `self.presentingViewController` property, which is `nil` before but non-`nil` after the call to `presentViewController`.

Tip: An alternative approach to animating segues is to create a `UIImage` snapshot of the contents of the views that you wish to animate, then put these images into `UIImageViews`, and perform the animation on the image views instead of on the actual views. *iOS 5 by Tutorials* includes an example of how to do this in Chapter 5, "Intermediate Storyboards".

Unwind it your way

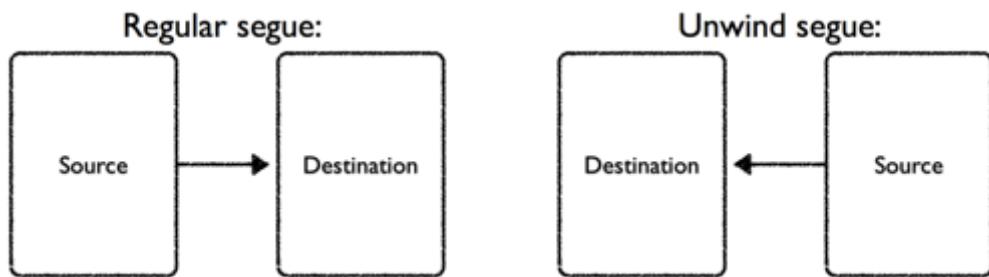
The app now has a cool animation for going to the Edit screen, but when you close that screen it still does the default slide-down-the-bottom animation. Unfortunately, unlike regular segues, unwind segues do not have a `Style` attribute that lets you choose a custom `UIStoryboardSegue` subclass from the Attributes Inspector. Instead, you have to write some code.

First add the code for the most basic unwind segue class possible. As before, add a new file to the project with the Objective-C class template. Name it **MyUnwindSegue** and make it a subclass of `UIStoryboardSegue`.

Add the following `perform` method to **MyUnwindSegue.m**:

```
- (void)perform
{
    [self.destinationViewController
        dismissViewControllerAnimated:NO completion:nil];
}
```

You can't get much simpler than that! It just dismisses the view controller. Notice that `destinationViewController` here is the one that is being returned to while `sourceViewController` is the one that closes, i.e. the one that triggered the segue. Imagine an arrow going from the view controller that closes to the one that handles the unwind action:



So how do you tell your app to use that custom unwind segue? You need to implement the following method-with-a-very-long-name:

```
- (UIStoryboardSegue *)segueForUnwindingToViewController:
    (UIViewController *)toViewController
    fromViewController:(UIViewController *)fromViewController
    identifier:(NSString *)identifier
```

The job of this method is to return a `UIStoryboardSegue` object that can perform the segue with the specified identifier. Now the big question is, in which class does this method go?

If you answered with `MeasurementsViewController`, then I can understand where you're coming from – it sounds like a good place for it, since that is also where the

unwind action methods live. Unfortunately, it's the wrong answer. That leaves one other view controller in the navigation stack, `CombinedViewController`, but that is also not the one.

In order to understand how this works, take a closer look at the sequence of events that take place when you perform an unwind action.

I've mentioned before that when you connect a button or other control to the Exit icon, you're not really connecting it to a method on any specific view controller. Instead, during runtime UIKit searches the chain of view controllers for one that implements the action method with that name.

Here is the important thing to remember: that search starts at the **container** view controller, if there is one. If the source view controller – the one that triggers the unwind segue – sits inside a container controller such as a navigation controller, a tab bar controller, or your own custom container class, then UIKit considers it the responsibility of that container to a) find the view controller that will handle the unwind action, and b) create the unwind segue object.

In this app, both `MeasurementsViewController` and `CombinedViewController` sit inside a `UINavigationController`, and therefore it is the navigation controller that gets this `segueForUnwindingToViewController` callback. That may not have been what you expected to happen, but that's how it works.

So what does this mean? In order to give the Edit screen a custom unwind segue, you have to make the `UINavigationController` return a new `MyUnwindSegue` object from that `segueForUnwindingToViewController` method. The only way to accomplish this is to subclass `UINavigationController`. (It's ugly but it also makes sense; after all, it's usually only the container that knows how to show and hide the controllers that it embeds.)

Add a new file to the project using the Objective-C class template. Name it **MyINavigationController** and make it a subclass of `UINavigationController`.

Replace **MyNavController.m** with the following:

```
#import "MyNavController.h"
#import "MyUnwindSegue.h"

@implementation MyNavController

- (UIStoryboardSegue *)segueForUnwindingToViewController:
    (UIViewController *)toViewController
    fromViewController:(UIViewController *)fromViewController
    identifier:(NSString *)identifier
{
    if ([identifier isEqualToString:@"DoneEdit"])
        return [[MyUnwindSegue alloc]
            initWithIdentifier:identifier]
```

```

        source:fromViewController
        destination:toViewController];
    else
        return [super segueForUnwindingToViewController:
                toViewController
                fromViewController:fromViewController
                identifier:identifier];
    }

@end

```

If the segue identifier is “DoneEdit,” then this method returns a new `MyUnwindSegue` object. For any other segues, it calls `super` and lets the navigation controller figure it out.

This won’t do anything unless you tell the storyboard to use this new subclass, so in the Storyboard Editor go to the left-most navigation controller (the initial view controller) and in the Identity Inspector change its Custom Class field to “`MyNavigationController`.”

Run the app again. Now pressing the Done button immediately closes the screen without an animation, just the way you programmed it in `MyUnwindSegue`.

If you want to make sure that it’s really the custom segue that got invoked, then add an `NSLog()` to the `done:` method in **MeasurementsViewController.m**:

```

- (IBAction)done:(UIStoryboardSegue *)segue
{
    NSLog(@"%@", segue);
    ...
}

```

The output pane should now say something like:

```
GumStats [...] segue = <MyUnwindSegue: 0xa4780d0>
```

Of course the reason you’re going through all this trouble is to make a segue that looks cool, so replace **MyUnwindSegue.m** with the version that comes with this chapter’s resources. It is essentially the inverse of what `MyCustomSegue` does.

The roles of the segue’s `sourceViewController` and `destinationViewController` properties are now reversed: the source is the one you’re leaving and the destination is the one you’re going to. The animation is almost identical, except that now the views slide out of the screen rather than onto it.

Another difference is that the source view controller is dismissed before the animation starts. This is necessary because you want to show the destination’s view

as well, and that will only work if there is no modal view controller on top. Take a peek at the source code to see exactly what it does.

Run the app again. Now tapping the Done button closes the screen with another fancy animation. Nice!

But what about the Cancel button? Same thing. Add the identifier for its segue to the `segueForUnwindingToViewController` method in **MyNavigationController.m**:

```
- (UIStoryboardSegue *)segueForUnwindingToViewController:
    (UIViewController *)toViewController
    fromViewController:(UIViewController *)fromViewController
    identifier:(NSString *)identifier
{
    if ([identifier isEqualToString:@"DoneEdit"] ||
        [identifier isEqualToString:@"CancelEdit"])
        return [[MyUnwindSegue alloc]
            initWithIdentifier:identifier
            source:fromViewController
            destination:toViewController];
    else
        return [super segueForUnwindingToViewController:
            toViewController
            fromViewController:fromViewController
            identifier:identifier];
}
```

Now the Cancel button also uses a `MyUnwindSegue` object to transition back to `MeasurementsViewController`.

What if you don't want it to go back there, but all the way to the beginning, just like you did earlier? Recall that you jumped back to the beginning by placing the `cancel:` method in `CombinedViewController` instead of `MeasurementsViewController` (it should still be there). To test this, remove the `cancel:` method from `MeasurementsViewController` again and run the app.

You would expect that tapping Cancel now takes you all the way back to the first screen, but it doesn't. Why not? If you place an `NSLog()` or breakpoint into the `cancel:` method, it certainly shows that `cancel:` is being executed, so what could be wrong here?

Well, it turns out that your custom unwind segue simply isn't smart enough. It properly calls `dismissViewControllerAnimated:` to close the Edit screen, but it should also have popped `MeasurementsViewController` off the navigation stack. The default unwind segue that navigation controller provides is clever enough to do that, but `MyUnwindSegue` isn't.

That is why UIKit asks the container view controller to provide the unwind segue. If you override the standard behavior to provide your own, then you also need to

mimic what the container's default segue does. Doing that for `MyUnwindSegue` is beyond the scope of this chapter, but a good exercise if you'd like to experiment more.

Note: As you can tell, writing your own unwind segues is trickier than it sounds. They need to handle the variety of circumstances in which they can be used. Just dismissing a modal view may not be enough.

Your own container controllers and unwind segues

Another place where unwind segues can become a bit complicated is when you combine them with your own container view controllers. You already saw that you had to override `UINavigationController`'s `segueForUnwindingToViewController` to make a custom unwind segue. It gets worse when you create your own containers.

Warning: this section gets a bit esoteric. If you don't have this situation or envision having it in your apps, you might just want to skip ahead and refer back to this when necessary.

Let's say that for some reason you want `DaysViewController` to handle the `cancel:` action. Recall that this view controller is embedded inside `CombinedViewController`. Take the `cancel:` method out of **CombinedViewController.m** and put it in **DaysViewController.m**. Also make sure it is no longer in **MeasurementsViewController.m**.

If you run the app now – surprise, surprise – the Cancel button no longer works.

Apparently UIKit doesn't find the `cancel:` unwind action inside `DaysViewController`. When you tap the Cancel button in the Edit screen, the following happens:

1. UIKit sends the `canPerformUnwindSegueAction:fromViewController:withSender:` message to `EditViewController`. This method returns `NO` because the Edit screen isn't able to handle its own unwind actions. (As mentioned before, it makes no sense for the view controller that triggers the segue to also handle that same segue – you can't segue to yourself.)
2. UIKit then locates the parent view controller of `EditViewController`, which in this case is the top-level navigation controller all the way to the left in the storyboard (now using the `MyNavigationController` class).
3. UIKit sends the message `viewControllerForUnwindSegueAction:fromViewController:withSender:` to the navigation controller. It is the job of the parent view controller to find a child view controller that can handle the unwind action.
4. Inside its implementation for `viewControllerForUnwindSegueAction`, the navigation controller sends the `canPerformUnwindSegueAction` message to all the

controllers on its navigation stack, from top to bottom. The default implementation of `canPerformUnwindSegueAction` simply looks to see if the unwind action method is present, and returns `YES` if it is. (You can also override this method to change its behavior.)

So the navigation controller starts by asking `MeasurementsViewController`, "Can you handle this unwind segue?", which returns `NO` because `MeasurementsViewController` doesn't have the `cancel:` action method. Then it asks `combinedViewController`, which also returns `NO` for the same reason. Finally, it asks itself (again, `NO`).

Because no view controller responds affirmatively, the navigation controller returns `nil` from `viewControllerForUnwindSegueAction`. As a result, nothing happens and the segue is ignored.

If any of the view controllers from the navigation stack had responded with `YES`, then the navigation controller would have returned a pointer to that view controller, and UIKit would have done the following. These are the steps you have already seen before:

5. Send the `segueForUnwindingToViewController:fromViewController:identifier:` message to the navigation controller. This is the method that is supposed to return the `UIStoryboardSegue` object (or subclass) that performs the actual animation.
6. Call `prepareForSegue:` on the source view controller.
7. Call the unwind action method on the destination view controller object that was returned from `viewControllerForUnwindSegueAction`.
8. Call `perform` on the segue object.

The reason the Cancel segue no longer works is that the navigation controller does not ask `DaysViewController` whether it wants to handle the segue. That is not so strange, because `DaysViewController` isn't technically part of the navigation stack – only its parent, `combinedViewController`, is – and therefore, the navigation controller doesn't know about it.

Unfortunately, overriding `canPerformUnwindSegueAction:fromViewController:withSender:` in `CombinedViewController` to return `YES` for the `cancel:` action does not work. If you try that, UIKit assumes that `CombinedViewController` will handle the action, and it tries to call the `cancel:` method on `combinedViewController`. This obviously crashes the app, because that method doesn't exist.

But there is no way for `combinedViewController` to tell UIKit that it really wants `DaysViewController` to handle the unwind action. Of course, this is a contrived situation. In practice, you would almost never want to handle an unwind action in a child view controller like that. But I hope this example makes it clear what goes on under the hood when an unwind segue is attempted.

Note: The only reason that `CombinedViewController` is ever asked to provide a view controller for an unwind segue is if that segue is triggered from within one of its embedded child view controllers.

That is how this whole unwinding system works: when a view controller triggers a segue, it is **the parent view controller container** that is asked to scan through its child view controllers to find one that can handle the unwind action. But if any of those child view controllers is itself a container for other view controllers, it will not ask its children as well. It only goes one level deep.

So remember, only the parent container for the view controller from which the segue originated gets the `viewControllerForUnwindSegueAction` and `segueForUnwindingToViewController` messages.

What does this mean if you want to build your own view controller containers, whether you do that using the new Container View or the manual `addChildViewController:` API?

It means you may need to implement several additional methods in your container – `canPerformUnwindSegueAction`, `viewControllerForUnwindSegueAction`, and `segueForUnwindingToViewController` – to handle unwind segues. But keep in mind that these methods may only be called if the segue was triggered by something inside one of your own child view controllers.

Where to go from here?

That's it for these two big new storyboard features. If you decide to use them in your own projects, then realize that they won't work for iOS 5, only for iOS 6 or better.

Have fun embedding and unwinding!

Chapter 22: What's New with User Interface Customization

By Adam Burkepile

With each consecutive version of iOS, Apple has given us greater ways to customize the appearance of our apps. iOS 6 is no exception, adding more customization options for `UISwitch`, `UISlider`, and many more controls. This makes it even easier to make your apps look unique and great!

In this chapter, you're going to take a look at the new changes with user interface customization in iOS 6. Specifically, you're going to examine:

- **New control appearance APIs.** iOS 5 introduced the ability to change the appearance of many controls (without having to subclass them) with the new appearance APIs. iOS 6 has added even more of these – so in this chapter you'll take a look at the new good stuff that got added!
- **Theming your app.** One of the WWDC 2012 demos covered how to arrange your user interface code into a “theme” so it is easy for you to switch to different themes while in development. This isn't a new idea to iOS 6, but is a pretty cool idea and makes your code simpler to update and understand so we're going to cover it here :] And in this chapter, you'll take this even further than the WWDC demo and learn how to allow the user to switch themes dynamically at runtime!
- **Gradient layers.** `CAGradientLayer` provides a neat way to easily apply a gradient to any view in your app – which can be a nice and easy way to make your app look a lot better. This isn't new to iOS 6, but we're going to briefly discuss this here anyway since it's such a useful technique and we haven't covered it in any of our previous tutorials.

To accomplish all this, you're going to work on a basic photo viewing app. The app will start out “plain vanilla” with the default UIKit controls, but then you'll make it legend... (wait for it) ...dary through the use of the new iOS 6 customization APIs and dynamic theming!

Remember that the focus of this chapter is “what's new in iOS 6”, not “everything there is to know about user interface customization”. This chapter assumes you already have some basic familiarity with the state of user interface customization in iOS 5. If you are completely new to user interface customization in iOS, you might

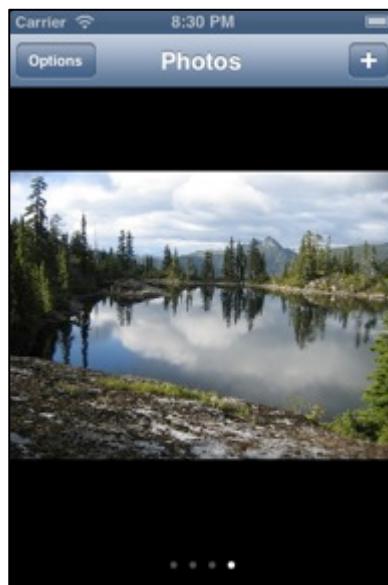
want to read the “Beginning/Intermediate UIKit Customization” chapters from *iOS 5 by Tutorials* first.

Ready to get customizing? Let’s take a stroll and find out what’s new with UI customization!

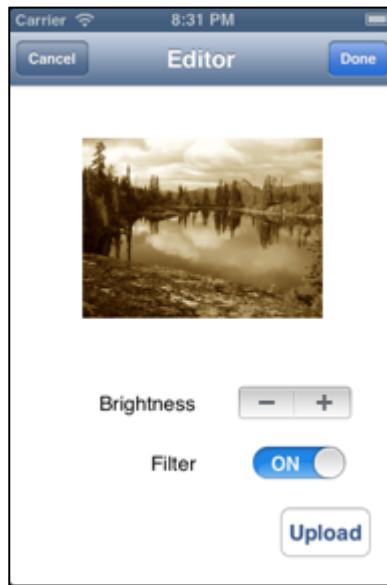


Getting Started

The resources for this chapter include a starter project. Extract the zip, open the project in Xcode, and build and run. You’ll see a simple photo editing app that looks like the following:

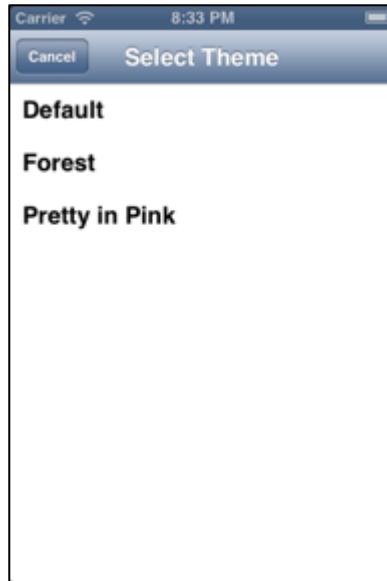


You can swipe left and right to cycle through the photos. You can also tap a photo to adjust its brightness or to apply a Core Image filter:



You can tap the Upload button to display a progress indicator, but it's just there for show (it doesn't actually do anything).

Finally, if you tap Cancel and then Options, you'll see a screen that allows the user to select a theme to apply:



However, selecting one of the themes doesn't do anything at the moment. That's where you come in! 😊

For the rest of this chapter, you're going to build upon this sample project to take things from "plain vanilla boring" to "themed and delicious."

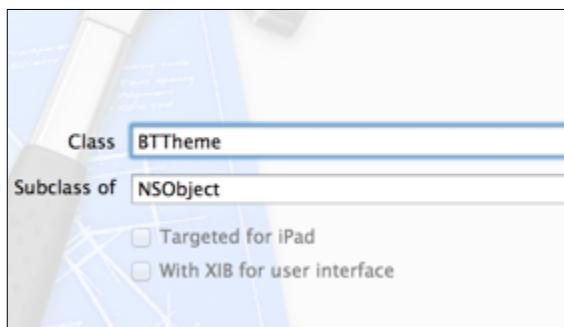
So take a look at the code at this point and make sure you understand how everything fits together. Once you feel comfortable that you understand what's there, read on!

Adding the theme manager

To help give this lackluster app a makeover, you're going to create a "theme manager". Think of this as a crazy (but talented) designer who runs around the app changing the colors and look of each control to make things beautiful.

The advantage of using a theme manager is that it centralizes all of the user interface look and feel logic in one place, which makes it much easier to change and update than the alternative - having it scattered across every view controller in your project!

Let's try this out. Create a new project group and call it **Theme**. Then control-click on the new **Theme** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **BTTheme**, make it a subclass of **NSObject**, and click **Next** and finally **Create**.



Open **BTTheme.h** and replace the contents with:

```
@protocol BTTheme

@end

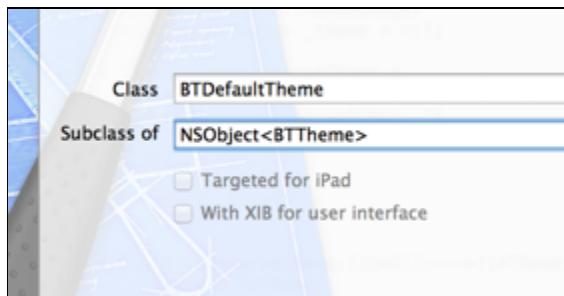
@interface BTThemeManager : NSObject
+ (id<BTTheme>)sharedTheme;
+ (void)setSharedTheme:(id<BTTheme>)inTheme;
+ (void)applyTheme;
@end
```

Here you define two main things:

1. **The theme protocol.** The protocol is called **BTTheme**, and right now is completely empty! You'll be adding new methods to this protocol where each theme can specify how controls should be customized, such as colors or images to apply.

2. The theme manager. `BTThemeManager` is the class that will manage the theme and will apply the theme to the controls. Notice that you define methods to get and set the shared theme, both of which deal with an object that conforms to the `BTTheme` protocol.

At this point, it would be helpful to have at least one default theme. Right-click on the **Theme** folder and create a new file. Use it to create a new Objective-C class called **BTDefaultTheme** and have it inherit from `NSObject<BTTheme>`.



If you haven't seen that <bracket> syntax before, all it means is that this class inherits from `NSObject` while also conforming to the `BTTheme` protocol (there's nothing in that protocol right now, but we have plans for it). Create the class, then add the following `#import` to **BTDefaultTheme.h**:

```
#import "BTTheme.h"
```

Now you can jump back to **BTTheme.m** and replace the contents with this:

```
#import "BTTheme.h"
#import "BTDefaultTheme.h"

@implementation BTThemeManager
static id<BTTheme> _theme = nil;

+ (id<BTTheme>)sharedTheme {
    return _theme;
}

+ (void)setSharedTheme:(id<BTTheme>)inTheme {
    _theme = inTheme;
    [self applyTheme];
}

+ (void)applyTheme {
    id<BTTheme> theme = [self sharedTheme];
}
@end
```

The class contains a static variable that holds the theme, and a getter and setter method to access that variable. The last method is the one that will apply the theme to all of the UI controls.

Let's load up the default theme when the app starts. Switch to **BTAppDelegate.m** and add the following line to the beginning of `application:didFinishLaunchingWithOptions:`:

```
[BTThemeManager setSharedTheme:[BTDefaultTheme new]];
```

Don't forget the needed imports at the top:

```
#import "BTTheme.h"  
#import "BTDefaultTheme.h"
```

It's pretty easy to see what's going on here. When the app starts up, you create a new `BTDefaultTheme` instance and pass that to the shared theme setter, which in turn applies the theme (or at least it will – you'll get to that soon enough).

Build and run to make sure everything works fine, but note that nothing will look or function any differently yet. Patience, my friend!

So now you have a theme manager set up and you are applying a default theme. Let's add another theme so that you have something to switch to.

Control-click on the **Theme** group and once again select **New File**. This will be an **Objective-C class** as well – name it **BTForestTheme**, make it inherit from **BTDefaultTheme**, and click **Next** and finally **Create**.



Now you have two themes! I bet you're feeling powerful already.

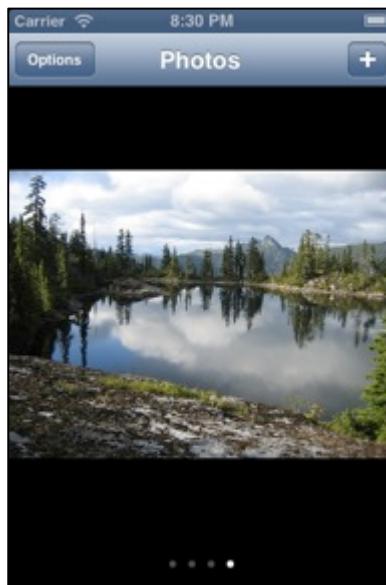
You just need to enable the theme manager to switch between them. To do this, open **BTThemeViewController.m** and add the following imports:

```
#import "BTTheme.h"  
#import "BTDefaultTheme.h"  
#import "BTForestTheme.h"
```

Then add the following to `tableView:didSelectRowAtIndexPath:` after the first line setting up the `idx` variable:

```
if ([self.themes[idx] isEqualToString:@"Forest"]) {
    [BTThemeManager setSharedTheme:[BTForestTheme new]];
}
else
    [BTThemeManager setSharedTheme:[BTDefaultTheme new]];
```

Give it a build and run. When you select the theme in the theme view controller, the theme manager should change the theme for the whole app.



"But wait a minute," the astute among you might say, "that looks exactly the same as before!"

Right you are – although you now have the scaffolding in place, the themes don't actually do anything yet. Let's customize our first object and change that!

The background

Switch back to **BTTheme.h**. You are going to add a method to the protocol that returns the background for the theme. Add the following to the **BTTheme** protocol (between the `@protocol` and `@end`):

```
- (UIColor*)backgroundColor;
```

Then open **BTDefaultTheme.m** and add the implementation:

```
- (UIColor*)backgroundColor {
    return [UIColor whiteColor];
}
```

In the default theme, you want the background to be a solid white color. Let's specify a different background for the Forest theme. Switch to **BTForestTheme.m** and add:

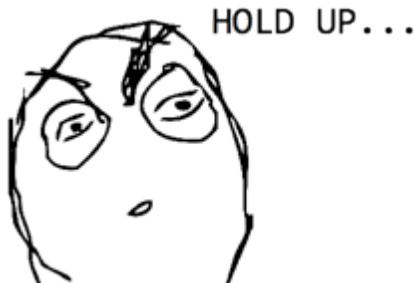
```
- (UIColor*)backgroundColor {
    return [UIColor colorWithPatternImage:
           [UIImage imageNamed:@"bg_forest.png"]];
}
```

This is a little trick to return an image as a color. `UIColors` are nice and easy to work with since all `UIViews` has a `backgroundColor` property – now you can set it to white in the default case, and the `bg_forest.png` color in the forest theme case.

Now you'll create a method that a view controller can call to set the background color of its views. Open **BTTheme.m** and add the following method:

```
+ (void)customizeView:(UIView *)view
{
    id <BTTheme> theme = [self sharedTheme];
    UIColor *backgroundColor = [theme backgroundColor];
    [view setBackgroundColor:backgroundColor];
}
```

As you're typing this method you might have thought the following:



"Why is there a separate `customizeView` method when there's an existing `applyTheme` method that is supposed to apply the current theme to all the relevant controls?"

Well, later on you'll be using the `applyTheme` method to resolve cases where you want to change the look of all controls of a certain type, globally. For example, you might want to change all the navigation bars to have a different background.

But in this case, a lot of standard UI controls are sub-classes of `UIView` – for instance, `UIButton`, `UILabel`, etc. And if you were to globally set the background for all views, then you'd discover that controls you might not have expected will also have their background changed! And this does not make our designer happy. ☺

So, instead you add a special method here that will change the background for only specific views. This gives you control over which views are customized and which aren't!



Don't forget to add the prototype for the above method to **BTTheme.h**:

```
+ (void)customizeView:(UIView *)view;
```

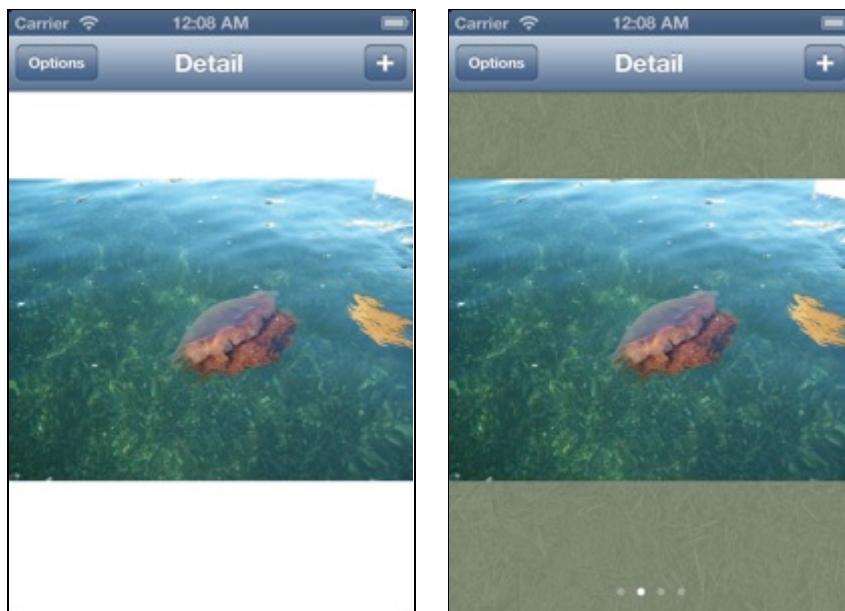
Now in **BTThemeViewController.m**, **BTPhotosViewController.m**, **BTPhotoViewController.m**, and **BTEditorViewController.m**, add the following line to the end of `viewWillAppear:`:

```
[BTThemeManager customizeView:self.view];
```

You will need to add the following import at the top of **all** of the above files, *except* for BTThemeViewController.m:

```
#import "BTTheme.h"
```

Build and run. Try switching to the Forest theme now.



BOOM - the results of all your efforts so far – a leafy background!

Although that is a fine looking leafy background, you might not be super impressed since all you did was change the background. But don't worry - now you have a solid framework that will make the rest of the customization easy.

You can see that your theme manager is working, storing the shared theme when you change it, and accessing and applying the properties from the theme. Now for the fun part - customizing the rest of the controls!

Navigation bar, meet proxy



In iOS 5 Apple introduced some APIs to allow you to change the appearance of a `UINavigationBar`, such as changing its background and title text attributes. In iOS 6 Apple has made this even better – navigation bars now have shadows that are fully customizable!

Before you get to changing the shadow on the navigation bar, let's start with changing the background, which will be a review of iOS 5 customization. Switch to `BTTheme.h` and add new method definitions to the protocol:

- `(UIImage*)imageForNavigationBar;`
- `(UIImage*)imageForNavigationBarLandscape;`
- `(NSDictionary*)navigationBarTextDictionary;`

Here you have getters for the `UINavigationBar` images for portrait and landscape, and a dictionary that will set the attributes of the title text (font, color, etc.) for the bar.

Add the following to `BTDefaultTheme.m`:

- `(UIImage*)imageForNavigationBar{return nil;}`
- `(UIImage*)imageForNavigationBarLandscape{return nil;}`
- `(NSDictionary*)navigationBarTextDictionary { return nil; }`

When you set the properties to `nil`, you just get the default appearances.

Now add the following to `BTForestTheme.m`:

- ```
- (UIImage*)imageForNavigationBar {
 return [[UIImage imageNamed:@"nav_forest_portrait.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 100.0, 0.0, 100.0)];
}
```

```

- (UIImage*)imageForNavigationBarLandscape{
 return [[UIImage imageNamed:@"nav_forest_landscape.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 100.0, 0.0, 100.0)];
}
- (NSDictionary*)navigationBarTextDictionary { return @{
 UITextAttributeFont:[UIFont fontWithName:@"Optima" size:24.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.910 green:0.914 blue:0.824
 alpha:1.000],
 UITextAttributeTextShadowColor:
 [UIColor colorWithRed:0.224 green:0.173 blue:0.114
 alpha:1.000],
 UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, -1)]
 };
}
}

```

The above code returns the images to use for the navigation bar in both portrait and landscape, and the properties to use for the navigation bar text.

Now switch to **BTTheme.m** and add the following to `applyTheme`:

```

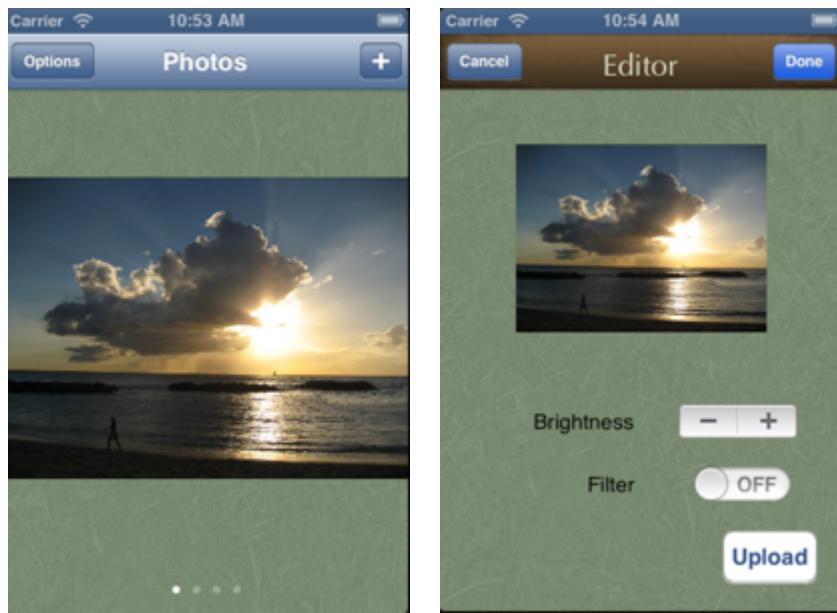
UINavigationBar* NavBarAppearence =
[UINavigationBar appearance];
NavBarAppearence setBackgroundImage:[theme
 imageForNavigationBar]
 forBarMetrics:UIBarMetricsDefault];
NavBarAppearence setBackgroundImage:[theme
 imageForNavigationBarLandscape]
 forBarMetrics:UIBarMetricsLandscapePhone];
NavBarAppearence setTitleTextAttributes:[theme
 navBarTextDictionary]];

```

And here you have your first use of the appearance proxy – remember this is the way to change the appearance of *all* controls of a particular type across the app. The above code gets the appearance proxy for the `UINavigationBar` class, and then sets the default appearance for all navigation bars to fit the current theme.

Build and run. Switch to the Forest theme now. Hmm, it didn't seem to work!

But hold on a second. When you tap on a photo and open up the Editor view, you can see that the navigation bar has changed!



Furthermore, when you close the Editor, the navigation bar in the Photos view has changed too! What gives?

This is something you have to be aware of when you use appearance proxies to customize appearances. When you set a property through the appearance proxy, it only applies to *new* objects that are added to the window. It will not automatically apply itself to objects already in the displayed window. You need to update those yourself.

To do this, you're going to use an `NSNotification` to send a message to the app delegate when you switch themes. Add the following to **BTTheme.m** in the **BTThemeManager** class:

```
+ (void)customizeNavigationBar:(UINavigationBar *)navigationBar {
 id <BTTheme> theme = [self sharedTheme];
 [navigationBar setBackgroundImage:[theme
 imageForNavigationBar]
 forBarMetrics:UIBarMetricsDefault];
 [navigationBar setBackgroundImage:[theme
 imageForNavigationBarLandscape]
 forBarMetrics:UIBarMetricsLandscapePhone];
 [navigationBar setTitleTextAttributes:[theme
 navBarTextDictionary]];
}
```

The above method customizes the look of a specific navigation bar instance, rather than all the navigation bars.

## Not... all the bars?



Even though it might make that little guy sad, it's all for the best. When a theme changes, you'll send out a notification which the app delegate will be listening for. When the notification occurs, the the app delegate can call this method to update the previously created navigation bar appropriately.

Also add the method declaration/prototype in **BTTheme.h**:

```
+ (void)customizeNavigationBar:(UINavigationBar *)navigationBar;
```

Next, open **Appearance-Prefix.pch** (it's in the Supporting Files group). This is the precompiled header that gets added to each file at compile time. If you have something that needs to be in a lot of the files in the project, this is a good place to add it.

Put this line at the end of the file (after the `#endif`):

```
#define kThemeChange @"NotificationThemeChanged"
```

Now add the following to **BTAppDelegate.m** at the end of `application:didFinishLaunchingWithOptions:` (but before the final `return`):

```
[NSNotificationCenter defaultCenter]
 addObserverForName:kThemeChange
 object:nil
 queue:[NSOperationQueue mainQueue]
 usingBlock:^(NSNotification *note) {
 UINavigationController* navController =
 (UINavigationController*)[[UIApplication sharedApplication]
 keyWindow] rootViewController];

 [BTThemeManager customizeNavigationBar:
 navController.navigationBar];
 }];
}
```

This sets up the "listener" part of the notification and calls the `customizeNavigationBar` method you just set up on the main app `UINavigationBar` when a notification about theme changes is received. This way, you can ensure that when your theme changes, the navigation bar theme is also explicitly changed.

Now, add the following line to `applyTheme` in **BTTheme.m**:

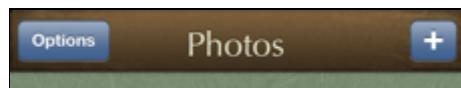
```
[[NSNotificationCenter defaultCenter]
 postNotification:[NSNotification
 notificationWithName:kThemeChange
 object:nil]];
```

This is the part that sends out the notification to anyone listening for it. So, whenever you change themes, you call the `applyTheme` method, which sends the `kThemeChange` notification, which the AppDelegate then picks up and calls the `customizeNavigationBar` method to set the navigation bar. Pretty cool, eh?

Build and run, and everything should change as expected:



Default Theme



Forest Theme

Whoohoo – now it's starting to come together!

Next, let's change the shadow below the navigation bar. It's a subtle thing added in iOS 6, but it's a nice touch that you shouldn't neglect. Add the following code to the appropriate files:

To the `BTTheme` protocol in **BTTheme.h**:

```
- (UIImage*)imageForNavigationBarShadow;
```

To **BTDefaultTheme.m**:

```
- (UIImage*)imageForNavigationBarShadow{return nil;}
```

To **BTForestTheme.m**:

```
- (UIImage*)imageForNavigationBarShadow{
 return [UIImage imageNamed:@"topShadow_forest.png"];
}
```

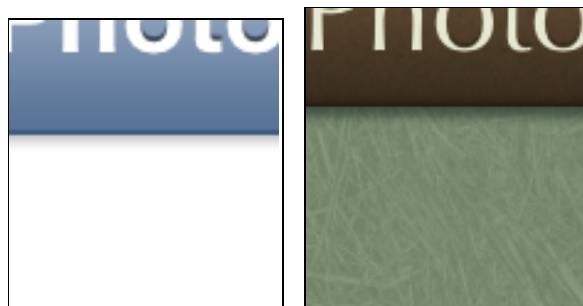
To `applyTheme` in **BTTheme.m**:

```
[NavBarAppearence setShadowImage:[theme
 imageForNavigationBarShadow]];
```

And to customizeNavigationBar in **BTTheme.m**:

```
[navigationBar setShadowImage:[theme
imageForNavigationBarShadow]];
```

Build and run.



Like I said, subtle. But these little details make an impression!

## Bar button items

So you got the navigation bar customized, but it looks pretty weird with the buttons not matching the Forest theme. So let's look at customizing those next.



iOS 5 gave us an easy way to customize bar buttons, and in iOS 6 Apple has added customization of the different button styles. You'll get to see examples of those in the Editor view. Add the following code to the **BTTheme** protocol in **BTTheme.h**:

```
- (UIImage*)imageForBarButtonNormal;
- (UIImage*)imageForBarButtonHighlighted;
- (UIImage*)imageForBarButtonNormalLandscape;
- (UIImage*)imageForBarButtonHighlightedLandscape;
- (UIImage*)imageForBarButtonDoneNormal;
- (UIImage*)imageForBarButtonDoneHighlighted;
- (UIImage*)imageForBarButtonDoneNormalLandscape;
- (UIImage*)imageForBarButtonDoneHighlightedLandscape;
- (NSDictionary*)barButtonTextDictionary;
```

As you can see, you have a few more properties this time. You have methods to return images for a standard button's normal and highlighted states, a done button's normal and highlighted states, and the corresponding landscape versions for each of those buttons, for a total of 8 different images that need to be provided. In addition, there's a method that will set the text attributes for a bar button item.

Add to **BTDefaultTheme.m**:

```

- (UIImage*)imageForBarButtonNormal { return nil; }
- (UIImage*)imageForBarButtonHighlighted { return nil; }
- (UIImage*)imageForBarButtonNormalLandscape { return nil; }
- (UIImage*)imageForBarButtonHighlightedLandscape { return nil; }
- (UIImage*)imageForBarButtonDoneNormal { return nil; }
- (UIImage*)imageForBarButtonDoneHighlighted { return nil; }
- (UIImage*)imageForBarButtonDoneNormalLandscape { return nil; }
- (UIImage*)imageForBarButtonDoneHighlightedLandscape
 { return nil; }
- (NSDictionary*)barButtonTextDictionary {
 return @{
 UITextAttributeFont:[UIFont fontWithName:@"Helvetica-Bold"
 size:12.0f],
 UITextAttributeTextColor:[UIColor whiteColor],
 UITextAttributeTextShadowColor:[UIColor blackColor],
 UITextAttributeTextShadowOffset:[NSValue
 valueWithUIOffset:UIOffsetMake(0, -1)]
 };
}

```

And to **BTForestTheme.m**:

```

- (UIImage*)imageForBarButtonNormal {
 return [[UIImage imageNamed:@"barbutton_forest_uns.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)]; }
- (UIImage*)imageForBarButtonHighlighted {
 return [[UIImage imageNamed:@"barbutton_forest_sel.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)]; }
- (UIImage*)imageForBarButtonDoneNormal {
 return [[UIImage imageNamed:@"barbutton_forest_done_uns.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)]; }
- (UIImage*)imageForBarButtonDoneHighlighted {
 return [[UIImage imageNamed:@"barbutton_forest_done_sel.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)]; }
- (UIImage*)imageForBarButtonNormalLandscape {
 return [[UIImage imageNamed:
 @"barbutton_forest_landscape_uns.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)]; }
- (UIImage*)imageForBarButtonHighlightedLandscape {

```

```

 return [[UIImage imageNamed:
 @"barbutton_forest_landscape_sel.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)];
}
- (UIImage*)imageForBarButtonDoneNormalLandscape {
 return [[UIImage imageNamed:
 @"barbutton_forest_done_landscape_uns.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)];
}
- (UIImage*)imageForBarButtonDoneHighlightedLandscape {
 return [[UIImage imageNamed:
 @"barbutton_forest_done_landscape_sel.png"]
 resizableImageWithCapInsets:UIEdgeInsetsMake(
 0.0, 5.0, 0.0, 5.0)];
}
- (NSDictionary*)barButtonTextDictionary {
 return @{
 UITextAttributeFont:[UIFont fontWithName:@"Optima" size:18.0],
 UITextAttributeTextColor:[UIColor colorWithRed:0.965 green:0.976
 blue:0.875 alpha:1.000],
 UITextAttributeTextShadowColor:[UIColor colorWithRed:0.224
 green:0.173 blue:0.114 alpha:1.000],
 UITextAttributeTextShadowOffset:[NSValue
 valueWithUIOffset:UIOffsetMake(0, 1)]
 };
}
}

```

This returns the images for the buttons in their normal and highlighted states, for both normal and landscape. Note that there is a separate case for done – that is the part that's new in iOS 6. You can now have separate images for different bar button styles (like the done style).

And finally, add this to the bottom of `applyTheme` in **BTTheme.m**:

```

UIBarButtonItem* barButton = [UIBarButtonItem appearance];

[barButton setBackgroundImage:[theme imageForBarButtonNormal]
 forState:UIControlStateNormal
 barMetrics:UIBarMetricsDefault];
[barButton setBackgroundImage:[theme
 imageForBarButtonHighlighted]
 forState:UIControlStateHighlighted
 barMetrics:UIBarMetricsDefault];
[barButton setBackgroundImage:[theme
 imageForBarButtonNormalLandscape]
 forState:UIControlStateNormal
 barMetrics:UIBarMetricsLandscapePhone];

```

```
[barButton setBackgroundImage:[theme
 imageForBarButtonHighlightedLandscape]
 forState:UIControlStateNormal
 barMetrics:UIBarMetricsLandscapePhone];

[barButton setBackgroundImage:[theme
 imageForBarButtonDoneNormal]
 forState:UIControlStateNormal
 style:UIBarButtonItemStyleDone
 barMetrics:UIBarMetricsDefault];
[barButton setBackgroundImage:[theme
 imageForBarButtonDoneHighlighted]
 forState:UIControlStateHighlighted
 style:UIBarButtonItemStyleDone
 barMetrics:UIBarMetricsDefault];
[barButton setBackgroundImage:[theme
 imageForBarButtonDoneNormalLandscape]
 forState:UIControlStateNormal
 style:UIBarButtonItemStyleDone
 barMetrics:UIBarMetricsLandscapePhone];
[barButton setBackgroundImage:[theme
 imageForBarButtonDoneHighlightedLandscape]
 forState:UIControlStateHighlighted
 style:UIBarButtonItemStyleDone
 barMetrics:UIBarMetricsLandscapePhone];
[barButton setTitleTextAttributes:[theme
 barButtonTextDictionary]
 forState:UIControlStateNormal];

[barButton setTitleTextAttributes:[theme
 barButtonTextDictionary]
 forState:UIControlStateNormal];
```

Here you'll see that there is a new `setBackgroundImage:forState:style:barMetrics` method, that allows you to pass in the style of the button to apply the background image (the done style in this case).

Give it a build and run. Ah, that's looks much better now!



Default Theme



## Forest Theme

## Paging custom control, do you copy?

Now let's look at the last remaining element to customize on the Photos View Controller, the page control at the bottom.



In iOS 5 you couldn't (easily) customize this, but now you can! iOS 6 provides a `tintColor` and `currentTintColor` property `UIPageControl`.

In the default theme, you can't even see the control because it's white-on-white, so that definitely needs to change. The Forest theme control could also benefit from a color change so that it stands out a bit more.

Add the following code to `BTTheme` protocol in **BTTheme.h**:

```
- (UIColor*)pageTintColor;
- (UIColor*)pageCurrentTintColor;
```

To **BTDefaultTheme.m**:

```
- (UIColor*)pageTintColor {
 return [UIColor lightGrayColor];
}
- (UIColor*)pageCurrentTintColor {
 return [UIColor blackColor];
}
```

To **BTForestTheme.m**:

```
- (UIColor*)pageTintColor {
 return [UIColor colorWithRed:0.973 green:0.984
 blue:0.875 alpha:1.000];
}
- (UIColor*)pageCurrentTintColor {
 return [UIColor colorWithRed:0.063 green:0.169
 blue:0.071 alpha:1.000];
}
```

And to `applyTheme` in **BTTheme.m**:

```
UIPageControl* pageAppearence = [UIPageControl appearance];
[pageAppearence setCurrentPageIndicatorTintColor:
 [theme pageCurrentTintColor]];
```

```
[pageAppearence setPageIndicatorTintColor:
 [theme pageTintColor]];
```

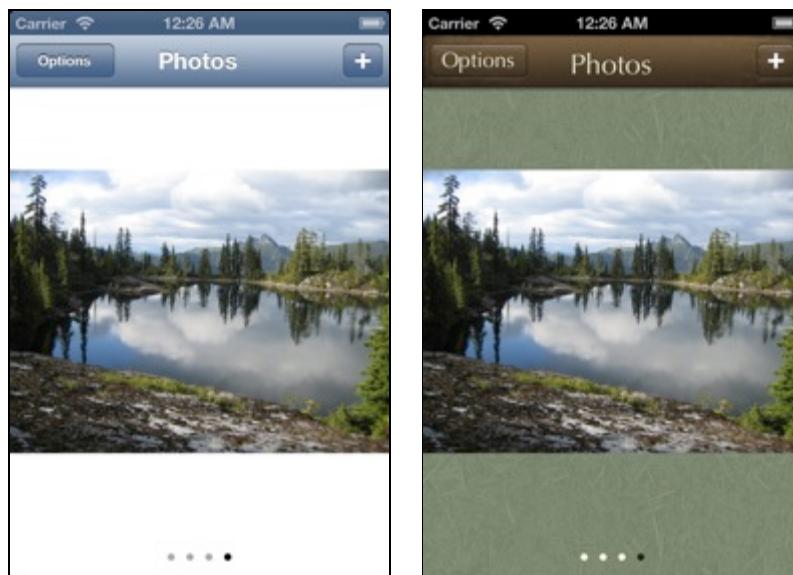
Build and run. You should now see the page control with the default theme:



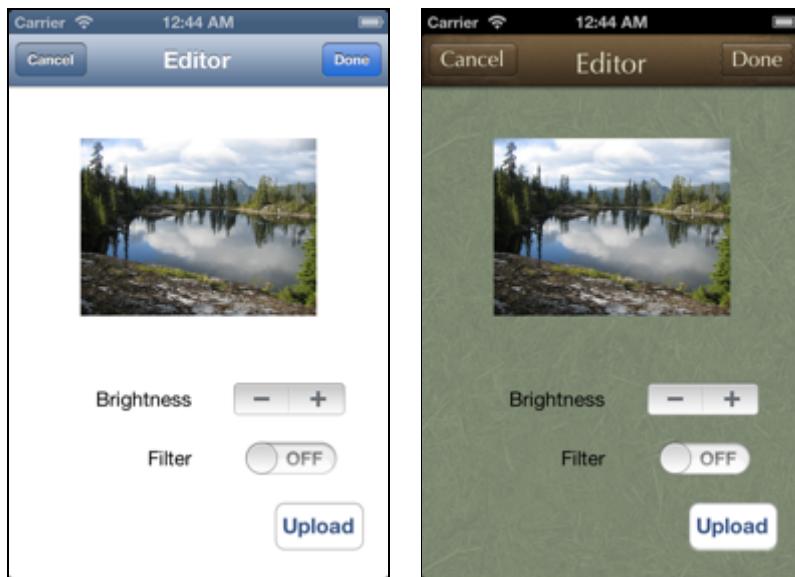
And with the Forest theme:



And with that, you've customized all of the controls on the Photos View Controller. Take a bow!



Looks pretty nice, huh? But this chapter's not over, which means it's time to move on to the Editor View Controller.



As you can see, all the customizations that you've already done are also present in this view (since the appearance proxy changes the look for the relevant controls globally), but some of the controls on this view still need customizing. Let's take them one-by-one.

## Stepping out

Begin with the `UIStepper` control.



This control was introduced in iOS 5, but it wasn't very customizable (by "very" I mean "at all"). With iOS 6, you now have access to the standard tint color, as well as the ability to set custom images for the plus and minus signs, the background for the control, and the middle divider.

Customize the `UIStepper` by adding the following code, first to the `BTTheme` protocol in **BTTheme.h**:

```
- (UIImage*)imageForStepperUnselected;
- (UIImage*)imageForStepperSelected;
- (UIImage*)imageForStepperDecrement;
- (UIImage*)imageForStepperIncrement;
- (UIImage*)imageForStepperDividerUnselected;
- (UIImage*)imageForStepperDividerSelected;
```

To **BTDefaultTheme.m**:

```
- (UIImage*)imageForStepperUnselected{return nil;}
- (UIImage*)imageForStepperSelected{return nil;}
- (UIImage*)imageForStepperDecrement{return nil;}
- (UIImage*)imageForStepperIncrement{return nil;}
- (UIImage*)imageForStepperDividerUnselected{return nil;}
- (UIImage*)imageForStepperDividerSelected{return nil;}
```

To **BTForestTheme.m**:

```
- (UIImage*)imageForStepperUnselected{
 return [UIImage imageNamed:@"stepper_forest_bg_uns.png"];
}
- (UIImage*)imageForStepperSelected{
 return [UIImage imageNamed:@"stepper_forest_bg_sel.png"];
}
- (UIImage*)imageForStepperDecrement{
 return [UIImage imageNamed:@"stepper_forest_decrement.png"];
}
- (UIImage*)imageForStepperIncrement{
 return [UIImage imageNamed:@"stepper_forest_increment.png"];
}
- (UIImage*)imageForStepperDividerUnselected{
 return [UIImage imageNamed:@"stepper_forest_divider_uns.png"];
}
- (UIImage*)imageForStepperDividerSelected{
 return [UIImage imageNamed:@"stepper_forest_divider_sel.png"];
}
```

And to applyTheme in **BTTheme.m**:

```
UIStepper* stepperAppearence = [UIStepper appearance];
[stepperAppearence setBackgroundImage:
 [theme imageForStepperUnselected]
 forState:UIControlStateNormal];
[stepperAppearence setBackgroundImage:
 [theme imageForStepperSelected]
 forState:UIControlStateHighlighted];
[stepperAppearence setDividerImage:
 [theme imageForStepperDividerUnselected]
 forState:UIControlStateNormal
 rightSegmentState:UIControlStateNormal];
[stepperAppearence setDividerImage:
 [theme imageForStepperDividerSelected]
 forState:UIControlStateSelected
 rightSegmentState:UIControlStateNormal];
[stepperAppearence setDividerImage:
```

```

 [theme imageForStepperDividerSelected]
forLeftSegmentState:UIControlStateNormal
rightSegmentState:UIControlStateSelected];
[stepperAppearance setDividerImage:
 [theme imageForStepperDividerSelected]
forLeftSegmentState:UIControlStateSelected
rightSegmentState:UIControlStateSelected];
[stepperAppearance setIncrementImage:
 [theme imageForStepperIncrement]
forState:UIControlStateNormal];
[stepperAppearance setDecrementImage:
 [theme imageForStepperDecrement]
forState:UIControlStateNormal];

```

Build and run. You should now see your customized stepper control:



## Switching it up

The switch is an interesting control that has been with us since its introduction in iOS 2, but it didn't have much customizability until iOS 5. Even then, it was just the tint color for one side of the control.

With iOS 6, you have much more robust control over the appearance of the switch, like being able to set the images and separate tint colors for each side, and to set the thumb image.

Add the following code, first to the `BTTheme` protocol in `BTTheme.h`:

```

- (UIColor*)switchOnTintColor;
- (UIColor*)switchThumbTintColor;
- (UIImage*)imageForSwitchOn;
- (UIImage*)imageForSwitchOff;

```

To `BTDefaultTheme.m`:

```

- (UIColor*)switchOnTintColor { return nil; }
- (UIColor*)switchThumbTintColor { return nil; }
- (UIImage*)imageForSwitchOn{return nil;}
- (UIImage*)imageForSwitchOff{return nil;}

```

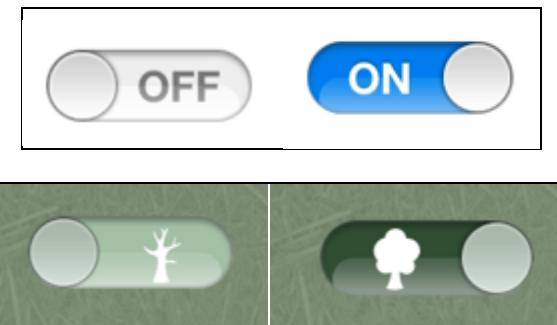
### To BTForestTheme.m:

```
- (UIColor*)switchOnTintColor {
 return [UIColor colorWithRed:0.192 green:0.298
 blue:0.200 alpha:1.000];
}
- (UIColor*)switchThumbTintColor {
 return [UIColor colorWithRed:0.643 green:0.749
 blue:0.651 alpha:1.000];
}
- (UIImage*)imageForSwitchOn{
 return [UIImage imageNamed:@"tree_on.png"];
}
- (UIImage*)imageForSwitchOff{
 return [UIImage imageNamed:@"tree_off.png"];
}
```

### And to applyTheme in BTTheme.m:

```
UISwitch* switchAppearence = [UISwitch appearance];
[switchAppearence setTintColor:[theme switchThumbTintColor]];
[switchAppearence setOnTintColor:[theme switchOnTintColor]];
[switchAppearence setOnImage:[theme imageForSwitchOn]];
[switchAppearence setOffImage:[theme imageForSwitchOff]];
```

Build and run. Go to the Editor screen for both themes. It's a pretty simple and straightforward customization.



Are you more of a winter tree person, or a summer tree person? That might say a lot about your personality. ☺

## Progressing to the progress view

The progress view is a control that has had good customization options since iOS 5. You can customize the track color and completed track color. Nothing is new in iOS

6, but we haven't covered this before so we thought we'd show it off real quick here.

Add the following to the **BTTheme** protocol in **BTTheme.h**:

```
- (UIColor*)progressBarTintColor;
- (UIColor*)progressBarTrackTintColor;
```

To **BTDefaultTheme.m**:

```
- (UIColor*)progressBarTintColor { return nil; }
- (UIColor*)progressBarTrackTintColor { return nil; }
```

To **BTForestTheme.m**:

```
- (UIColor*)progressBarTintColor {
 return [UIColor colorWithRed:0.200 green:0.345
 blue:0.212 alpha:1.000];
}
- (UIColor*)progressBarTrackTintColor {
 return [UIColor colorWithRed:0.541 green:0.647
 blue:0.549 alpha:1.000];
}
```

And to `applyTheme` in **BTTheme.m**:

```
UIProgressView* progressAppearence = [UIProgressView
appearance];
[progressAppearence setProgressTintColor:[theme
progressBarTintColor]];
[progressAppearence setTrackTintColor:[theme
progressBarTrackTintColor]];
```

Build and run. You should see the new progress view now in the Forest theme.



## Labels, which can be cranky

The standard label control can be difficult to work with at times. You could always set basic color and font properties, but more complex attributed strings were not available until iOS 6, with a slew of third-party libraries trying to fill the gap.

For this chapter, you're just going to focus on the font and color properties, but this book has a ton of more information about cool things you can do with attributed strings! If you want to learn more, check out Chapter 15, "Attributed Strings."

Let's customize the labels on the Editor view controller to match the themes. First, add the following to the **BTTheme** protocol in **BTTheme.h**:

```
- (NSDictionary*)labelTextDictionary;
```

To **BTDefaultTheme.m**:

```
- (NSDictionary*)labelTextDictionary { return nil; }
```

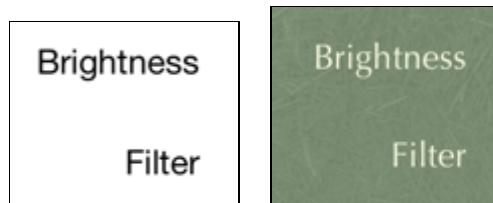
To **BTForestTheme.m**:

```
- (NSDictionary*)labelTextDictionary {
 return @{@"
 UITextAttributeFont:[UIFont fontWithName:@"Optima"
 size:18.0],
 UITextAttributeTextColor:[UIColor colorWithRed:0.965
 green:0.976 blue:0.875 alpha:1.000],
 UITextAttributeTextShadowColor:[UIColor colorWithRed:0.212
 green:0.263 blue:0.208 alpha:1.000],
 UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
 };
}
```

And to `applyTheme` in **BTTheme.m**:

```
UILabel* labelAppearance = [UILabel appearance];
[labelAppearance setTextColor:
 [theme labelTextDictionary][UITextAttributeTextColor]];
[labelAppearance setFont:
 [theme labelTextDictionary][UITextAttributeFont]];
```

Build and run again. You should now see the new font and color on the Editor screen.



However, you will also start noticing some weird issues at this point – the table view for displaying the themes will be formatted differently, and the button text on the bar button items will show different fonts at times.

These are both known issues of globally-customizing labels, since labels are used as internal building blocks for other controls, such as table cell views. So it's definitely something to watch out for.

Since you already know how to modify individual controls (the same way view backgrounds were set, and the same way buttons will be customized in the next section), I'll leave it as an exercise for you to modify the implementation to customize just the labels you want.

## Buttons, one at a time

Ever since the beginning of iOS, it's been easy to customize buttons, since it's as simple as setting the background image of a button. But iOS 5 made it easy to globally update buttons with the `UIAppearance` proxy. In this section, you'll update the button with a resizable image.

You'll supply both in your themes. First add the following to the `BTTheme` protocol in `BTTheme.h`:

```
- (NSDictionary*)buttonTextDictionary;
- (UIImage*)imageForButtonNormal;
- (UIImage*)imageForButtonHighlighted;
```

Then to the `BTThemeManager` interface in `BTTheme.h`:

```
+ (void)customizeButton:(UIButton*)button;
```

Buttons are similar to views, in that they are so ubiquitous, you don't want to try customizing them globally – you just wouldn't like the result. ☺ So here you add a method that allows you to customize the look of individual buttons.

Add this to `BTDefaultTheme.m`:

```
- (NSDictionary*)buttonTextDictionary { return nil; }
- (UIImage*)imageForButtonNormal { return nil; }
- (UIImage*)imageForButtonHighlighted { return nil; }
```

To `BTForestTheme.m`:

```
- (NSDictionary*)buttonTextDictionary { return @{
 UITextAttributeFont:[UIFont fontWithName:@"Optima" size:15.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.965 green:0.976
 blue:0.875 alpha:1.000],
 UITextAttributeTextShadowColor:
```

```

 [UIColor colorWithRed:0.212 green:0.263
 blue:0.208 alpha:1.000],
UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, -1)]
};

}

- (UIImage*)imageForButtonNormal {
 return [[UIImage imageNamed:@"button_forest_uns.png"]
 resizableImageWithCapInsets:
 UIEdgeInsetsMake(0.0, 8.0, 0.0, 8.0)];
}

- (UIImage*)imageForButtonHighlighted {
 return [[UIImage imageNamed:@"button_forest_sel.png"]
 resizableImageWithCapInsets:
 UIEdgeInsetsMake(0.0, 8.0, 0.0, 8.0)];
}
}

```

And to **BTTheme.m**:

```

+ (void)customizeButton:(UIButton*)button {
 id <BTTheme> theme = [self sharedTheme];

 [button setTitleColor:
 [theme buttonTextDictionary][UITextAttributeTextColor]
 forState:UIControlStateNormal];
 [[button titleLabel] setFont:
 [theme buttonTextDictionary][UITextAttributeFont]];
 [button setBackgroundImage:
 [theme imageForButtonNormal]?
 [theme imageForButtonNormal] : nil
 forState:UIControlStateNormal];
 [button setBackgroundImage:
 [theme imageForButtonHighlighted]?
 [theme imageForButtonHighlighted]:nil
 forState:UIControlStateHighlighted];
}

```

Because you only want to apply the customization to the Upload button on the Editor view controller, add the following line in `viewDidLoad:` in

**BTEditorViewController.m**:

```
[BTThemeManager customizeButton:ibUploadButton];
```

Build and run, and check out the results:



And just like that, you should now have a beautiful Upload button in the Forest theme.

## Gradient layers in table view cells

You're now going to turn your attention to the final part of the app that needs to be themed, the theme selection view controller itself. And really, all that's left to be customized are the table view cells, since you've already taken care of the navigation bar and the buttons.

This will be a little different than the other customizations you've done in this chapter, because you are going to use a `CAGradientLayer` to create a nice-looking background for the table view cells. You'll specify the colors you need, the text properties, and a gradient layer class that uses the colors to describe the gradient that should be used with the cell.

First add the following to the `BTTheme` protocol in `BTTheme.h`:

```
- (UIColor*)upperGradient;
- (UIColor*)lowerGradient;
- (UIColor*)seperatorColor;
- (Class)gradientLayer;
- (NSDictionary*)tableViewCellTextDictionary;
```

Then to the `BTThemeManager` interface in `BTTheme.h`:

```
+ (void)customizeTableViewCell:(UITableViewCell*)tableViewCell;
```

To `BTDefaultTheme.m`:

```
- (UIColor*)upperGradient{
 return [UIColor whiteColor];
}
- (UIColor*)lowerGradient {
 return [UIColor whiteColor];
}
- (UIColor*)seperatorColor {
 return [UIColor blackColor];
}
- (Class)gradientLayer {
```

```

 return [BTGradientLayer class];
 }
- (NSDictionary*)tableViewCellTextDictionary {
 return @{
 UITextAttributeFont:[UIFont boldSystemFontOfSize:20.0]
 };
}
}

```

To **BTForestTheme.m**:

```

- (UIColor*)upperGradient{
 return [UIColor colorWithRed:0.976 green:0.976
 blue:0.937 alpha:1.000];
}
- (UIColor*)lowerGradient {
 return [UIColor colorWithRed:0.969 green:0.976
 blue:0.878 alpha:1.000];
}
- (UIColor*)seperatorColor {
 return [UIColor colorWithRed:0.753 green:0.749
 blue:0.698 alpha:1.000];
}
- (NSDictionary*)tableViewCellTextDictionary { return @{
 UITextAttributeFont:[UIFont fontWithName:@"Optima" size:24.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.169 green:0.169
 blue:0.153 alpha:1.000],
 UITextAttributeTextShadowColor:
 [UIColor colorWithWhite:1.000 alpha:1.000],
 UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
 };
}
}

```

And to **BTTheme.m**:

```

+ (void)customizeTableViewCell:(UITableViewCell*)tableViewCell {
 id <BTTheme> theme = [self sharedTheme];

 [[tableViewCell.textLabel] setTextColor:
 [theme tableViewCellTextDictionary][UITextAttributeTextColor]];
 [[tableViewCell.textLabel] setFont:
 [theme tableViewCellTextDictionary][UITextAttributeFont]];
}

```

Finally, add the following to `tableView:cellForRowAtIndexPath:` (right after the cell initialization) in **BTThemeViewController.m**:

```
[BTThemeManager customizeTableViewCell:cell];
```

So you have set up the font properties you need and are applying them to each cell when the cell gets retrieved for the table view. You have also defined the colors you are going to be using for the gradient background,

But where's the gradient background? Time to create it! Add the following code to **BTDefaultTheme.h**:

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

@interface BTGradientLayer : CAGradientLayer
@end
```

And to **BTDefaultTheme.m** (after the final @end in the file):

```
@implementation BTGradientLayer
- (id)init{
 if(self = [super init]) {
 UIColor *colorOne = [[BTThemeManager sharedTheme]
 upperGradient];
 UIColor *colorTwo = [[BTThemeManager sharedTheme]
 lowerGradient];
 UIColor *colorThree = [[BTThemeManager sharedTheme]
 separatorColor];

 NSArray *colors = [NSArray arrayWithObjects:
 (id)colorOne.CGColor,
 colorTwo.CGColor,
 colorThree.CGColor, nil];

 self.colors = colors;

 NSNumber *stopOne = [NSNumber numberWithFloat:0.0];
 NSNumber *stopTwo = [NSNumber numberWithFloat:0.98];
 NSNumber *stopThree = [NSNumber numberWithFloat:1.0];

 NSArray *locations = [NSArray arrayWithObjects:
 stopOne,
 stopTwo,
 stopThree, nil];

 self.locations = locations;
 }
}
```

```
 self.startPoint = CGPointMake(0.5, 0.0);
 self.endPoint = CGPointMake(0.5, 1.0);
}

return self;
}
@end
```

It's not too hard to see what a `CAGradientLayer` does or how it works. You set up an array of colors, which you are able to pull from your shared theme, and you also set up an array that specifies the positions where each color falls.

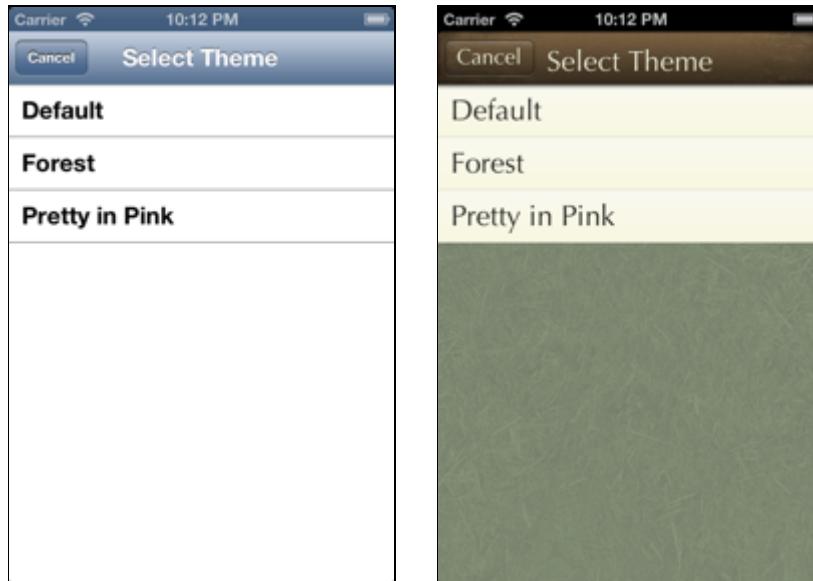
Then to implement your custom gradient layer, you override the `layerClass` method in the custom table view cell class to return your gradient layer.

Handle that final step by adding the following code to **BTThemeTableViewCell.m**:

```
#import "BTTheme.h"

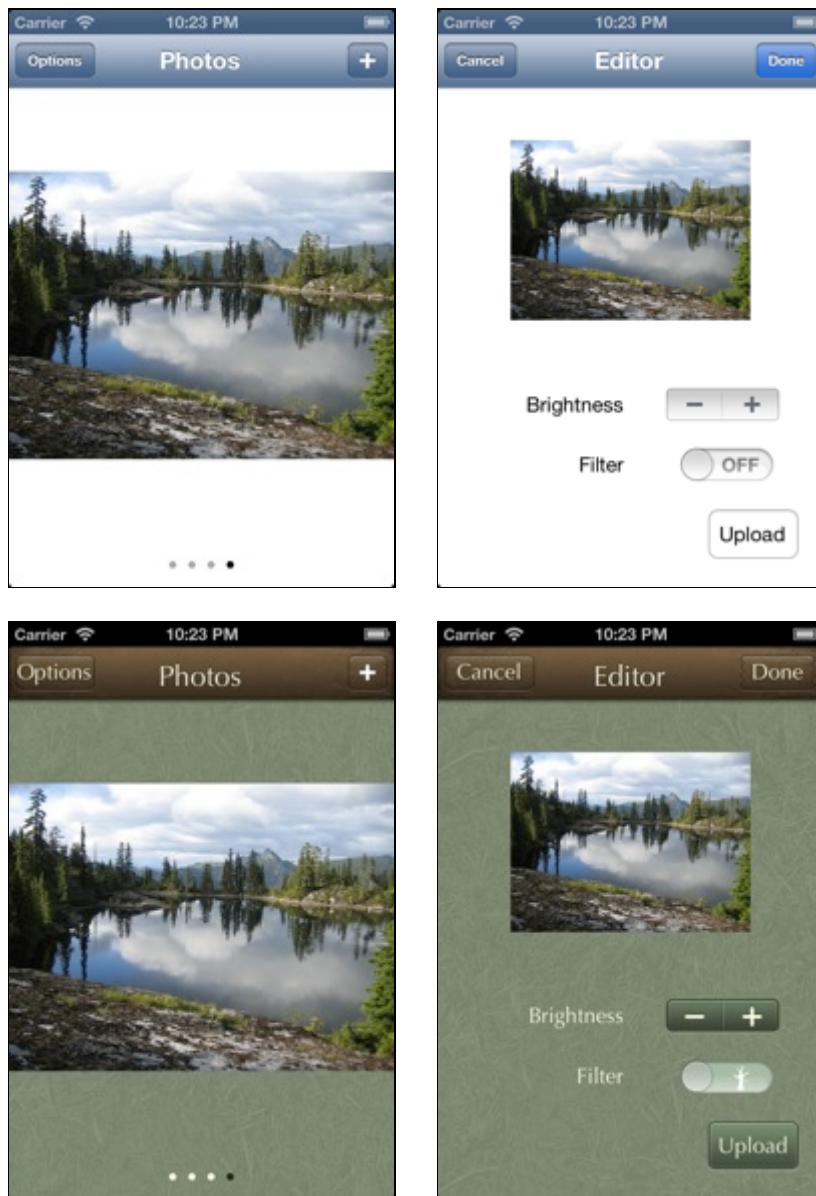
+(Class)layerClass{
 return [[BTThemeManager sharedTheme] gradientLayer];
}
```

Build and run, and you can see the new styling for the table view cells when using the Forest theme.



Speaking of which, what is this Pretty in Pink theme you see in the table view? It's about time to find out.

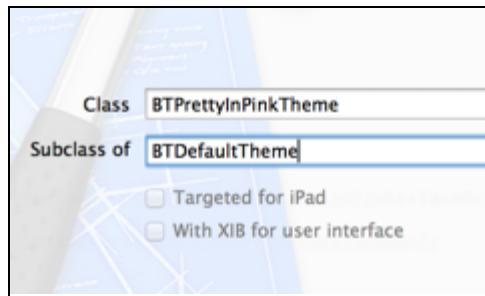
Before you move on though, spend a moment enjoying your fully-customized app:



## Adding your own theme

Now that you've customized the entire application and have the theme manager infrastructure set up and implemented, I'd like to show you how easy it is to add a completely new theme.

Control-click on the **Theme** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **BTPrettyInPinkTheme**, make it a subclass of **BTDefaultTheme**, and click **Next** and finally **Create**.



Paste the following into BTPrettyInPinkTheme.m (kids, do not try to type all of this at home!):

```
#import "BTPrettyInPinkTheme.h"

@implementation BTPrettyInPinkTheme
- (UIColor*)backgroundColor {
 return [UIColor colorWithPatternImage:
 [UIImage imageNamed:@"bg_prettyinpink.png"]];
}

- (UIColor*)upperGradient{
 return [UIColor colorWithRed:0.961 green:0.878
 blue:0.961 alpha:1.000];
}
- (UIColor*)lowerGradient {
 return [UIColor colorWithRed:0.906 green:0.827
 blue:0.906 alpha:1.000];
}
- (UIColor*)seperatorColor {
 return [UIColor colorWithRed:0.871 green:0.741
 blue:0.878 alpha:1.000];
}

- (UIColor*)progressBarTintColor {
 return [UIColor colorWithRed:0.600 green:0.416
 blue:0.612 alpha:1.000];
}
- (UIColor*)progressBarTrackTintColor {
 return [UIColor colorWithRed:0.749 green:0.561
 blue:0.757 alpha:1.000];
}

- (UIColor*)switchOnTintColor {
 return [UIColor colorWithRed:0.749 green:0.561
 blue:0.757 alpha:1.000];
}
```

```
- (UIColor*)switchThumbTintColor {
 return [UIColor colorWithRed:0.918 green:0.839
 blue:0.922 alpha:1.000];
}

- (UIColor*)pageTintColor {
 return [UIColor colorWithRed:0.290 green:0.051
 blue:0.302 alpha:1.000];
}

- (UIColor*)pageCurrentTintColor {
 return [UIColor colorWithRed:0.749 green:0.561
 blue:0.757 alpha:1.000];
}

- (UIImage*)imageForBarButtonNormal {
 return [[UIImage imageNamed:@"barbutton_pretty_uns.png"]
 resizableImageWithCapInsets:
 UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}

- (UIImage*)imageForBarButtonHighlighted {
 return [[UIImage imageNamed:@"barbutton_pretty_sel.png"]
 resizableImageWithCapInsets:
 UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}

- (UIImage*)imageForBarButtonDoneNormal {
 return [[UIImage imageNamed:
 @"barbutton_pretty_done_uns.png"]
 resizableImageWithCapInsets:
 UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}

- (UIImage*)imageForBarButtonDoneHighlighted {
 return [[UIImage imageNamed:
 @"barbutton_pretty_done_sel.png"]
 resizableImageWithCapInsets:
 UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}

- (UIImage*)imageForBarButtonNormalLandscape {
 return [[UIImage imageNamed:
 @"barbutton_pretty_landscape_uns.png"]
 resizableImageWithCapInsets:
 UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}

- (UIImage*)imageForBarButtonHighlightedLandscape {
 return [[UIImage imageNamed:
```

```
 @"barbutton_pretty_landscape_sel.png"]
resizableImageWithCapInsets:
UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}

- (UIImage*)imageForBarButtonDoneNormalLandscape {
 return [[UIImage imageNamed:
 @"barbutton_pretty_done_landscape_uns.png"]
resizableImageWithCapInsets:
UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}

- (UIImage*)imageForBarButtonDoneHighlightedLandscape {
 return [[UIImage imageNamed:
 @"barbutton_pretty_done_landscape_sel.png"]
resizableImageWithCapInsets:
UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}

- (UIImage*)imageForButtonNormal {
 return [[UIImage imageNamed:@"button_pretty_uns.png"]
resizableImageWithCapInsets:
UIEdgeInsetsMake(0.0, 13.0, 0.0, 13.0)];
}

- (UIImage*)imageForButtonHighlighted {
 return [[UIImage imageNamed:@"button_pretty_sel.png"]
resizableImageWithCapInsets:
UIEdgeInsetsMake(0.0, 13.0, 0.0, 13.0)];
}

- (UIImage*)imageForNavigationBar{
 return [[UIImage imageNamed:@"nav_pretty_portrait.png"]
resizableImageWithCapInsets:
UIEdgeInsetsMake(0.0, 12.0, 0.0, 12.0)];
}

- (UIImage*)imageForNavigationBarLandscape{
 return [[UIImage imageNamed:@"nav_pretty_landscape.png"]
resizableImageWithCapInsets:
UIEdgeInsetsMake(0.0, 12.0, 0.0, 11.0)];
}

- (UIImage*)imageForNavigationBarShadow{
 return [UIImage imageNamed:@"topShadow_pretty.png"];
}

- (UIImage*)imageForSwitchOn{
 return [UIImage imageNamed:@"floweron.png"];
}
```

```
- (UIImage*)imageForSwitchOff{
 return [UIImage imageNamed:@"floweroff.png"];
}

- (UIImage*)imageForStepperUnselected{
 return [UIImage imageNamed:@"stepper.pretty_bg_uns.png"];
}
- (UIImage*)imageForStepperSelected{
 return [UIImage imageNamed:@"stepper.pretty_bg_sel.png"];
}
- (UIImage*)imageForStepperDecrement{
 return [UIImage imageNamed:@"stepper.pretty_decrement.png"];
}
- (UIImage*)imageForStepperIncrement{
 return [UIImage imageNamed:@"stepper.pretty_increment.png"];
}
- (UIImage*)imageForStepperDividerUnselected{
 return [UIImage imageNamed:
 @"stepper.pretty_divider_uns.png"];
}
- (UIImage*)imageForStepperDividerSelected{
 return [UIImage imageNamed:
 @"stepper.pretty_divider_sel.png"];
}

- (NSDictionary*)navBarTextDictionary {
return @{
 UITextAttributeFont:
 [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.290 green:0.051
 blue:0.302 alpha:1.000],
 UITextAttributeTextShadowColor:
 [UIColor colorWithRed:0.965 green:0.945
 blue:0.965 alpha:1.000],
 UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
 };
}
- (NSDictionary*)barButtonTextDictionary {
return @{
 UITextAttributeFont:
 [UIFont fontWithName:@"Arial Rounded MT Bold" size:15.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.506 green:0.314
 blue:0.506 alpha:1.000]
 };
}
```

```
 blue:0.510 alpha:1.000],
 UITextAttributeTextShadowColor:
 [UIColor colorWithRed:0.965 green:0.945
 blue:0.965 alpha:1.000],
 UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
 };
}
- (NSDictionary*)buttonTextDictionary {
 return @{
 UITextAttributeFont:
 [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.290 green:0.051
 blue:0.302 alpha:1.000],
 UITextAttributeTextShadowColor:
 [UIColor colorWithRed:0.965 green:0.945
 blue:0.965 alpha:1.000],
 UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
 };
}
- (NSDictionary*)labelTextDictionary {
 return @{
 UITextAttributeFont:
 [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.290 green:0.051
 blue:0.302 alpha:1.000],
 UITextAttributeTextShadowColor:
 [UIColor colorWithRed:0.965 green:0.945
 blue:0.965 alpha:1.000],
 UITextAttributeTextShadowOffset:
 [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
 };
}
- (NSDictionary*)tableViewCellTextDictionary {
 return @{
 UITextAttributeFont:
 [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
 UITextAttributeTextColor:
 [UIColor colorWithRed:0.290 green:0.051
 blue:0.302 alpha:1.000],
 UITextAttributeTextShadowColor:
```

```
[UIColor colorWithRed:0.965 green:0.945
 blue:0.965 alpha:1.000],
UITextAttributeTextShadowOffset:
[NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
};

}

@end
```

Yes, that's the full theme implementation. ☺ If you look at the code, you'll realize that this is identical to what you did for the Forest theme – just the images, colors, and fonts are different.

So you can see that once you have one theme done, it's a fairly simple matter to implement additional themes by changing style elements.



Of course, you still need to add the hooks in the theme controller to allow selection of the theme.

In **BTThemeViewController.m**, add the following import at the top:

```
#import "BTPrettyInPinkTheme.h"
```

And add the following code to `tableView:didSelectRowAtIndexPath:` (right before the `else` statement):

```
else if ([self.themes[idx]
 isEqualToString:@"Pretty in Pink"]) {
 BTThemeManager setSharedTheme:[BTPrettyInPinkTheme new]];
}
```

The final method should look like this:

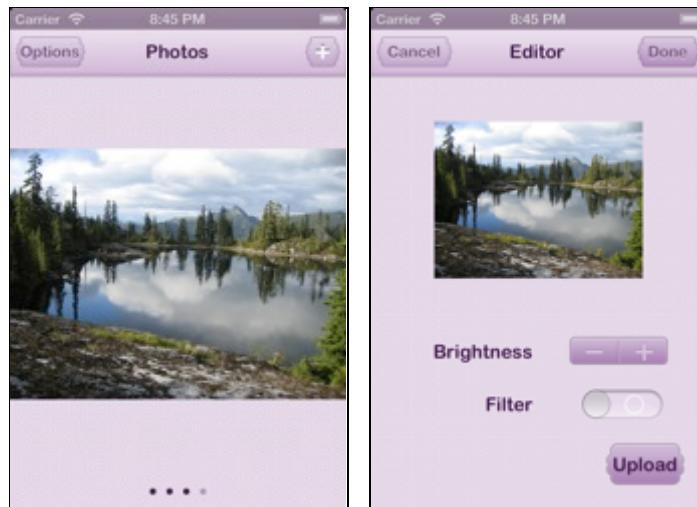
```
-(void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
NSInteger idx = indexPath.row;

if ([self.themes[idx] isEqualToString:@"Forest"]) {
 [BTThemeManager setSharedTheme:[BTForestTheme new]];
}
else if ([self.themes[idx]
 isEqualToString:@"Pretty in Pink"]) {
 [BTThemeManager setSharedTheme:[BTPrettyInPinkTheme new]];
}
else
 [BTThemeManager setSharedTheme:[BTDefaultTheme new]];

[self.navigationController popViewControllerAnimated:YES];
}
```

Build and run, go to Options, and select the "Pretty in Pink" theme.



Oooh, pretty, huh? ☺

## Where to go from here?

At this point you have learned about all the new user interface customization options in iOS 6, as well as a bunch of other cool stuff, like creating a dynamic theme manager and using gradient alyers!

What's most important is that you understand the theme manager and how to create new customizations. As you have seen, after your manager is set up, adding a completely new theme is comparatively simple.

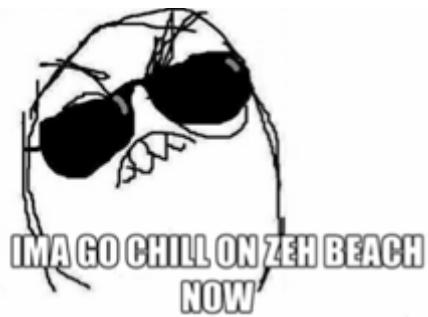
I would encourage you to take one of the themes and:

- Play with it and modify it a bit.

- Use it as a template to create a brand new theme.
- Add a new type of control not covered here and learn how to customize it in your theme.

For more information, I encourage you to watch session #216 from the WWDC 2012 developer videos, "Advanced Appearance Customization on iOS."

Have fun theming!





# Chapter 23: Beginning Automated Testing with XCode

By Charlie Fulton

All developers have to test their software, and many of the smart ones create **test suites** for this purpose. A test suite is a collection of test cases, also known as **unit tests**, targeting small “units” of code, usually a specific method or class.

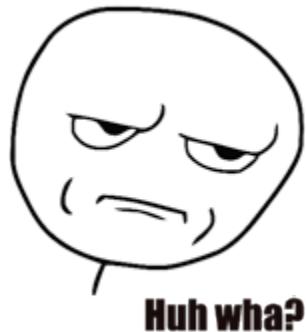
Unit tests allow you to employ test-driven development practices and ensure that your code has fewer bugs. And by creating a unit test for each bug found, you can help make sure the bug never comes back! By unit testing and debugging proactively, you significantly narrow the chances that your app will behave unexpectedly out of the development environment.



But as you know, we’re often rushed to get finished – or we get lazy – so automating your unit tests to run automatically each time you change your code is even better. This is especially important when you’re working as part of a team.

So if you could put in place a system to automatically build, test, and then submit your app to beta testers, would you be interested? If so, then read on!

This bonus chapter and the next will walk you through setting up an automated building and testing system for your iOS apps. You will take an example project, add some unit tests to it, add a remote GitHub ([github.com](https://github.com)) repository, and then set up a continuous integration (CI) server with Jenkins. Jenkins will periodically check your GitHub repository for changes and automatically build, test, and submit your app to TestFlight ([testflightapp.com](https://testflightapp.com))!



If you're new to automated testing, you might be wondering what some (or all) of the above terms are, including GitHub, Jenkins, and TestFlight. We'll go into each of these in detail in this chapter, but for now here are instant micro-introductions to these tools:

- **GitHub** – Free public Git repositories, collaborator management, issue tracking, wikis, downloads, code review, graphs, and much more!
- **Jenkins** – A continuous integration server and scheduler that can watch for changes to Git repositories, kick off builds and tests automatically, and notify you of results.
- **TestFlight** – A service to automate the sending of iOS apps over the air to testers, and perform crash log analysis.

That ought to tide you over for a short while, giving me time to introduce the geektastic example project that will be your vehicle for this educational odyssey – but be ready to learn more about all of this in the coming pages!

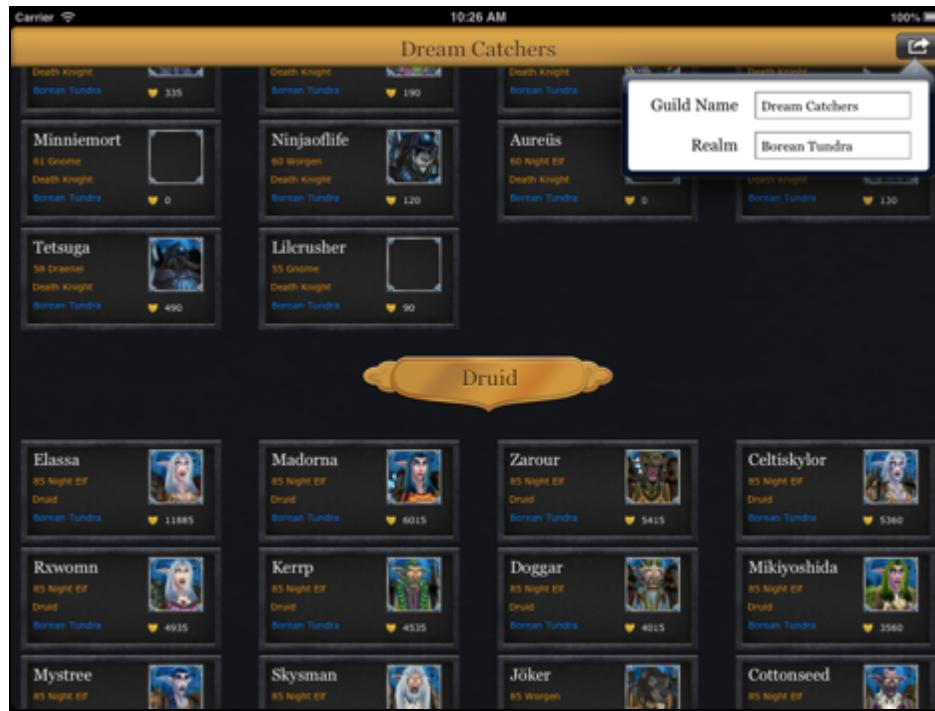
## CHALLENGE ACCEPTED



## Introducing GuildBrowser

In this chapter, you will be creating an automated build and test system for an app called **GuildBrowser**. GuildBrowser is a simple app that lets you browse the members of a guild from the popular game World of Warcraft. You can enter a guild

and a realm name to see all of the members listed in a `UICollectionViewController`, as you can see below:



The app is driven by the freely-available WOW API from Blizzard. You will be using the AFNetworking framework to make RESTful calls to the WOW API, retrieving JSON data that contains guild and character information.

If you never have time to play games, the next best thing is making apps that help with games!

**Note:** For more details about the WOW API, check out Blizzard's GitHub project (<https://github.com/Blizzard/api-wow-docs>), the official WOW API docs (<http://blizzard.github.com/api-wow-docs/>), and the WOW forum devoted to the community platform API (<http://us.battle.net/wow/en/forum/2626217/>).

## Getting started

If you're not using some form of version control for your code these days, then you're definitely not doing it right! Arguably the most popular system right now is **Git**. Linus Torvalds (yes, he who created Linux) designed and developed Git as a better system than its predecessor, SVN, to manage the code for the Linux kernel. Not only is git a great source control system, but it's integrated directly into Xcode, which makes it very convenient to use in iOS development.

Git is a “distributed” version control system, meaning that every Git working directory is its own repository, and not dependent upon a central system.

However, when you want to share your local repository with other developers, you can define a connection in your “remotes” setting. Remote repositories are versions of your project that are usually set up on a central remote server. These can even be shared via a Dropbox folder (although this isn’t particularly recommended).

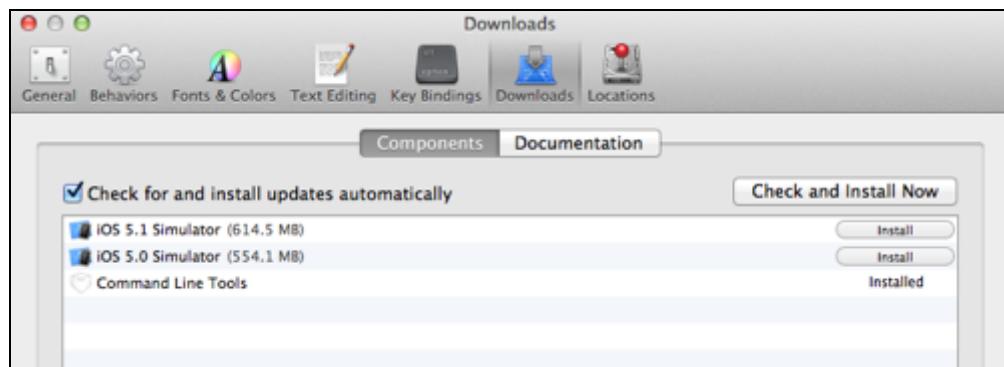
A number of services have emerged in the last few years to make it easier to set up remote repositories for Git, and other distributed version control systems. Github has become one of the best and most popular of these services, and for good reason. It offers free public Git repositories, collaborator management, issue tracking, wikis, downloads, code review, graphs and much more... sorry to sound like an advertisement, but Github is pretty awesome!

**Note:** For a more detailed description of Git and how you can work with it, check out the tutorial on How To Use Git Source Control with Xcode in iOS 6, coming soon to [raywenderlich.com](http://raywenderlich.com).

Your first action in this chapter will be setting up a remote repository (**repo** for short) on GitHub. It will be your remote “origin” repo. It will allow you to work locally as normal, and then when you are ready to sync with teammates or send a new build to testers, you’ll push all of your local commits to this repo.

**Note:** This chapter assumes you already have a GitHub account with SSH key access setup. If you do not, please consult [this guide](https://help.github.com/articles/generating-ssh-keys) (<https://help.github.com/articles/generating-ssh-keys>), which shows you how to get set up by generating and using an SSH key pair. You can also use an https URL with your GitHub username/password, but the guide assumes SSH key access setup.

First, make sure you have the command line tools installed, choose the **Xcode/Preferences/Downloads** tab, and make sure your screen looks similar to this (the important thing is to have **Installed** next to Command Line Tools):



This chapter comes with a starter project, which you can find among the chapter resources. Copy the project to a location of your choice on your hard drive, and open the **GuildBrowser** project in Xcode.

The GuildBrowser app was created with the Single View Application template using ARC, Unit testing, Storyboards, iPad-only, and a local Git repository. Compile and run, and you should see the app showing some characters from the default guild.

Open a browser, go to [github.com](https://github.com) and login to your account, or create an account if don't have one already. Then, click on the **Create a New Repo** button on the top-right corner:



Enter whatever you like for the **Repository name** and **Description** fields – the name does not actually have to correspond with the name of your Xcode project. Then click the **Create repository** button.

**Note:** Do **NOT** check the **Initialize this repository with a README** box. You want to create an empty repo and then push your Xcode project into it.

A screenshot of the GitHub repository creation form. It shows the following fields:

- Owner:** charliefulton
- Repository name:** GuildBrowser
- Description (optional):** An app using blizzard's wow api to show guild members and character items
- Visibility:** Private (selected)
- Initialize this repository with a README:** Unchecked
- Add .gitignore:** None
- Create repository** button

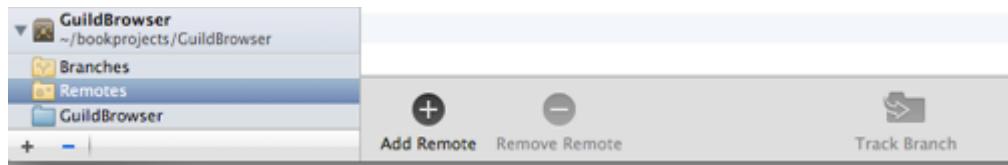
Get a copy of the repository SSH URL by clicking the **copy to clipboard** button or typing ⌘-C (remember to tap the **SSH** button first to switch to the SSH URL).



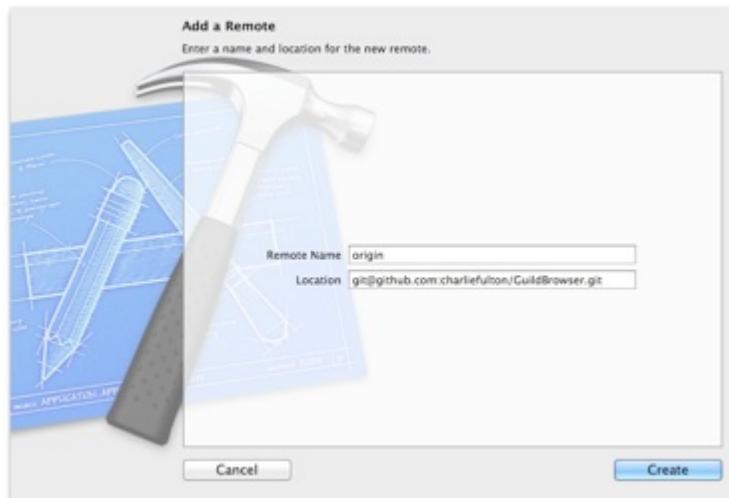
Now open **Organizer (Cmd-shift-2)** and then select the **Repositories** tab.



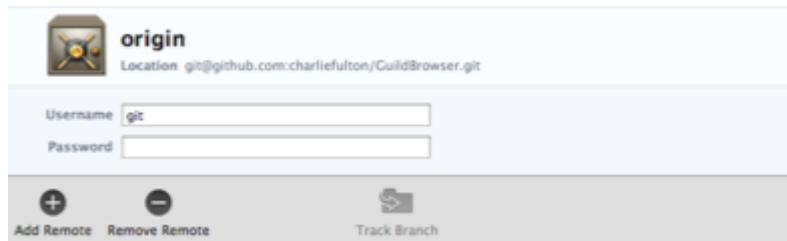
You should see the GuildBrowser project listed at the bottom. Click on the **Remotes** folder and then click the **Add Remote** button.



Enter **origin** for the Remote Name, and paste in your personal GitHub URL that you copied earlier into Location:



Click **Create**, and Organizer should look like the following:



Don't be alarmed that there is no password entered for the origin location. You'll be using Git's SSH access, so no password is needed.

You now have a remote repo set up! You can use these steps for all of your Xcode projects. This is great, but you still have not pushed anything from your local Git repo to the remote Git repo. Let's do that now with Xcode.

Go back to your project and first make sure you have committed all of your local changes. Xcode will not allow you to push to a remote repo until you have done that. Here are the steps:

1. Go to **File\Source Control\Commit** (**⌘-C**) and on the following screen, enter a commit message like "initial commit" and click **Commit**.
2. Now you need to push the local master branch to the remote origin/master branch. Go to **File\Source Control\Push** and click **Push**.
3. Since this is the first time you're pushing to this repo, you should see **(Create)** beside the Remote name, **origin/master**. Tap the **Push** button, and Xcode will handle the rest.



Once the push is complete, go back to your browser, refresh the page, and the changes you committed should now show up in your GitHub repo:

| name                   | age            | message                                | history |
|------------------------|----------------|----------------------------------------|---------|
| GuildBrowser.xcodeproj | 15 minutes ago | added character class [Charlie Fulton] |         |
| GuildBrowser           | 8 minutes ago  | test change [Charlie Fulton]           |         |
| GuildBrowserTests      | 5 days ago     | Initial Commit [Charlie Fulton]        |         |

You should also now see your remote repo and its master branch listed in **Organizer\Repositories\Guild Browser project\Remotes**:



Congratulations, you know how to connect your local Git repo to a remote repo on GitHub and push changes to it!

Now that you have a GitHub repository set up, you could invite others to clone your repository, accept pull requests, or take a look at your code. When someone else pushes their commits to the repo on GitHub **origin/master** repo you go to **File\Source Control\Pull (⌘-X)**, in the popup Remote should be set to **origin/master**, select **Choose** and those changes will be added to your project.

To see what has changed, go to **Organizer\Repositories\GuildBrowser**, expand the arrow and click on the **View Changes** button.



## Continuous integration

**Continuous integration** (CI) is having your code automatically built and tested (and even deployed!) by a server that is continuously watching your revision control system for changes.

CI becomes tremendously important when you have a lot of developers on a project sending in their changes multiple times per day. If any conflicts or errors arise, everyone is notified immediately. Eventually, someone breaks the build!

### Meet Jenkins



[Jenkins](http://jenkins-ci.org) (<http://jenkins-ci.org>) is the open source CI server you're going to use in this chapter to automatically build, test, and deploy code that is pushed to the GitHub remote repo.

You'll spend the rest of this section getting Jenkins up and running.

## Installing Jenkins

There are a few different ways you can install Jenkins on a Mac:

1. You can use the installer from the Jenkins website, which will set up Jenkins to run as a daemon. This is nice because it is configured to use the Max OS X startup system, but it will result in you, the user, having to go through a few more steps in order to get Xcode building to work properly.
2. You can run the WAR file (a Java archive filetype) from the Terminal, which uses the built-in winstone servlet container to run Jenkins.
3. You can deploy the WAR file to an application container you have already set up, such as Tomcat, JBoss, etc.

In this chapter, you are going to run Jenkins from the Terminal, because it's the quickest and easiest way to do it. Download the Jenkins WAR file from <http://mirrors.jenkins-ci.org/war/latest/jenkins.war>. Open a Terminal window, and then run this command:

```
nohup java -jar ~/Downloads/jenkins.war --httpPort=8081 --
ajp13Port=8010 > /tmp/jenkins.log 2>&1 &
```

**Note:** The above assumes that you have the Jenkins WAR file in your Downloads folder. If you downloaded the WAR file to a different location, adjust the path in the command accordingly.

You might find it useful to create an alias for starting up Jenkins. From a Terminal, or text editor showing hidden files, open up **/Users/<username>/.bash\_profile** and enter this line:

```
alias jenkins="nohup java -jar ~/jenkins.war --httpPort=8081 --
ajp13Port=8010 > /tmp/jenkins.log 2>&1 &"
```

Save **.bash\_profile**, open a new Terminal, and now type in **jenkins** to start it up.

This starts the Jenkins server on port 8081. The **nohup** command is a UNIX command that allows a process to keep running after you have logged out or the shell has exited – it stands for “no hangup”.

The above command also sets up a log file to **/tmp/jenkins.log**. I like to configure the ports to avoid any conflicts with servers I have running. To access Jenkins, open the following URL in a browser:

```
http://localhost:8081
```

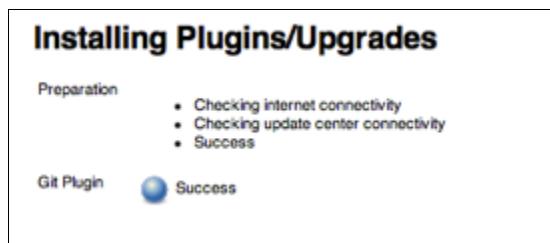
## Configure Jenkins plugins

You need to add a few things to Jenkins before you can start setting up jobs. Open the Jenkins dashboard (via <http://localhost:8081>) and click on the **Manage Jenkins\Manage Plugins\Available** tab. There are a lot of plugins available, so filter the choices down by typing **Git** into the filter search box in the top right.

Here's what you want:



Check the box next to the **Git Plugin** and then click the **Install without restart** button. You should see the following screen when it's done installing:



Now enter **Chuck Norris** in the filter search box, and look for the Chuck Norris quote plugin. While this one is optional for the chapter, I highly recommend you check it out! If you don't install it, he will know! Beware the wrath of Chuck. ☺

**Note:** My friends call me Charlie, but my real name is Charles. My Dad's middle name was Norris. So I had legitimate claims to naming one of my sons Chuck Norris Fulton... but unfortunately **she** wouldn't let me. ☺

## Setting up Jenkins email notification

It would be nice to have an SMTP server so that Jenkins can send you build notification errors. But if you're not running an SMTP server, you can use your Gmail account with Jenkins (or any other SMTP server you prefer).

Go to **Manage Jenkins\Configure System** and in the **Email Notification** section, enter the following settings (use the **Advanced Settings** button to access some of the settings):

- **SMTP Server** : smtp.gmail.com
- **Sender Email address** : <your Gmail address>
- **Use SMTP Authentication**
- **User Name**: <your full Gmail address>
- **Password**: <your Gmail account password>

- **Use SSL**
- **SMTP Port: 465**

E-mail Notification

SMTP server: smtp.gmail.com

Default user e-mail suffix:

Sender E-mail Address: ios@jenkins@gmail.com

Use SMTP Authentication

User Name: ios@jenkins@gmail.com

Password: \*\*\*\*\*

Use SSL:

SMTP Port: 465

Reply-To Address:

Charset: UTF-8

Test configuration by sending test e-mail

Test e-mail recipient: ios@jenkins.com

Email was successfully sent

Test configuration

Now test out email notifications by checking the **Test configuration by sending test-email**, entering an email address, and clicking the **Test configuration** button. Once you've confirmed that the test is successful, remember to tap the **Save** button to save your configuration changes.

Now you can receive emails from Jenkins about build failures.

## Creating a Jenkins job

You are now ready to create a Jenkins job! Open the Jenkins dashboard, click on **New Job**, enter **GuildBrowser** for the job name, choose the **Build a free-style software project** radio button, and finally click the **OK** button. You should now see the job configuration screen.

In the **Source Code Management** section, click the **Git** radio button, and enter your GitHub remote repo (origin/master) URL into the **Repository URL** text field. Your screen should look something like this:

Source Code Management

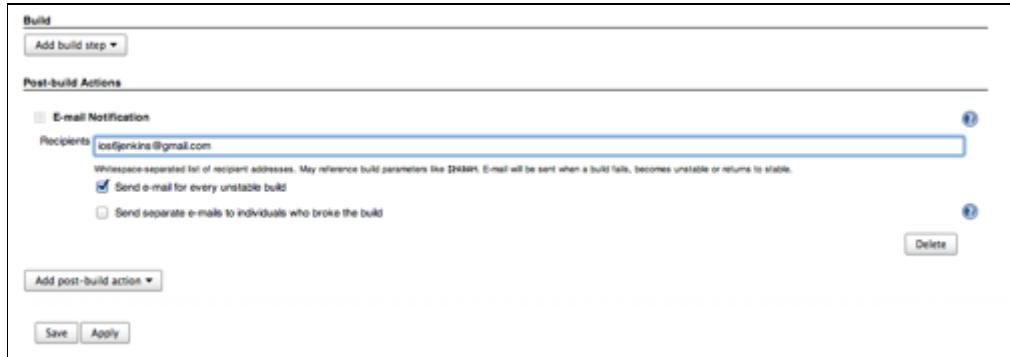
CVS

Git

Repositories

Repository URL: git@github.com:charlefulton/GuildBrowser.git

Go to the **Post-build Actions** section, select **Email Notification** from the **Add post-build action** button, enter the email address(es) for the recipient(s), and finally check **Send email for every unstable build**. You should see the following:

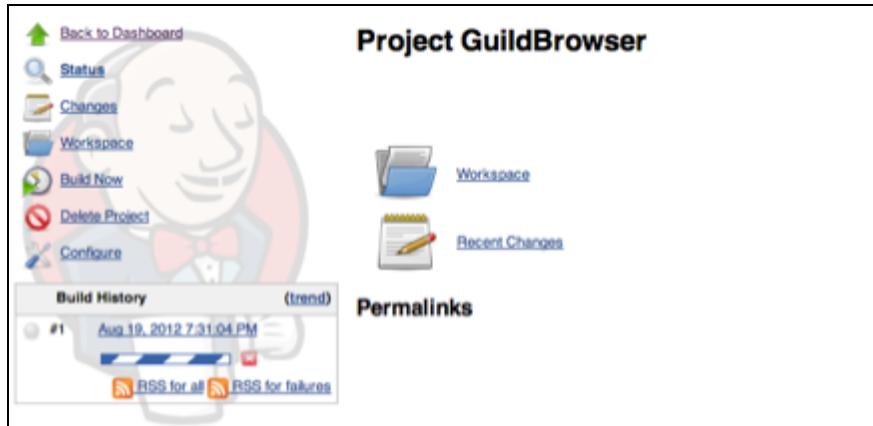


If you installed the Chuck Norris plugin, you can also add a post-build action to **Activate Chuck Norris**. That sounds kinda scary though! Warning, don't break the build!!

**Note:** The programs that Chuck Norris writes don't have version numbers because he only writes them once. If a user reports a bug or has a feature request, they don't live to see the sun set. ☺

Don't forget to click the **Save** button.

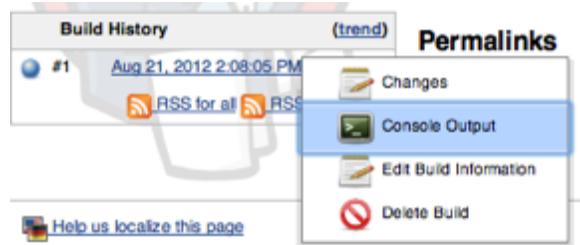
Let's make sure everything is set up correctly with GitHub by clicking the **Build Now** button on the left sidebar. When the job starts, you will see an update in the Build History box:



If everything works correctly, your build #1 should be blue, indicating success:



To see what exactly happened during the build, hover over the build and look at the **Console Output**:



Your console output should look similar to this, showing that the job successfully pulled down the code from the GitHub repo:



So far, you've connected to GitHub with your job and set up email notification. Let's press on and add the interesting bits: testing, building, and uploading your project to TestFlight.

**Note:** A Jenkins pro tip! To quickly get to the job configure screen, hover over the job name on the breadcrumb trail at the top of the screen and select **Configure** from the menu. This is especially helpful when you're looking at console output.



## Command line tools

Before you begin to set up Jenkins to build your iOS app, let's talk a little bit about the command line tools that come with Xcode.

## xcodebuild

`xcodebuild` is the command line tool for building, archiving, and querying your project or workspace. You're going to use it to build a project from Jenkins. You will soon see all the ways it can be put to work.

## xcode-select

This sets the Xcode.app that is used by tools in **/usr/bin**. Open a Terminal and run the following command, and note the output:

```
xcode-select -print-path
/Applications/Xcode.app/Contents/Developer
```

This shows which Xcode.app will be used when running the tools found in **/usr/bin** like `xcodebuild`, `xcrun`, `opendiff`, `instruments`, `agvtool`, and `git`.

When you have multiple versions of Xcode installed, like a beta release, it's really important for your CI server to have this specifically defined. You do that by setting the `DEVELOPER_DIR` environment variable and running all of your build commands through the `xcrun` command that comes with Xcode.

## xcrun

This finds or runs tools inside Xcode.app. Apple recommends running all command line tools (such as the `git` command line tool) through this command.

**Note:** It is possible to change the Xcode.app that is configured. At the time of this writing, I had both 4.4.1 and 4.5 beta4 installed.

The following sequence of commands shows switching between Xcode 4.5 beta4 and Xcode version 4.4.1.

```
xcode-select -print-path
```

Output: /Applications/Xcode45-DP4.app/Contents/Developer

```
xcodebuild -version
```

Output: Xcode 4.5

Build version 4G144I

```
sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer/
```

```
xcode-select -print-path
```

Output: /Applications/Xcode.app/Contents/Developer

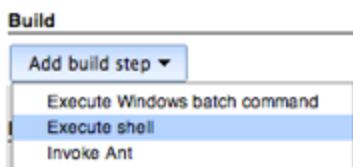
```
xcodebuild -version
```

Output: Xcode 4.4.1

Build version 4F1003

## Automating xcodebuild from Jenkins

Now that you know a little more about `xcodebuild` and the other command line tools, let's update your Jenkins job to do the build. Go to the **Jenkins Dashboard\GuildBrowser job\Configure**. In the **Build** section, click on the **Add build step** dropdown select **Execute shell**.



Enter the following code into the Command window:

```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer/
xcrun xcodebuild clean build
```

Your command window should look like this:



Now click **Save**.

Let's test the job again by clicking on **Build Now**. When the job starts executing, pay attention! You might see the following screen, asking you if you want to allow Jenkins access to run codesign so that it can sign your code. Select **Always Allow**.



If you missed the prompt (because you were chasing your two year-old who was running off with your iPhone, for example), you will see this message in your failed job's console output:

```
Command /usr/bin/codesign failed with exit code 1
** BUILD FAILED **

The following build commands failed:
CodeSign build/Release-iphoneos/GuildBroswer.app
(1 failure)
Build step 'Execute shell' marked build as failure
```

If this happens, simply try running the job again, but don't miss the prompt (or let your two year-old run away with your iPhone) this time. ☺

Assuming you didn't miss the prompt, then in your output you will see a message similar to this:

```
/Users/charlie/.jenkins/workspace/GuildBrowser/build/Release-
iphoneos/GuildBrowser.app
Unable to validate your application. - (null)
```

This means the build was successful! You can safely ignore the "unable to validate" message – it's a known bug that we were able to verify at WWDC this year.

**Note:** There is a Jenkins plugin for building iOS apps named [Xcodeplugin](#). It works great, but I think it's best to understand exactly what is going on when building your app. You can achieve the same results as the plugin with some simple build scripts of your own.

What's more, becoming dependent on the plugin carries some risks. If Apple changes how things work and that breaks the plugin, and it's no longer updated, you will still be able to adapt if you know how things work.

# Unit testing

Unit testing ensures that the output of your code remains unchanged as you are developing it. After establishing expectations of what a class will do, you write unit tests to verify these expectations remain true. Unit testing also allows you to make changes incrementally, seeing results step-by-step, which makes it easier to develop new features and make big changes.

Some other benefits of including unit tests in your projects are:

- **Reliability** – With a good set of tests, you will most likely have fewer bugs. This is especially true with code that was developed with tests from the beginning, verifying every change and new bit of code along the way.
- **Code confidence** – It's a lot less nerve-wracking to go in and refactor a massive amount of code with unit tests in place, knowing that your code should still be producing the same results and passing tests.
- **Stability** – When bugs are found, new tests can be added ensuring the same bugs don't harass you again.

Xcode has built-in support for unit tests via the SenTestingKit framework. When writing unit tests, there are two terms to be aware of:

- **Test case** – The smallest component of a unit test is the test case. It is what verifies the expectations of what your unit of code should produce. For example, a test case might focus on testing a single method, or a small group of methods in your class. In Xcode, all test cases must start with the name **test**. You will see an example of this below.
- **Test Suite** – A group of test cases. This is usually a group of test cases against a particular class. In Xcode, there is a template for creating a test suite named **Objective-C test case class**. It produces a class that is a subclass of **SenTestCase**.

There are two general approaches to setting up unit tests: you can either test from the bottom up (testing each method/class individually), or the top down (testing functionality of the app as a whole). Both approaches are important – it's usually good to have a combination of both.

In this chapter, you are going to test from the bottom up, focusing on a few of the model classes. In the next chapter, you'll try your hand at testing from the top down, from the User Interface to the model code via UI Automation.

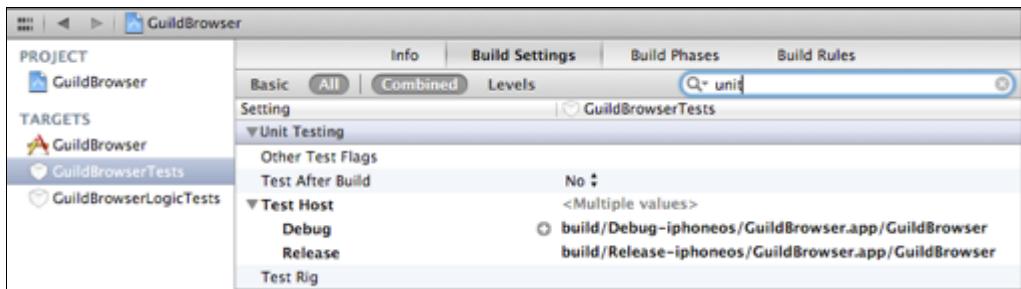
## Application vs. logic test targets

Next let's take a look at the two different kinds of unit tests available in Xcode.

## Application unit tests

An application unit test basically means any code that needs to use something in UIKit, or really needs to run in the context of a host app. This is the type of unit test target you get by default when creating a new project and including unit tests.

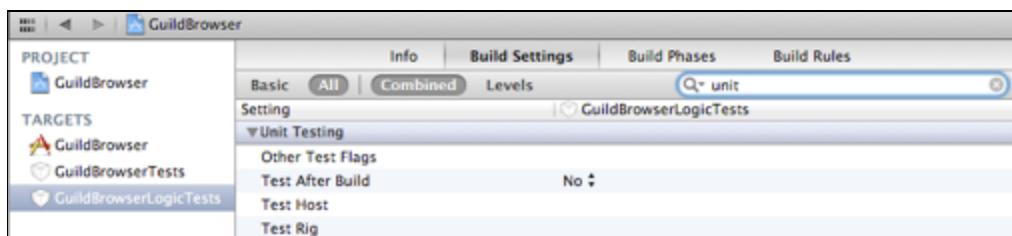
All of the code in this target will be run within your app bundle. You can see the difference by looking in the build settings for each target and checking the settings for **Test Host** and **Bundle Loader**:



## Logic unit tests

For now, this is the only type of unit test you can run on your units of code from the command line. This code is tested in isolation from the app.

These are unit tests that test your own code, basically anything that doesn't need to run in the Simulator. It's great for testing custom logic, network access code, or your data model classes.



## Running tests from command line notes – no touching!

Unfortunately, as of Xcode 4.5 we are still not "officially" allowed to run application unit tests via the `xcodebuild` command. This is very frustrating, because you obviously can run application unit tests from within Xcode. For this reason, you will create a special logic test target in the next section.

Some creative developers have made patches to the testing scripts inside of the Xcode.app contents. For this chapter and the next, you are going to stick to running your logic tests from Jenkins, and application tests from Xcode. If you're curious about the patches, take a peek at these files:

```
/Applications/Xcode.app/Contents/Developer/Tools/RunUnitTests
```

```
/Applications/Xcode.app/Contents/Developer/Tools/RunPlatformUnitTests.include
```

This is the script that runs by default when you run unit tests via ⌘-U:

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/Tools/RunPlatformUnitTests
```

Here is the error message you'll get, and the reason we can't run application unit tests in the Simulator from the command line (remember that this works fine from within Xcode):

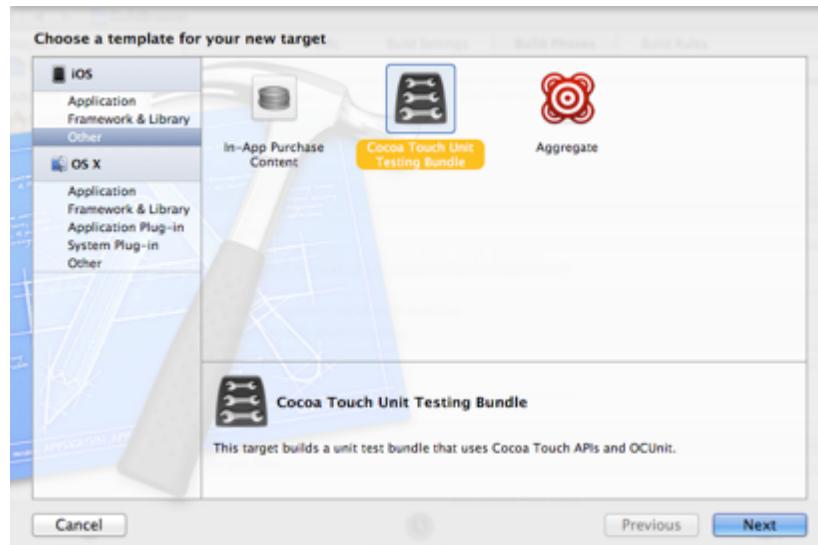
```
RunTestsForApplication() {
 Warning ${LINENO} "Skipping tests; the iPhoneSimulator platform
 does not currently support application-hosted tests (TEST_HOST
 set)."
}
```

Some clever hacks have gotten application tests to work, but the scripts have changed from Xcode 4.4 to 4.5, so do so at your own risk.

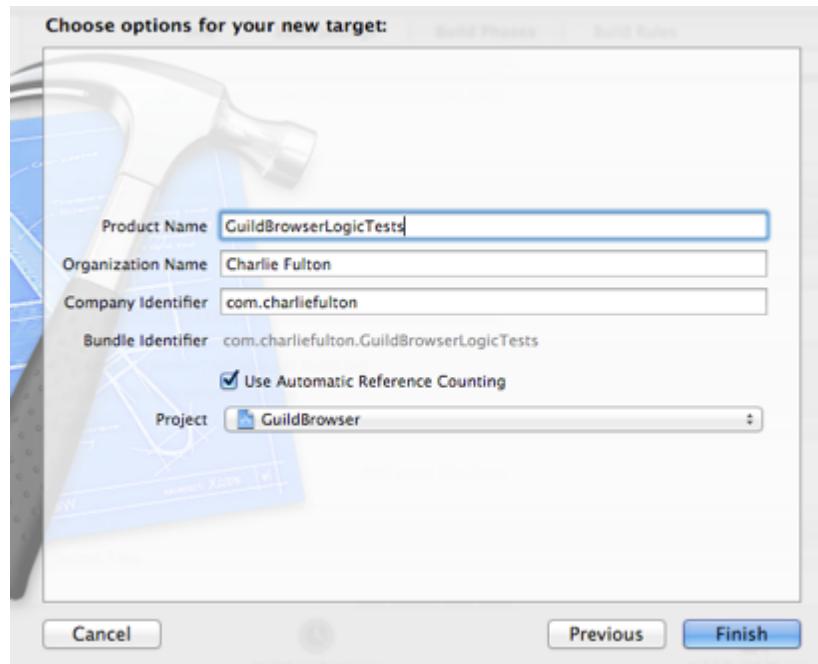
## Adding a logic unit test target

You are now going to create a new test target that can be called from your build script.

In Xcode, Go to **File\New\Target...\iOS\Other\Cocoa Touch Unit Testing Bundle** and then click **Next**.

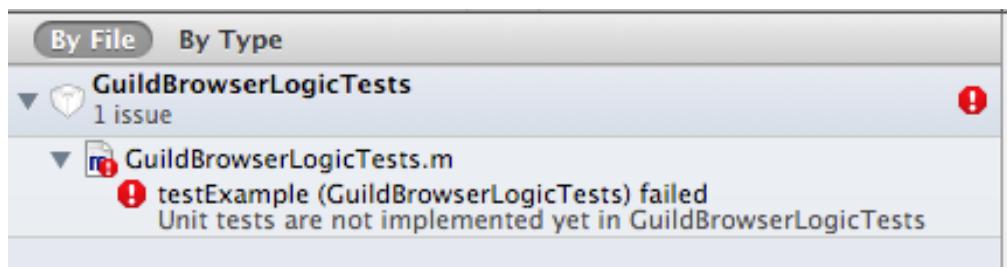


Enter a name that ends with **Tests**, for example **GuildBrowserLogicTests**, make sure that **Use Automatic Reference Counting** is checked, and then click **Finish**.



You will now have a new scheme with the same name. To test that everything is working, switch to the new scheme (using the scheme selector on the Xcode toolbar next to the Run and Stop buttons) and go to **Product\Test** ( $\text{⌘}-\text{U}$ ).

You should see that the build succeeded, but the test failed. ☹



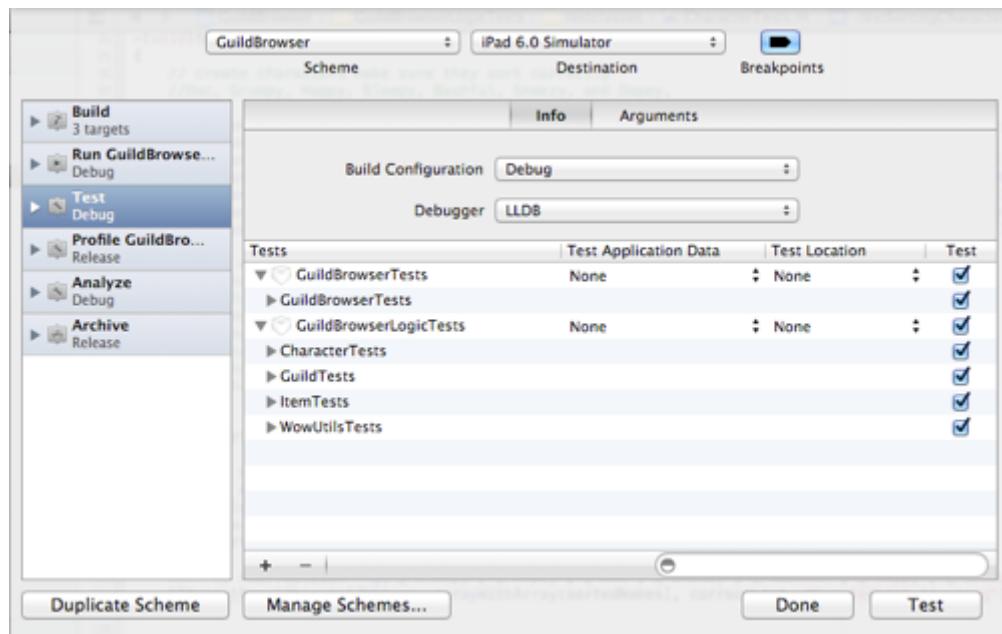
As the error message indicates, this is because the unit tests have not been implemented yet. Let's fix that!

First, delete the default test class by highlighting **GuildBrowserLogicTest.h** and **GuildBrowserLogicTest.m**, tapping **Delete**, and then selecting **Move to Trash**.

So that you don't have to remember to switch schemes, let's modify the main app scheme, **GuildBrowser**, to include your new target when running **Product\Test**.

Switch to the **GuildBrowser** scheme and then edit the scheme via **Product>Edit Scheme** (or you can simply Alt-click on the Run button). In the scheme editor, highlight the **Test** configuration, click on the **+** button to add a new target, select **GuildBrowserLogicTests** in the window and then click **Add**.

Your screen should look similar to this:

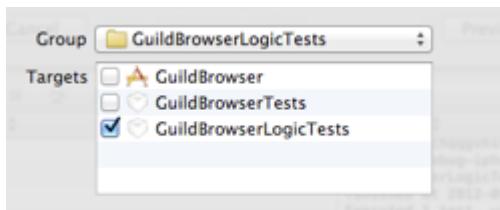


## Creating a unit test class

Now let's add a simple unit test class for testing the `WowUtils` class. `WowUtils` looks up the string values for the Character class, Character race, and Item quality from the web service JSON data.

It's best to create a new unit test class for each new class you want to test. Xcode has an **Objective-C test case class** template just for this purpose.

In the Xcode project navigator, right-click on the **GuildBrowserLogicTests** group, choose **New File\ Cocoa Touch\ Objective-C test case class**, click **Next**, enter **WowUtilsTests** for the class name, click **Next** again, make sure only the **GuildBrowserLogicTests** target is selected, and then click **Create**.



Now you have a **test suite** to which you can add some **test cases**. Let's create your first test case. This test case will make sure that you get the correct response when looking up a character's class.

Here are the methods you are testing from **WowUtils.h**:

```
+ (NSString *)classFromCharacterType:(CharacterClassType)type;
+ (NSString *)raceFromRaceType:(CharacterRaceType)type;
```

```
+ (NSString *)qualityFromQualityType:(ItemQuality)quality;
```

**Note:** These methods are expected to get the correct name based on the data retrieved from Blizzard's web service. Below you can see the output used by the `WowUtils` class (Chrome is a great tool for inspecting JSON output from a web service):



```
{
 - classes: [
 - {
 id: 3,
 mask: 4,
 powerType: "focus",
 name: "Hunter"
 },
 - {
 id: 4,
 mask: 8,
 powerType: "energy",
 name: "Rogue"
 },
 - {
 id: 1,
 mask: 1,
 powerType: "rage",
 name: "Warrior"
 },
 - {
 id: 2,
 mask: 2,
 powerType: "mana",
 name: "Paladin"
 },
 - {
 id: 7,
 mask: 64,
 powerType: "mana",
 name: "Shaman"
 },
 - {
 id: 8,
 mask: 128,
 powerType: "mana",
 name: "Mage"
 }
]
}
```

Replace the contents of `WowUtilsTests.m` with:

```
#import "WowUtilsTests.h"
#import "WowUtils.h"

@implementation WowUtilsTests

// 1
-(void)testCharacterClassNameLookup
{
 // 2
 STAssertEqualObjects(@"Warrior",
 [WowUtils classFromCharacterType:1],
 @"ClassType should be Warrior");
 // 3
```

```
 STAssertFalse([@"Mage" isEqualToString:[WoWUtils
classFromCharacterType:2]],
nil);

// 4
STAssertTrue([@"Paladin" isEqualToString:[WoWUtils
classFromCharacterType:2]],
nil);
// add the rest as an exercise
}

- (void)testRaceTypeLookup
{
 STAssertEqualObjects(@"Human", [WoWUtils raceFromRaceType:1],
nil);
 STAssertEqualObjects(@"Orc", [WoWUtils raceFromRaceType:2],
nil);
 STAssertFalse([@"Night Elf" isEqualToString:[WoWUtils
raceFromRaceType:45]],nil);
 // add the rest as an exercise
}

- (void)testQualityLookup
{
 STAssertEquals(@"Grey", [WoWUtils qualityFromQualityType:1],
nil);
 STAssertFalse([@"Purple" isEqualToString:[WoWUtils
qualityFromQualityType:10]],nil);
 // add the rest as an exercise
}

@end
```

Let's break this down bit-by-bit.

1. As I stated earlier, all test cases must start with the name **test**.
2. The expectation is that the `WoWUtils` class will give you the correct class name, given an ID. The way you verify this expectation is with the `STAssert*` macros. Here you are using the `STAssertEqualObjects` macro.  
The expected result, "**Warrior**", is compared to the result from the `WoWUtils` method; if the test fails, you log the message "**ClassType should be Warrior**".
3. It's always good to include a "failing test" in your test case. This is a test where the result is expected to fail. Again, you are using one of the assertion macros from the `SenTestingKit` – this time `STAssertFalse`.

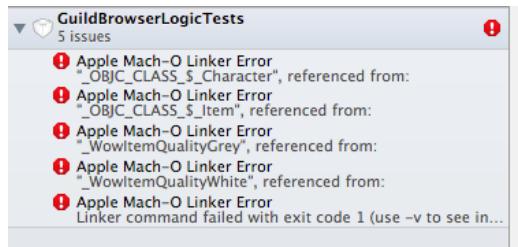
The expected result, “**Mage**”, is compared to the result from the `WowUtils` method; if the test fails, you use the default message, since you passed in `nil` in this example.

4. Finally, you have another example test macro to use.

**Note:** For a complete list of the testing macros, check out the [Unit-Test Result Macro Reference](#) (<http://bit.ly/Tsi9ES>) from the Apple Developer library.

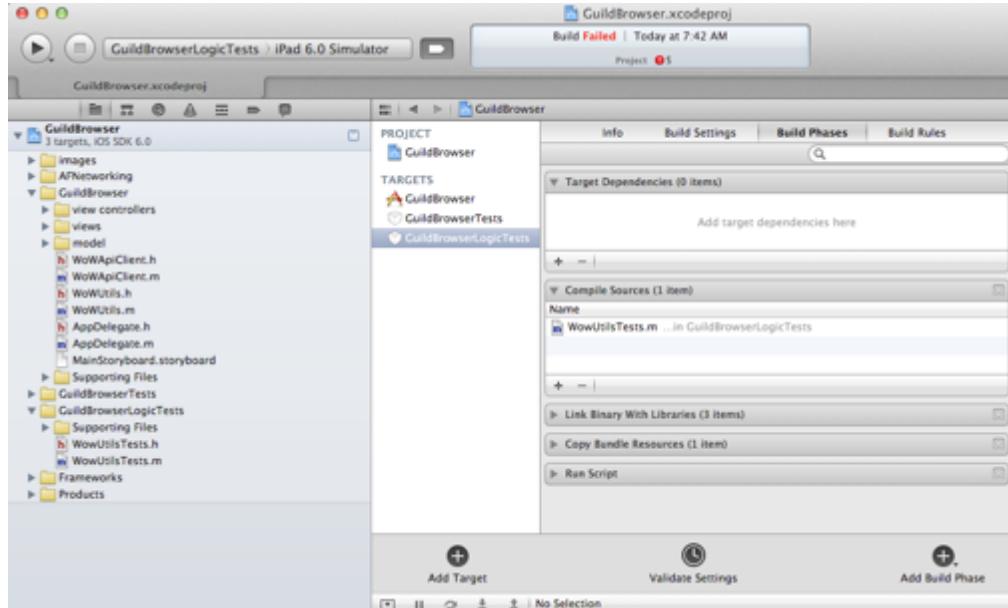
Now you can run your new test suite that presently contains one test case. Go to **Product\Test** (⌘-U).

Doh! You’ll see a compile error:

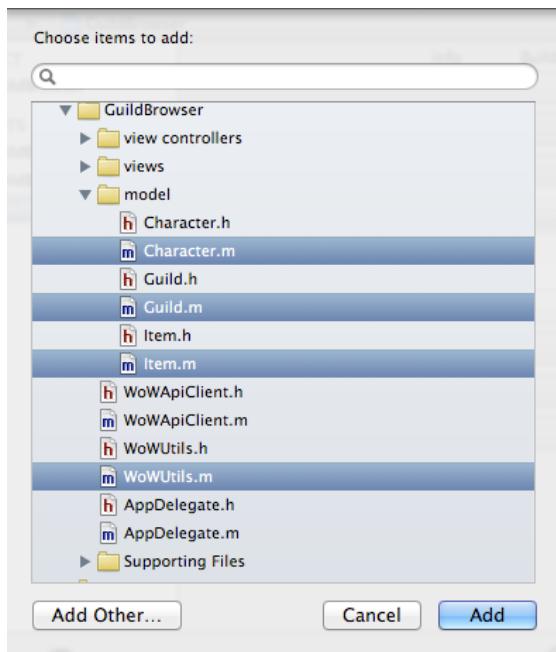


Since you added a new target, you need to let it know about the classes you’re trying to test. Every target has its own set of source files available to it. You need to add the source files manually, since you’re running a logic test target without the bundle and test host set.

1. Switch to the project navigator – if it’s not open, use the **View\Navigators>Show Project Navigator** menu item (⌘-1).
2. Click on the project root to bring up the **Project** and **Targets** editor in the main editor area.
3. Select the **Build Phases** tab along the top of Xcode’s main editing pane.
4. Now click on the **GuildBrowserLogicTests** target in the **TARGETS** section. You should see this:

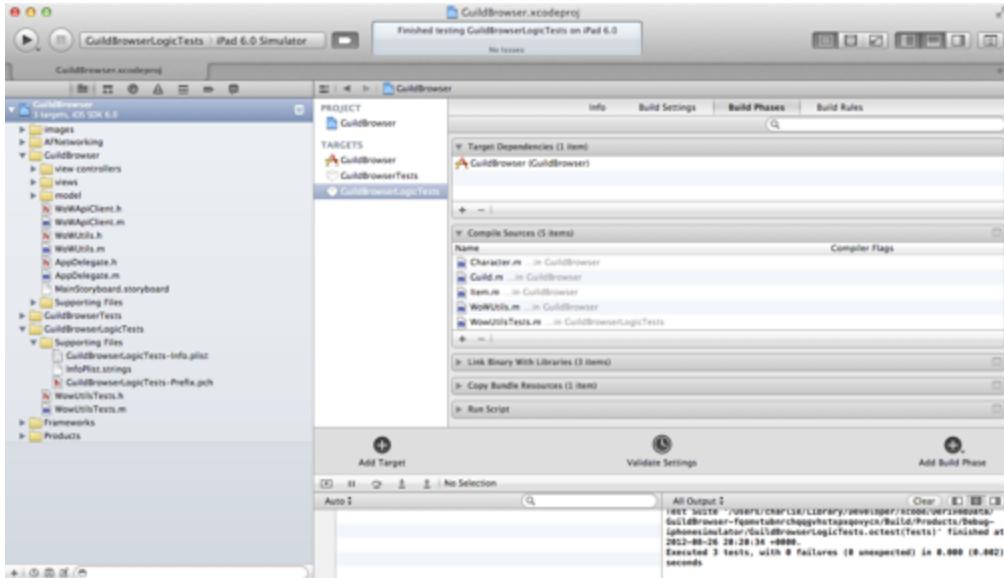


5. Expand the disclosure arrow in the **Compile Sources** section, and click on the **+** button. In the popup window, choose the following classes: **Character.m**, **Guild.m**, **Item.m**, and **WowUtils.m**. Then click **Add**.



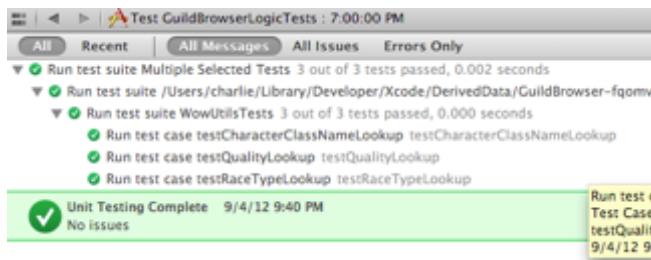
6. Now add the app target as a dependency so that it will get built before your test target runs. Expand the disclosure arrow in the **Target Dependencies** section, click the **+** button, choose the **GuildBrowser** app target and click **Add**.

When these steps are complete, you should see the following:



Now run Product\Test ( $\text{⌘}-\text{U}$ ) and the test should succeed. You can see the output from all the tests by switching to the Log Navigator (go to View\Navigators>Show Log Navigator ( $\text{⌘}-7$ )).

You should see this when you click on the last build on the left sidebar:



Make sure to commit and push your latest changes to Github.

## Testing a class with local JSON data

Let's take a look at the `Character` class and add a test suite for it. Ideally, you would have added these tests as you were developing the class.

What if you wanted to test creating your `Character` class from local data? How would you do that? What if you wanted to share such data with each test case?

So far, you haven't used two special methods that are available in your unit tests: **setUp** and **tearDown**. Every time you run your unit tests, each test case is invoked independently. Before each test case runs, the `setUp` method is called, and afterwards the `tearDown` method is called. This is how you can share code between each test case.

When building an app that relies on data from web services, I find it very helpful to create tests using data that matches the payload from the services. Quite often in a

project, you will only be working on the client side and waiting for the services from another developer. You can agree on a format and can “stub” out the data in a JSON file. For this app, I downloaded some character and guild data to create tests for the model classes so I could flesh those out before working on the networking code.

First add your test data. Extract the **TestData.zip** file found in the chapter resources and drag the resulting folder into your Xcode project. Make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, and that the **GuildBrowserLogicTests** target is checked, and then click **Finish**.

Now add an Objective-C test case class for your character test cases. In the Xcode project navigator, right-click on the **GuildBrowserLogicTests** group, choose **New File\ Cocoa Touch\ Objective-C test case class**, click **Next**, enter **CharacterTests** for the class name, click **Next** again, make sure only the **GuildBrowserLogicTests** target is selected, and then click **Create**.

Replace the contents of **CharacterTests.m** with:

```
#import "CharacterTests.h"
#import "Character.h"
#import "Item.h"

@implementation CharacterTests
{
 // 1
 NSDictionary *_characterDetailJson;
}

// 2
-(void)setUp
{
 // 3
 NSURL *dataServiceURL = [[NSBundle bundleForClass:self.class]
 URLForResource:@"character"
 withExtension:@"json"];

 // 4
 NSData *sampleData = [NSData
 dataWithContentsOfURL:dataServiceURL];
 NSError *error;

 // 5
 id json = [NSJSONSerialization JSONObjectWithData:sampleData
 options:kNilOptions
 error:&error];
}
```

```
 STAssertNotNil(json, @"invalid test data");

 _characterDetailJson = json;
}

-(void)tearDown
{
 // 6
 _characterDetailJson = nil;
}

@end
```

Hammer time, break it down! ☺

1. Remember that test classes can have instance variables, just like any other Objective-C class. Here you create the instance variable `_characterDetailJson` to store your sample JSON data.
2. Remember that `setUp` is called before **each** test case. This is useful because you only have to code up the loading once, and can manipulate this data however you wish in each test case.
3. To correctly load the data file, remember this is running as a test bundle. You need to send `self.class` to the `NSBundle` method for finding bundled resources.
4. Create `NSData` from the loaded resource.
5. Now create the JSON data and store it in your instance variable.
6. Remember that `tearDown` is called after **each** test case. This is a great spot to clean up.

Make sure everything is loading up correctly by running Product\Test (⌘-U). OK, now you have your test class set up to load some sample data.

After the `tearDown` method, add the following code:

```
// 1
- (void)testCreateCharacterFromDetailJson
{
 // 2
 Character *testGuy1 = [[Character alloc]
 initWithCharacterDetailData:_characterDetailJson];
 STAssertNotNil(testGuy1, @"Could not create character from
 detail json");

 // 3
```

```
Character *testGuy2 = [[Character alloc]
initWithCharacterDetailData:nil];
STAssertNotNil(testGuy2, @"Could not create character from nil
data");
}
```

Break it down!

1. Here you are creating test cases for the `Character` class designated initializer method, which takes an `NSDictionary` from the JSON data and sets up the properties in the class. This might seem trivial, but remember that when developing the app, it's best to add the tests while you are incrementally developing the class.
2. Here you are just validating that `initWithCharacterDetailData` does indeed return something, using another `STAssert` macro to make sure it's not `nil`.
3. This one is more of a negative test, verifying that you still got a `Character` back even though you passed in a `nil` `NSDictionary` of data.

Run `Product\Test` (`⌘-U`) to make sure your tests are still passing!

After the `testCreateCharacterFromDetailJson` method in **CharacterTests.m**, add the following:

```
// 1
-(void)testCreateCharacterFromDetailJsonProps
{
 STAssertEqualObjects(_testGuy.thumbnail, @"borean-
tundra/171/40508075-avatar.jpg", @"thumbnail url is wrong");
 STAssertEqualObjects(_testGuy.name, @"Hagrel", @"name is
wrong");
 STAssertEqualObjects(_testGuy.battleGroup, @"Emberstorm",
@"battlegroup is wrong");
 STAssertEqualObjects(_testGuy.realm, @"Borean Tundra", @"realm
is wrong");
 STAssertEqualObjects(_testGuy.achievementPoints, @3130,
@"achievement points is wrong");
 STAssertEqualObjects(_testGuy.level, @85, @"level is wrong");

 STAssertEqualObjects(_testGuy.classType, @"Warrior", @"class
type is wrong");
 STAssertEqualObjects(_testGuy.race, @"Human", @"race is
wrong");
 STAssertEqualObjects(_testGuy.gender, @"Male", @"gener is
wrong");
 STAssertEqualObjects(_testGuy.averageItemLevel, @379, @"avg
item level is wrong");
```

```
 STAssertEqualObjects(_testGuy.averageItemLevelEquipped, @355,
 @"avg item level is wrong");
}

// 2
-(void)testCreateCharacterFromDetailJsonValidateItems
{
 STAssertEqualObjects(_testGuy.neckItem.name, @"Stoneheart
Choker", @"name is wrong");
 STAssertEqualObjects(_testGuy.wristItem.name, @"Vicious Pyrium
Bracers", @"name is wrong");
 STAssertEqualObjects(_testGuy.waistItem.name, @"Girdle of the
Queen's Champion", @"name is wrong");
 STAssertEqualObjects(_testGuy.handsItem.name, @"Time Strand
Gauntlets", @"name is wrong");
 STAssertEqualObjects(_testGuy.shoulderItem.name, @"Temporal
Pauldrons", @"name is wrong");
 STAssertEqualObjects(_testGuy.chestItem.name, @"Ruthless
Gladiator's Plate Chestpiece", @"name is wrong");
 STAssertEqualObjects(_testGuy.fingerItem1.name, @"Thrall's
Gratitude", @"name is wrong");
 STAssertEqualObjects(_testGuy.fingerItem2.name, @"Breathstealer
Band", @"name is wrong");
 STAssertEqualObjects(_testGuy.shirtItem.name, @"Black
Swashbuckler's Shirt", @"name is wrong");
 STAssertEqualObjects(_testGuy.tabardItem.name, @"Tabard of the
Wildhammer Clan", @"name is wrong");
 STAssertEqualObjects(_testGuy.headItem.name, @"Vicious Pyrium
Helm", @"neck name is wrong");
 STAssertEqualObjects(_testGuy.backItem.name, @"Cloak of the
Royal Protector", @"neck name is wrong");
 STAssertEqualObjects(_testGuy.legsItem.name, @"Bloodhoof
Legguards", @"neck name is wrong");
 STAssertEqualObjects(_testGuy.feetItem.name, @"Treads of the
Past", @"neck name is wrong");
 STAssertEqualObjects(_testGuy.mainHandItem.name, @"Axe of the
Tauren Chieftains", @"neck name is wrong");
 STAssertEqualObjects(_testGuy.offHandItem.name, nil, @"offhand
should be nil");
 STAssertEqualObjects(_testGuy.trinketItem1.name, @"Rosary of
Light", @"neck name is wrong");
 STAssertEqualObjects(_testGuy.trinketItem2.name, @"Bone-Link
Fetish", @"neck name is wrong");
 STAssertEqualObjects(_testGuy.rangedItem.name, @"Ironfeather
Longbow", @"neck name is wrong");
}
```

```
}
```

You need to make sure that the properties correctly match up with the JSON data that you loaded initially. Just a few comments on what's going on here:

1. This tests the information shown in the main screen Character cell.
2. This tests the information shown in the `CharacterDetailViewController` when you click on a Character cell from the main screen.

To have a little fun, “forget on purpose” to run `Product\Test (⌘-U)`, and commit and push your changes. You will see what your little friend Jenkins will find, when you update your Jenkins job script to include the tests.

## Finalizing your Jenkins job

First off, make sure to commit all changes and push them to your origin/master branch on GitHub.

As a reminder, you will need to do the following:

1. Go to **File\Source Control\Commit (⌃⌘-C)**, then on the following screen, enter a commit message like “added test target” and click **Commit**.
2. Push the local master branch to the remote origin/master branch. Go to **File\Source Control\Push** and click **Push**.

Now edit your Jenkins job again to include the tests you just set up. Go to the **Jenkins Dashboard\GuildBrowser job\Configure**. In the **Build** section, replace the existing code with this:

```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer/

xcodebuild -target GuildBrowserLogicTests \
-sdk iphonesimulator \
-configuration Debug \
TEST_AFTER_BUILD=YES \
clean build
```

Click **Save** and then **Build Now**.

Ummm, RUN! You broke the build and Chuck Norris is going to find you! The gloves are on! This is serious, man. No one can escape him. ☺



So what happened? You could scour the output logs of the build that put you on Mr. Norris's radar, or... you could set up Jenkins to give you some test results reporting! I don't know about you, but if I'm on that radar I want to be off it FAST! You should have also received an email telling you that you broke the build. Let's get some reporting, stat!

## Getting a unit test report

It would be quite tedious to pour through the logs after each build, and it sure would be nice if there were a handy report you could look at to see what passed and what failed.

Well, it turns out there is a script that does exactly that! Christian Hedin has written an awesome Ruby script to turn the OCUnit output into JUnit-style reports. You can find it on GitHub at <https://github.com/ciryon/OCUnit2JUnit>.

A copy of the Ruby script is included in the resources for this chapter. Work quickly; remember whose radar you are on!

Copy the **ocunit2junit.rb** file to somewhere that Jenkins can access it – I placed mine in **/usr/local/bin**. Make a note of the location, and use it below when updating the build job.

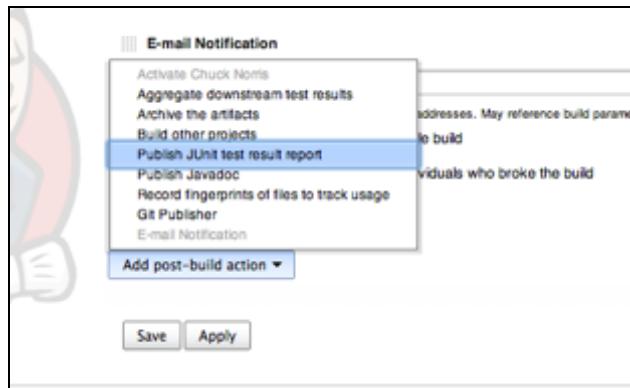
In the **GuildBrowser Jenkins job**, update the shell script to the following:

```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer/

xcodebuild -target GuildBrowserLogicTests \
-sdk iphonesimulator \
-configuration Debug \
TEST_AFTER_BUILD=YES \
clean build | /usr/local/bin/ocunit2junit.rb
```

The only new bit is on the final line, where the output from the build is piped through to the Ruby script for processing.

Now you need to add another Post-Build Action to your job to capture these reports. At the bottom of the configure screen, click on the **Add post-build action** button and choose **Publish JUnit test result report**.



Enter **test-reports/\*.xml** in the **Test report XMLs** text field.

Click **Save** and then **Build Now**. When the job is finished and you go to the main project area for the GuildBrowser job, you should see a **Latest Test Result** link. Work quickly; get on that link!

Now it's really obvious what has angered "He who must not be named."

| Test Name                                                     | Duration | Age |
|---------------------------------------------------------------|----------|-----|
| CharacterTests.testCreateCharacterFromDetailJsonValidateItems | 1 ms     | 3   |

| Package | Duration | Fail | SKIP | Skip | Total | Time |
|---------|----------|------|------|------|-------|------|
| com     | 31 ms    | 1    | +1   | 0    | 14    | +11  |

It looks like a bug has been found by the `CharacterTests` class. Let's investigate this further by clicking on the link to the test name:

**Error Message**

```
"Vicious Pyrium Bracers" should be equal to "Girdle of the Queen"s Champion" name is wrong
```

**Stacktrace**

```
/Users/charlie/.jenkins/workspace/GuildBrowser/GuildBrowserLogicTests/testclasses/CharacterTests.m:58
```

OK, let's go check out the test code. First look at line 76 from **CharacterTests.m**:

```
STAssertEqualObjects(hagrel.waistItem.name, @"Girdle of the Queen's Champion", @"name is wrong");
```

Interesting! Open the file **character.json** to see what you have; maybe your data was wrong, but surely not your code!

**Note:** When working with JSON, I highly recommend verifying your test data or any JSON using the awesome website [jsonlint.com](http://jsonlint.com). Not only does it validate your JSON, but it will format it too!

Remember character.json represents what's returned from Blizzard's real character web service. You'll find this snippet:

```
{
 "thumbnail": "borean-tundra/171/40508075-avatar.jpg",
 "class": 1,
 "items": {
 ...
 "wrist": {
 "icon": "inv_bracer_plate_dungeonplate_c_04",
 "tooltipParams": {
 "extraSocket": true,
 "enchant": 4089
 },
 "name": "Vicious Pyrium Bracers",
 "id": 75124,
 "quality": 3
 },
 "waist": {
 "icon": "inv_belt_plate_dungeonplate_c_06",
 "tooltipParams": {
 "gem0": 52231
 },
 "name": "Girdle of the Queen's Champion",
 "id": 72832,
 "quality": 4
 },
 ...
 }
}
```

Hmmm, the plot thickens. Your test found "Vicious Pyrium Bracers", the name of the wrist item, but was looking for "Girdle of the Queen's Champion", the name of the waist item. Now it's looking more like a bug!

You can see in the test in **CharacterTest.m** that you create a character like so:

```
Character *hagrel = [[Character alloc]
initWithCharacterDetailData:characterDetailJson];
```

Open **Character.m** and let's take a look at **initWithCharacterDetailData:**. Can you spot the bug?

```
_wristItem = [Item initWithData:data[@"items"][@"wrist"]];
_waistItem = [Item initWithData:data[@"items"][@"wrist"]];
```

You were using the wrong key for the `waistItem` property. Fix that by changing the last line to:

```
_waistItem = [Item initWithData:data[@"items"][@"waist"]];
```

OK, now commit your changes and push them to GitHub. Go to the Jenkins project and click on **Build Now**.

Test result: no failures! Chuck gives that a thumbs up – what a huge relief!



Drill down into the report by clicking on **Latest Test Result\root** and you will see all the tests that were run. By drilling further into the **CharacterTests**, you can see that you fixed the issue (look closely at the status on line three ☺):

| Test Result : CharacterTests                     |              |            |
|--------------------------------------------------|--------------|------------|
| 0 failures (-1)                                  | 6 tests (±0) | Took 6 ms. |
|                                                  |              |            |
| All Tests                                        |              |            |
| Test name                                        | Duration     | Status     |
| testCreateCharacterFromDetailJson                | 1 ms         | Passed     |
| testCreateCharacterFromDetailJsonProps           | 1 ms         | Passed     |
| testCreateCharacterFromDetailUserValidationItems | 1 ms         | Fixed      |
| testGettingProfileImageUriFromCharacter          | 1 ms         | Passed     |
| testGettingThumbnailImageUriFromCharacter        | 1 ms         | Passed     |
| testSortingCharacters                            | 1 ms         | Passed     |

## Polling for changes

Now let's set up the Jenkins project to look for changes every 10 minutes, and if it finds any, to run the build. Edit your Jenkins job again by going to the **Jenkins Dashboard\GuildBrowser job\Configure**. In the **Build Triggers** section, check the **Poll SCM** box. In the schedule text area, enter:

```
*/10 * * * *
```

Click **Save**. Now builds will happen if there is new code that has been pushed to the origin/master repo. Of course, you can still manually build like you've been doing with the Build Now button.

## Automate archiving

Let's update the build script to archive your app after successfully running the test script. Edit your Jenkins job again by, you guessed it, going to the **Jenkins Dashboard\GuildBrowser job\Configure**.

You are now going to add another shell step, which will happen *after* the test script step. In the **Build** section, click on the **Add build step** dropdown and select **Execute shell**.

Enter the following, replacing the `CODE_SIGN_IDENTITY` with your distribution certificate name:

```
tests passed archive app

export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer

/usr/bin/xcrun xcodebuild -scheme GuildBrowser clean archive \
CODE_SIGN_IDENTITY="iPhone Distribution: Charles Fulton"
```

Your build section should now look similar to this:

The screenshot shows the Jenkins build configuration for the 'GuildBrowser' project. It contains two 'Execute shell' build steps:

```
export DEVELOPER_DIR=/Applications/Xcode65-DR4.app/Contents/Developer
xcodebuild -target GuildBrowserLogicTests
-sdk iphonesimulator
-configuration Debug
TEST_AFTER_BUILD=YES
clean build | /usr/local/bin/ecunit2junit.rb
tests passed archive app
export DEVELOPER_DIR=/Applications/Xcode65-DR4.app/Contents/Developer
/usr/bin/xcrun xcodebuild -scheme GuildBrowser clean archive
CODE_SIGN_IDENTITY="iPhone Distribution: Charles Fulton"
```

Below the build steps is a 'Add build step ▾' button.

If you save your changes and build again via Jenkins, you should now see the latest archive in the Xcode organizer for this project. In Xcode, open **Window\Organizer** and switch to the **Archives** tab, and you should see the **GuildBrowser** project listed on the left.

When you click on the project, you will see all of your archived builds. This is great, because now you will know that you are building and testing the same archive that will be submitted to the App Store!

**Note:** If the archiving fails, you will not see an archive listed in the Xcode Organizer, nor will you see any immediate indication from Jenkins as to the archive failure. You would need to check the build logs to see if the archiving actually succeeded.

Usually, the archive process fails because the `CODE_SIGN_IDENTITY` is not correctly specified, or because it doesn't match the Bundle ID for the project. So if you run into any archival failures, those would be the items to check. One solution here would be to set the `CODE_SIGN_IDENTITY` to iPhone Distribution, since that will match the default distribution profile.

Also remember that if you make any project changes to fix the above, you need to commit to Git and push to GitHub. Otherwise, Jenkins won't pick up your changes for the next build. ☺

Next you will add sending this archived build to TestFlight, and keeping track of the artifact (which is Jenkins terminology for the results of the build) in Jenkins.



## Uploading the archive to TestFlight

One of the best things about the iOS development community is the variety of awesome frameworks and services that have emerged over the past few years.

Back in the day, it was quite an effort to get a beta build to your testers. You would have to get your IPA file to them by email, have them drag that to iTunes, then connect their device to sync up with iTunes just to get it on their device. You also had to send them an email asking for the UDID of their device, scribble that down or copy it up to create the new provisioning profile, then create the new build. It was a nightmare to keep track of which device belonged to what user, what iOS version they were running, etc.

Enter TestFlight! This is a website that makes distribution and testing of beta versions of apps a breeze. ☺ Before TestFlight, the only way to distribute builds to your beta testers was via ad hoc builds. The ad hoc builds still remain, since TestFlight works within the ad hoc mechanism, but it makes distribution and management of these builds much simpler.

Testflight will also allow you to set up their TestFlight SDK packaging in your app for crash log analysis, usage analysis, and more!

We are going to focus on the auto upload and distribution pieces that TestFlight offers.

### PackageApplication

Here is a sweet little Perl script included by Apple in the Xcode.app bundle. You can take a peek at it (no touching!) by opening:

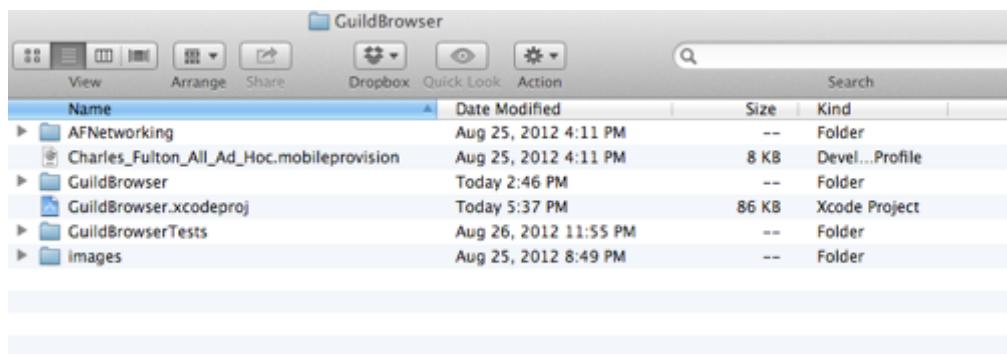
```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/PackageApplication
```

This is the tool that will allow you to do the following from the latest archived build, the one you just created in the previous step. You will be modifying your Jenkins archive step to include:

- Creating the GuildBrowser.app bundle;
- Embedding your ad hoc provisioning profile;
- Codesign with your distribution certificate.

To make sure you're ready, download your latest **ad hoc provisioning profile** from the iOS Provisioning Portal and add it to the top level of your project folder.

After downloading it, your project should look like the image below. Notice my ad hoc provisioning profile named **Charles\_Fulton\_All\_Ad\_Hoc.mobileprovision**:



**Note:** Your .mobileprovision file can be located anywhere. You just have to make sure to give it the full absolute path – no relative paths. For example:

**~/Library/MobileDevice/Provisioning\ Profiles/**

Must be:

**/Users/charlie/Library/MobileDevice/Provisioning\ Profiles/**

I like to keep mine in Git, so that when new devices are added, I just check in a new provisioning profile. Then I can do a manual build in Jenkins and it's ready to go.

Make sure to **commit** and **push** to GitHub after adding the file, so Jenkins can see it.

If you are using Xcode to commit to Git, then note that you would need to add the mobile provisioning profile to your project before Xcode sees the new file. Otherwise, you will not be able to commit it to Git. If you use the command-line Git tools or a separate Git client, this issue should not arise.

Let's edit your Jenkins job again. Go to the **Jenkins Dashboard\GuildBrowser job\Configure**. Add a new Build step of type **execute shell**:

```
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer

#
Setup
#
1
PROJECT="GuildBrowser"
SIGNING_IDENTITY="iPhone Distribution: Charles Fulton"
PROVISIONING_PROFILE="${WORKSPACE}/Charles_Fulton_All_Ad_Hoc.mobileprovision"

2
this is the latest archive from previous build step
ARCHIVE=$(ls -dt
~/Library/Developer/Xcode/Archives/*/${PROJECT}*.xcarchive|head -1)
#
3
IPA_DIR="${WORKSPACE}"
DSYM="${ARCHIVE}/dSYMs/${PROJECT}.app.dSYM"
APP="${ARCHIVE}/Products/Applications/${PROJECT}.app"

#
PackageApplication
#
package up the latest archived build
/bin/rm -f "${IPA_DIR}/${PROJECT}.ipa"

4
/usr/bin/xcrun -sdk iphoneos PackageApplication \
-o "${IPA_DIR}/${PROJECT}.ipa" \
-verbose "${APP}" \
-sign "${SIGNING_IDENTITY}" \
--embed "${PROVISIONING_PROFILE}"

zip and ship
/bin/rm -f "${IPA_DIR}/${PROJECT}.dSYM.zip"

5
/usr/bin/zip -r "${IPA_DIR}/${PROJECT}.dSYM.zip" "${DSYM}"
```

There is a lot going on in this build script. You know what that means.



Break it down!

1. These are the settings for what certificate and provisioning profile to use when creating your IPA file. You should change the `SIGNING_IDENTITY` and `PROVISIONING_PROFILE` to use your ad hoc distribution profile.
2. This bit of shell trickery finds the latest archive build location. This gives you a sneaky way to get the latest archive result directory. To make more sense of this one, open a terminal and run the command:

```
ls -dt
~/Library/Developer/Xcode/Archives/*/*GuildBrowser*.xcarchive|head
-1
```

You should see output like this:

```
/Users/charlie/Library/Developer/Xcode/Archives/2012-08-
27/GuildBrowser 8-27-12 10.37 AM.xcarchive/
```

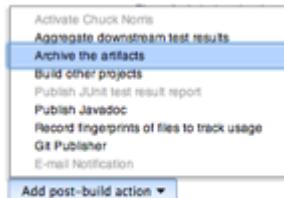
3. You can now use the `ARCHIVE` variable to create the variables `APP` and `DSYM` saving the absolute paths to send to `PackageApplication`. Take a peek inside by trying this command:

```
ls -l "/Users/charlie/Library/Developer/Xcode/Archives/2012-08-
27/GuildBrowser 8-27-12 10.37
AM.xcarchive/Products/Applications/GuildBrowser.app"
```

4. Here you are calling the `PackageApplication` script. Notice that Jenkins gives you some nice environment variables in `$WORKSPACE`. The `$WORKSPACE` variable lets you get an absolute path to the Jenkins job. You can now create artifacts in Jenkins of exactly what gets sent to your users.
5. Compress the `dsyms` from the archive. `dsyms` are used to symbolicate crash logs so that you can find out which source file, method, line, etc. had an issue instead of getting memory addresses that would mean nothing to you.

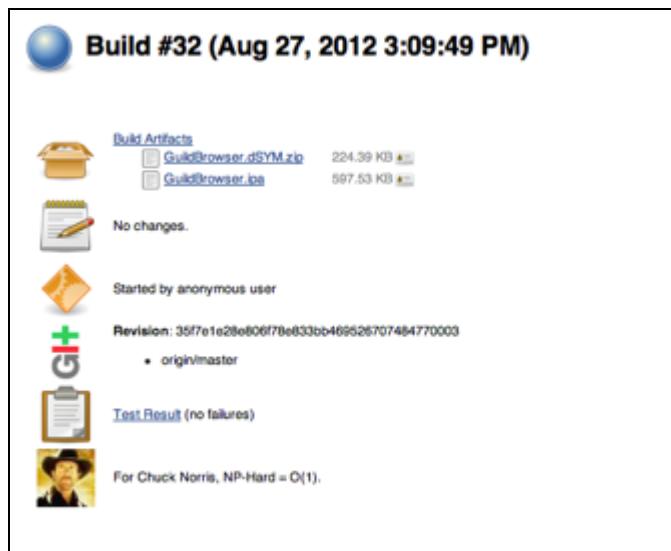
Before saving and building the updated script, let's add a step to create artifacts of all successfully-created .apps and dSYMs.

Go to the **Post-build Actions** section, and select **Archive the artifacts** from the **Add post-build action** menu.



Enter **\*.ipa, \*.dSYM.zip** in the **files to archive** text field.

Click **Save** and select **Build Now**. Once the build completes, you should see this:



If something fails at this point, it usually is because the code signing information wasn't correct, or because the ad hoc provisioning profile isn't in the project root folder. So check the build logs to see what is going on.

### Mission completion

OK, let's send those artifacts to TestFlight and notify your users of the new build.

**Note:** This section assumes you already have a TestFlight ([testflightapp.com](http://testflightapp.com)) account. You will need your TestFlight team and API tokens.

You can get your API token here: <https://testflightapp.com/account/#api>

Your team token can be found by clicking on the team info button while logged into TestFlight.

Edit your Jenkins job again by going to the **Jenkins Dashboard\GuildBrowser job\Configure**. You will be editing the script you added in the previous step.

Add this code, filled out with your own TestFlight info, after the `export DEVELOP_DIR` line:

```
testflight stuff
API_TOKEN=<YOUR API TOKEN>
TEAM_TOKEN=<YOUR TEAM TOKEN>
```

Add this to the end of the existing script:

```
#
Send to TestFlight
#
/usr/bin/curl "http://testflightapp.com/api/builds.json" \
-F file=@"${IPA_DIR}/${PROJECT}.ipa" \
-F dsym=@"${IPA_DIR}/${PROJECT}.dSYM.zip" \
-F api_token="${API_TOKEN}" \
-F team_token="${TEAM_TOKEN}" \
-F notes="Build ${BUILD_NUMBER} uploaded automatically from
Xcode. Tested by Chuck Norris" \
-F notify=True \
-F distribution_lists='all'

echo "Successfully sent to TestFlight"
```

Click **Save** and do another **Build Now**.

Now when the job completes, your build should have been sent to TestFlight! And your users should have received an email telling them that a new build is available. This will allow them to install your app right from the email and begin testing for you!

## Where to go from here?

You should now be equipped to set up an automated building, testing, and distribution system for all of your iOS apps!

Let's recap what you did in this chapter:

- First you learned how to set up a remote repo on Github, giving you a spot to share and test your code.
- After that, you took a look at continuous integration with Jenkins, and created a nice build script step-by-step, first building, then testing, and finally uploading your archived app to Testflight.

- You also looked at how to include a “bottom up” approach to unit testing your code. If you’re interested in learning more about unit testing in iOS, I highly recommend the book *Test-Driven iOS Development* by Graham Lee. I also encourage all of you to submit a radar to apple to make it easier to run the application unit tests from scripts, without hacks!

In the next chapter, you are going to take the same app and do some “top down” unit testing by creating a cool little testing robot. This robot will use instruments and a UI Automation script to drive some UI interactions in the GuildBrowser app.

☺

# Chapter 24: Intermediate Automated Testing with Xcode

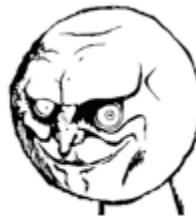
By Charlie Fulton

In this chapter, you are going to continue to learn how to set up automated testing for your iOS apps, this time taking a top-down approach – testing the functionality of the app as a whole, from the user's point of view.

To do this, you will build a robot! It won't be quite as cool as NASA's Mars rover, but it will be similar in one sense: you will be running a script that sends commands to your app automatically, piloting your user interface! You'll create your robot script using Apple's UI Automation framework in the Instruments package.

But first, as an hors d'oeuvres, I'm going to show you how to test more complex objects that have dependencies (such as objects that access the network) by using "mock objects" with the OCMock framework.

Between your robot script and these mock objects, this chapter will put an army of minions at your disposal to test your apps. In combination with the online tools you learned about in the last chapter, you'll have all you need to build a robust automated testing system!



**I am the master of build & run..**

## Unit testing with OCMock

Mock objects are used often in unit testing – they provide a way to simulate one object in order to test the behavior of another. For example, you might control the output of a network client class in order to test how another class uses this output data.

One simple way of thinking about mock objects is to visualize a crash test dummy!



Instead of hopping in a car yourself and driving towards a wall to see what will happen (which might be quite painful!), you send in the dummy to see how it responds instead.

OCMock is a framework that makes creating mock objects for unit testing nice and easy (and not painful at all!). This Objective-C implementation takes advantage of the dynamism of the language, allowing your mock objects to look almost identical to the real ones.

It's not necessary to use fancy tricks with selectors and so forth – you just set up and call whatever method(s) you are simulating. The idea is, when an object uses another object to perform work that takes some time (such as fetching a file from the network), you replace it with a fake object that returns the data immediately. This speeds up your testing process and better isolates the code.

Let's try it out!

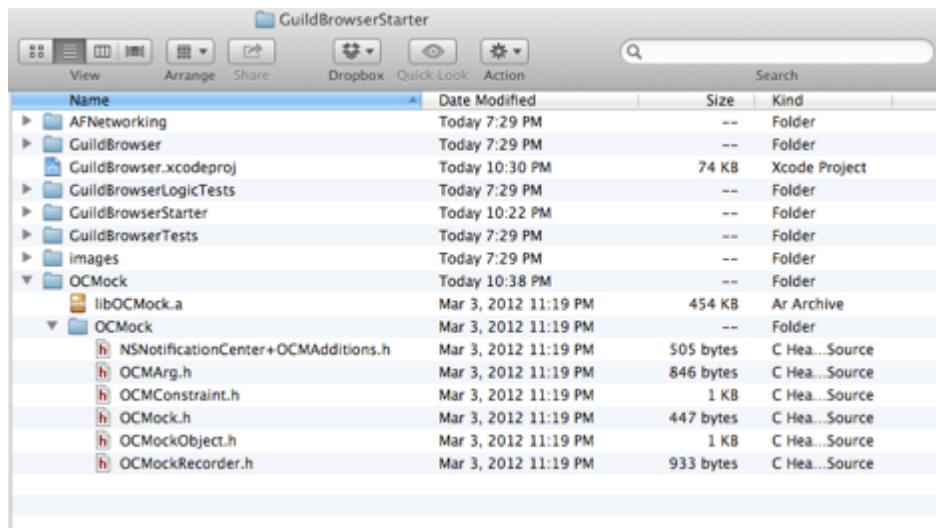
## Adding OCMock to your project

You are going to continue to test the GuildBrower app from the last chapter. This app retrieves remote data, in the form of JSON output, via an API client, and then creates custom `UICollectionViewCells` representing a member of a World of Warcraft guild. All of the guild members are laid out in the UI using a `UICollectionViewController`. The user also has the ability to enter in a guild or realm name to change the data being displayed. All of these networking calls are made asynchronously and use blocks to show the new data after the calls to the RESTful web services are complete.

First, let's get the OCMock framework added to the project. This chapter comes with a starter project that should be among the chapter's resources. Copy the

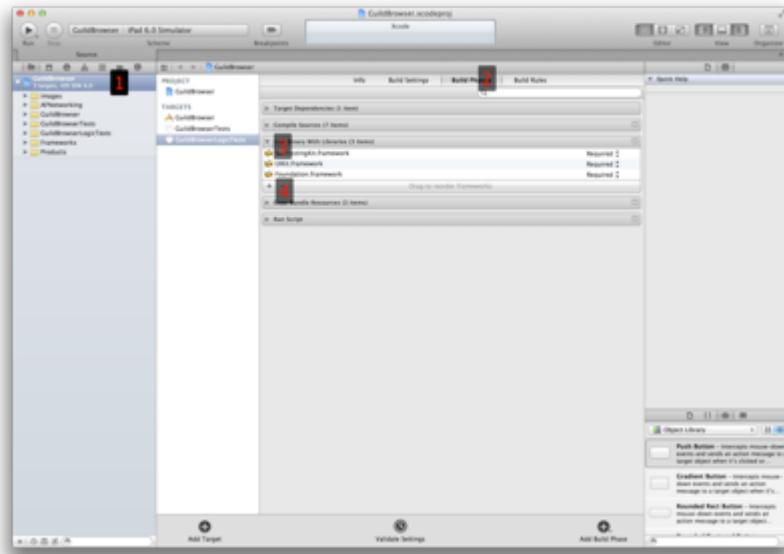
project to a location of your choice on your hard drive, and open the **GuildBrowser** project in Xcode.

Also in the resources for this chapter, you should find a ZIP file named **OCMock.zip**. Extract the file and drag the resulting folder onto the top level of your project folder in Finder, not directly onto Xcode. The reason for this is you just want to add the library file to the Xcode project. Your folder structure should now look like this:



Now add **libOCMock.a** to your Xcode project. OCMock ships as a static library and to use this library in your unit tests, you need to link it against the **GuildBrowserLogicTests** target.

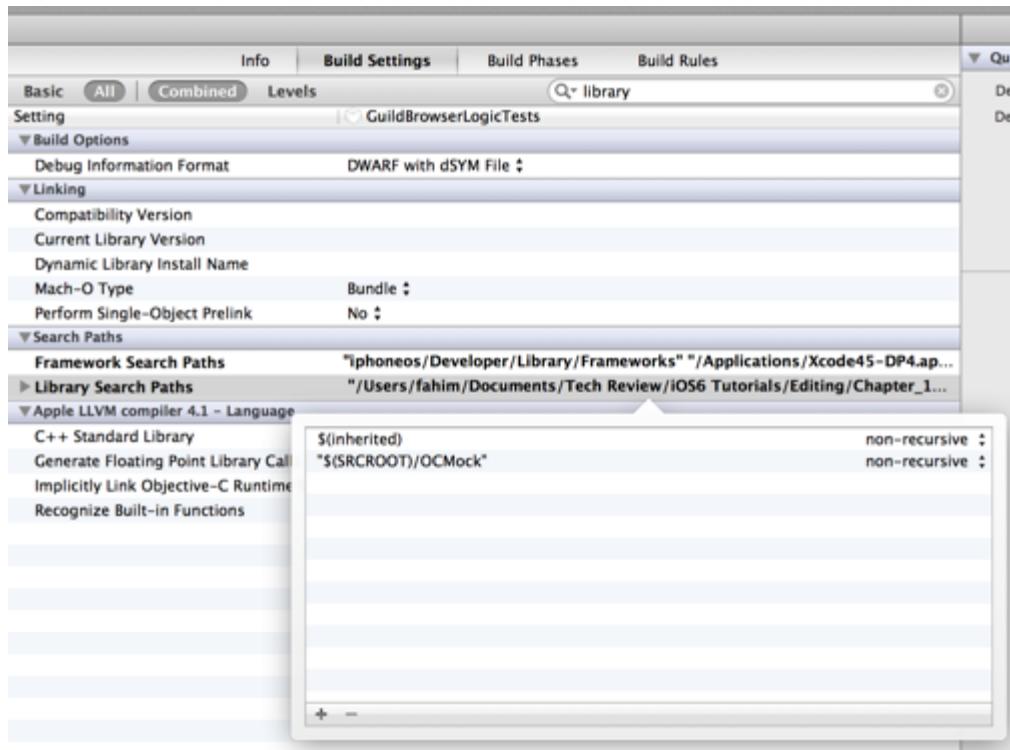
To do this, click on the project root in the project navigator and then select the **GuildBrowserLogicTests** target. Switch to the **Build Phases** tab, expand the **Link Binary With Libraries** section and then click the plus (+) button.



Now click the **Add Other** button and browse to your project folder on your hard disk. Select **OCMock\libOCMock.a** and click **Open**. This will automatically set up the search path to find the library.

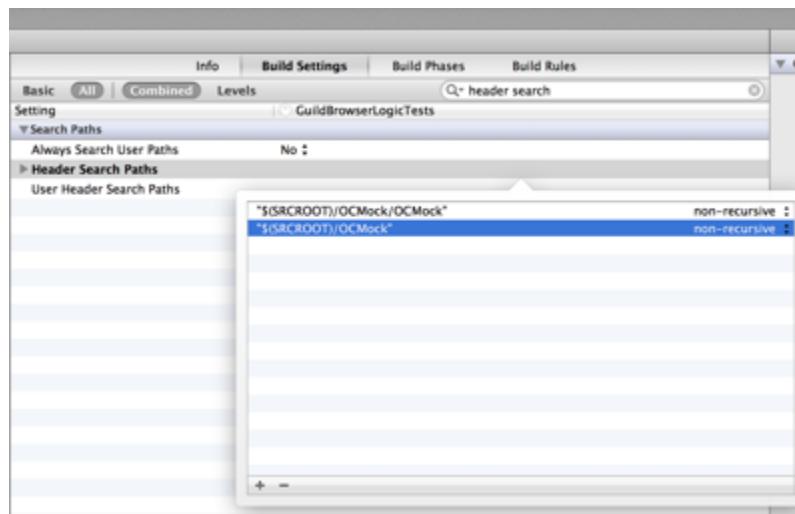
Next you need to set up the header search path so that the compiler will be happy. You want a happy compiler, right? ☺ Click on the **Build Settings** tab and enter **library** in the search field. You should see that these were added to the **Library Search Paths**:

```
"$(SRCROOT)/OCMock"
```



Now enter **header search paths** in the search field and double-click on the **Header Search Paths** line. Click the plus (+) button and enter the following (pressing (+) to add each line):

```
"$(SRCROOT)/OCMock/OCMock"
"$(SRCROOT)/OCMock"
```



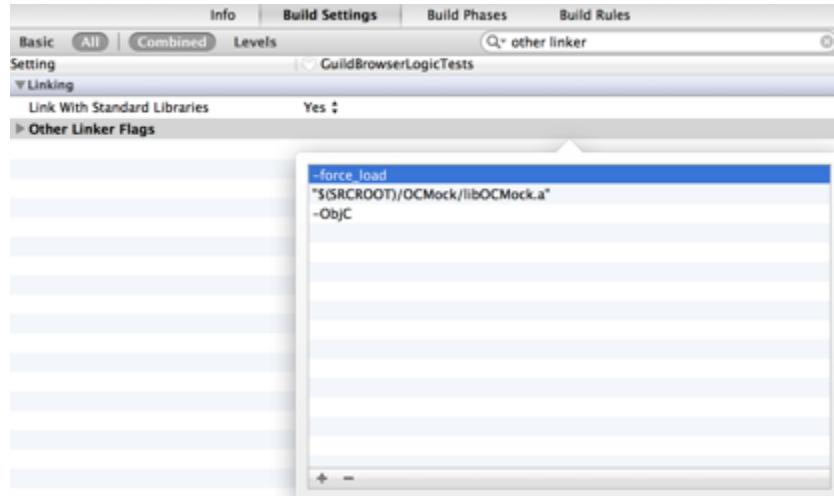
Press Enter, and then click anywhere outside the window to close it.

There's one more linker issue to work around in order to get all the symbols loaded properly for the OCMock library. One final time enter **other linker** in the **Build**

**Settings** search field and double-click on the **Other Link Flags** line. Click the plus (+) button and enter the following entries (one at a time):

```
-ObjC
"$(SRCROOT)/OCMock/libOCMock.a"
-force_load
```

Make sure that you enter them in the above order so that they end up looking like this:



Now for the moment of truth. ☺

Try building your test target by selecting Product\Test (⌘-U). The test should succeed, with no warnings. If you get a compile error, make sure your other linker settings are in the correct order and that you extracted the OCMock.zip file correctly and added the contents to the project folder at the top level.

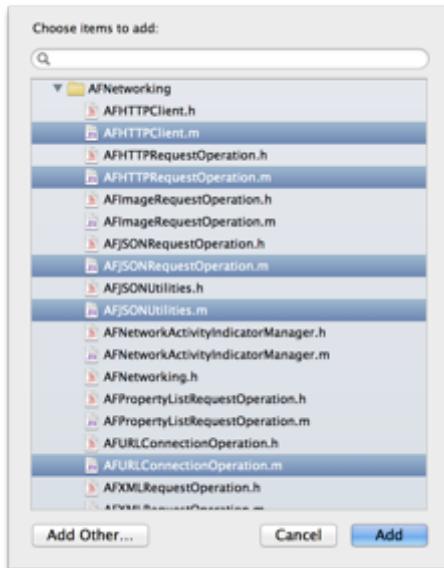
Before you proceed to create a unit test, add a few more classes to your `GuildBrowserLogicTests` target.

You are going to be creating a unit test that will mock up the `WowApiClient` class from the previous chapter. As you might recall, this class is a subclass of `AFHTTPClient` from the AFNetworking library. So you need to add `WowApiClient` and a few of the AFNetworking classes to your test target so your unit test will compile correctly.

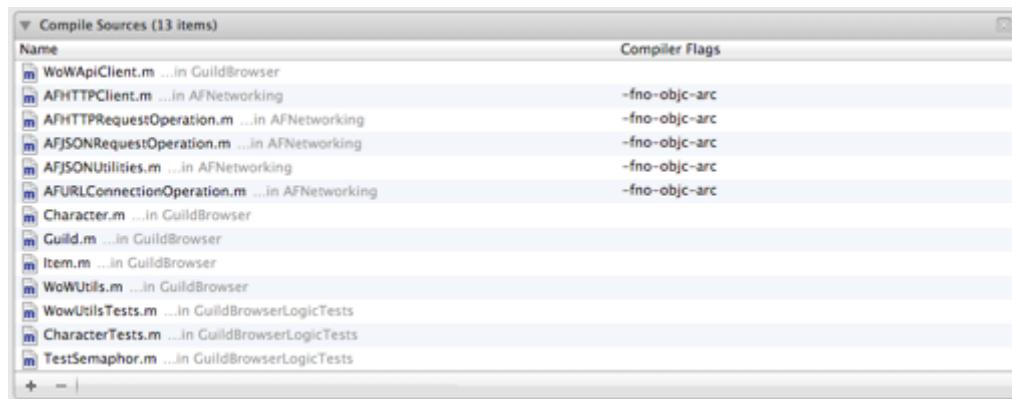
Click on the project root in the project navigator and then select the **GuildBrowserLogicTests** target. Click on the **Build Phases** tab, expand the **Compile Sources** section and click the plus (+) button. Select the following files:

- **AFHTTPClient.m**
- **AFHTTPRequestOperation.m**
- **AFJSONRequestOperation.m**
- **AFJSONUtilities.m**

- **AFURLConnectionOperation.m**
- **WoWApiClient.m**



Now you need to turn off ARC for the AFNetworking files. So, highlight all of the files that start with AF, tap Enter on the keyboard, type in `-fno-objc-arc`, and press the Enter key again. Your screen should now look like this:



Make sure to build and test now to ensure your project is set up correctly. It's easy to miss a file here. Build your test target by selecting Product\Test (⌘-U) and the test should succeed, with no warnings. If you get a compile error, make sure the correct list of files has been included in the Compile Sources section.

**Note:** You have to do this on a file-by-file basis because this is a logic unit test target that you want to be able to call from a script using `xcodebuild`.

If you're setting up OCMock in a test target that was created from one of the Xcode project templates, your test bundle will be run inside a host and all of the project files would already be available.

## Unit testing with OCMock

Remember that GuildBrowser retrieves data about characters in a World of Warcraft guild from the network, and then displays the results in a neat-looking UICollectionView.

Let's assume for a moment that you have no UI for this app. Imagine it's before you created the UICollectionViewController to show all of the guild members, and you want to make sure that the data representing the guild members is returned in the correct groupings (by class type), and in the correct order.

Let's also assume you've been following good TDD practices while creating this app, and you have no networking code done, but you know what you want your networking API to look like.

In **WowApiClient.h**, you can see that you decided to return a block type that looks like this:

```
typedef void (^GuildBlock)(Guild *guild);
```

So the callers of the method to get the guild data will receive a populated Guild object in the success block. Here is what the method to retrieve guild data looks like:

```
-(void)guildWithName:(NSString *)guildName
 onRealm:(NSString *)realmName
 success:(GuildBlock)successBlock
 error:(ErrorBlock)errorBlock;
```

So how do you create a unit test involving blocks, asynchronous calls, and simulated data from a unit test, without having to depend on the network services?

With OCMock and TestSemaphor, that's how!

TestSemaphor is a class developed by fellow author Marin Todorov that makes testing asynchronous block-based API calls much easier. This class is already included in the starter app for this chapter. Without this class, you end up driving straight to Hackville with loads of RunLoops and other madness all over your test code. Trust me, you want to use TestSemaphor for this. I've seen unit-testing code without it, and it's scary.

**Note:** Here is Marin's blog post about TestSemaphor – you will go over it below during the unit test discussion: <http://www.touch-code-magazine.com/unit-testing-for-blocks-based-apis/>

Now let's create a mock for the `WoWApiClient` class, which is the class providing all of the data for your app. Your unit test will make sure that the `Guild` object is correctly loading all of the members, in the correct order for use by the UI.

In the Xcode project navigator, right-click on the **GuildBrowserLogicTests** group, choose **New File... \iOS\ Cocoa Touch\ Objective-C test case** class, click **Next**, enter **GuildTests** for the class name, click **Next** again, make sure only the **GuildBrowserLogicTests** target is selected, and then click **Create**.

Replace the contents of **GuildTests.m** with the following code:

```
#import "GuildTests.h"
#import "WoWApiClient.h"
#import <OCMock/OCMock.h>
#import "Guild.h"
#import "TestSemaphore.h"
#import "Character.h"

@implementation GuildTests
{
 // 1
 Guild *_guild;
 NSDictionary *_testGuildData;
}

- (void)setUp
{
 // 2
 NSURL *dataServiceURL = [[NSBundle bundleForClass:self.class]
URLForResource:@"guild" withExtension:@"json"];
 NSData *sampleData = [NSData
dataWithContentsOfURL:dataServiceURL];
 NSError *error;
 id json = [NSJSONSerialization JSONObjectWithData:sampleData
options:kNilOptions error:&error];
 _testGuildData = json;
}

- (void)tearDown
{
 // Tear-down code here.
 _guild = nil;
 _testGuildData = nil;
}
```

```
@end
```

So far, this is just set up for the actual test. This should be review from the last chapter, but here is what is happening:

1. Create some instance variables: `_testGuildData` to stuff with your simulated network data found in `guild.json`, and a `Guild` variable that will be created from this data in the mocked `WowApiClient`.
2. Load the JSON file from the test bundle. Note the use of the `self.class` to `NSBundle`.

Now let's add your test case. Add the following to **GuildTests.m**:

```
- (void)testCreatingGuildDataFromWowApiClient
{
 // 1
 id mockWowApiClient = [OCMockObject mockForClass:[WowApiClient
class]];

 //
 // using OCMock to mock our WowApiClient object
 //

 // 2
 [[[mockWowApiClient stub] andDo:^(NSInvocation *invocation)
 {
 // how the success block is defined from our client
 // this is how we return data to caller from stubbed
method

 // 3
 void (^successBlock)(Guild *guild);

 //
 // gets the success block from the call to our stub method
 // The hidden arguments self (of type id) and _cmd (of
type SEL) are at indices 0 and 1;
 // method-specific arguments begin at index 2.

 // 4
 [invocation getArgument:&successBlock atIndex:4];

 // first create sample guild from file vs network call

 // 5
 Guild *testData = [[Guild alloc]
initWithGuildData:_testGuildData];
 }
]];
}
```

```
// 6
successBlock(testData);
}]
// 7
// the actual method we are stubb'ing, accepting any args
guildWithName:[OCMArg any]
onRealm:[OCMArg any]
success:[OCMArg any]
error:[OCMArg any]];

// String used to wait for block to complete

// 8
NSString *semaphoreKey = @"membersLoaded";

//
// now call the stubbed out client, by calling the real method
//

// 9
[mockWowApiClient guildWithName:@"Dream Catchers"
 onRealm:@"Borean Tundra"
 success:^(Guild *guild)
{
 // 10
 _guild = guild;

 // this will allow the test to continue by lifting the
 // semaphore key
 // and satisfying the running loop that is waiting on it
 // to lift

 // 11
 [[TestSemaphor sharedInstance] lift:semaphoreKey];

} error:^(NSError *error) {
 // 12
 [[TestSemaphor sharedInstance] lift:semaphoreKey];
}];

// 13
// Marin is so awesome
[[TestSemaphor sharedInstance] waitForKey:semaphoreKey];
```

```
// 14
STAssertNotNil(_guild, @"");
STAssertEqualObjects(_guild.name, @"Dream Catchers", nil);
STAssertTrue([_guild.members count] == [[_testGuildData
valueForKey:@"members"] count], nil);

//
// Now validate that each type of class was loaded in the
correct order
// this tests the calls that our CharacterViewController will
be making
// for the UICollectionViewDataSource methods
//

// 15

//
// Validate 1 Death Knight ordered by level, achievement
points
//
NSArray *characters = [_guild
membersByWowClassTypeName:WowClassTypeDeathKnight];
STAssertEqualObjects(((Character*)characters[0]).name,
@"Lixiu", nil);

//
// Validate 3 Druids ordered by level, achievement points
//
characters = [_guild
membersByWowClassTypeName:WowClassTypeDruid];
STAssertEqualObjects(((Character*)characters[0]).name,
@"Elassa", nil);
STAssertEqualObjects(((Character*)characters[1]).name,
@"Ivymoon", nil);
STAssertEqualObjects(((Character*)characters[2]).name,
@"Everybody", nil);

//
// Validate 2 Hunter ordered by level, achievement points
//
characters = [_guild
membersByWowClassTypeName:WowClassTypeHunter];
STAssertEqualObjects(((Character*)characters[0]).name,
@"Bulldogg", nil);
```

```
 STAssertEqualObjects(((Character*)characters[1]).name,
 @"Bluekat", nil);

 //
 // Validate 2 Mage ordered by level, achievement points
 //
 characters = [_guild
membersByWowClassTypeName:WowClassTypeMage];
 STAssertEqualObjects(((Character*)characters[0]).name,
 @"Mirai", nil);
 STAssertEqualObjects(((Character*)characters[1]).name,
 @"Greatdane", nil);

 //
 // Validate 3 Paladin ordered by level, achievement points
 //
 characters = [_guild
membersByWowClassTypeName:WowClassTypePaladin];
 STAssertEqualObjects(((Character*)characters[0]).name,
 @"Verikus", nil);
 STAssertEqualObjects(((Character*)characters[1]).name,
 @"Jonan", nil);
 STAssertEqualObjects(((Character*)characters[2]).name,
 @"Desplaines", nil);

 //
 // Validate 3 Priest ordered by level, achievement points
 //
 characters = [_guild
membersByWowClassTypeName:WowClassTypePriest];
 STAssertEqualObjects(((Character*)characters[0]).name,
 @"Mercpriest", nil);
 STAssertEqualObjects(((Character*)characters[1]).name,
 @"Monk", nil);
 STAssertEqualObjects(((Character*)characters[2]).name,
 @"Blian", nil);

 //
 // Validate 3 Rogue ordered by level, achievement points
 //
 characters = [_guild
membersByWowClassTypeName:WowClassTypeRogue];
 STAssertEqualObjects(((Character*)characters[0]).name,
 @"Lailet", nil);
```

```
 STAssertEqualObjects(((Character*)characters[1]).name,
 @"Britaxis", nil);
 STAssertEqualObjects(((Character*)characters[2]).name,
 @"Josephus", nil);

}
```

Here's a breakdown of the above code:

1. Create a mock object that will be used just like a real instance of `WowApiClient`. Any method that you call on this instance needs to be defined – you do this in step 2.
2. In steps 2-6, you define what you want to happen when you call the `guildWithName:onRealm:success:error` method on your mock instance of `WowApiClient`. If you call methods on the mock object that you hadn't previously defined, you will get exceptions. If you don't want that to happen, you can create a "nice" instance of your class by calling `niceMockForClass` instead of `mockForClass`. So when you call the `guildWithName:onRealm:success:error` method, you define what you want to happen in the OCMock stub's `andDo:` invocation block.
3. Create a block variable to pass to the success block, which is how you will get your test data to the test case.
4. OCMock lets you have access to the method arguments, but your arguments start at index 2. This is because the `_id` and `_cmd` arguments are at indexes 0 and 1. Note that the `successBlock` is the fourth argument when starting at index 2 (`guildWithName = 2, onRealm = 3, success = 4`). You store this reference into the block type you defined in step 3.
5. You now create the `Guild` instance `testData`, by passing the simulated JSON data to `initWithGuildData:`.
6. Your `Guild` instance is now passed to the `successBlock` you previously got a reference to in step 4.
7. Here you finally see the method definition on your mock object, `guildWithName:onRealm:success:error`. Note that this is continuing the method call that begins in step 2 – it's just that one of the arguments contained a block of code, which made it span so many lines. All the `[ocmArg any]` parameters indicate that you will allow any type of argument to be passed into your mock and that there are no constraints.
8. Since you are using blocks, the execution of the `success` block once the test completes will be asynchronous. Without adding some sort of "wait" or timeout, the test will not wait for the block to come back.

That's where the `TestSemaphore` object, mentioned previously, comes in. It's really simple to use. You create a string value to wait for, and when your call is done – in this case, when you get returned to your `success` block – you "lift" this string.

Confused? Yes, it might be a little confusing at this point, but consider the `semaphoreKey` as the key to a gate blocking you, or rather, blocking the test. The test waits until it receives the key to proceed further, and you send the key to the test in step 11. So when you get to that, this might make better sense. ☺

To prove this is working, comment out the line of code under `// 8` and step through in the debugger, and you will fly past the rest of the test. ☺

9. Call `guildWithName:onRealm:success:error:` on your mock object.
10. When the method returns via the `success` block, you will have the `Guild` object all set up and ready to use. Now you can test it to make sure that the returned data is correct.
11. Since you are done waiting for the data to be returned, it's time to let the test continue by lifting the semaphore key. This satisfies the run loop that is waiting for your particular string to be lifted, so that it can continue operations.
12. You also lift the semaphore if the error block returned instead. In this case, you don't set the `Guild` data, since there is none, but you do lift the semaphore so that the execution of the test can continue.
13. You saw in steps 11 and 12 that you lift the `semaphoreKey` when the block is done. In order to make the test wait for this to happen, you have to first set the semaphore to wait. You do this by calling `waitForKey` and passing in the string value that will lift the key. The test will wait at this step until the `semaphoreKey` is lifted.
14. If you got to this part, then the semaphore waiting is done and the data has been received (or data retrieval failed), so you can actually test something! First you make sure that `_guild` is not `nil`, because if it is, then that means receiving data failed. You also make sure that the name is correct, and that the `_guild.members` count matches the JSON data.
15. The way the code is written, the UI expects the data to be in a specific order. So here and through the rest of the test case, you verify that the `guild` class returns the proper data in the correct order.

If you weren't using a mock, with data that you control, you couldn't guarantee what data the web service would return. With `ocMock` you have a great way to simulate network conditions and test out your classes without having to depend on a network connection.

OK, let's see if this thing works! Try building your test target by selecting `Product\Test` (⌘-U) and the test should succeed, with no warnings.

To gain a deeper understanding of the test you just performed, open up the **guild.json** file found in the project in **GuildBrowserLogicTests\testdata\guild.json** and take a look at some of the things you are testing. For example, try finding all of the Druids and note their level and achievement data. Changing the class type for one of them should break the test. Let's see what that looks like.

WowUtils.h has an enum defined for showing what the class types are by number:

```
// class types
typedef enum {
 Warrior = 1,
 Paladin = 2,
 Hunter = 3,
 Rogue = 4,
 Priest = 5,
 DeathKnight = 6,
 Shaman = 7,
 Mage = 8,
 Warlock = 9,
 Monk = 10,
 Druid = 11,
} CharacterClassType;
```

So Druids are classType: 11 in guild.json. Let's change one of those to the upcoming Monk class. Hey, it might make me play WOW again! After all, the world needs more Pandas. ☺

Let's change the character named – wait for it, this is a real character name – **Everybody** from a Druid to a Monk.

In the **guild.json** file, search for **Everybody**, and change “**class": 11** to “**class": 10** and save the file.

Now run the test again by selecting Product\Test (⌘-U). The test fails, because there are no longer three Druids.

OK, you are now set up to include OCMock in your unit testing adventures. You looked at a pretty complex test involving blocks and networking calls, but testing with mock objects is a huge subject. Take some time going through more unit tests with OCMock, perhaps creating a mock for a datasource providing data to a UITableView or UICollectionView.

## UI Automation

Now that you know how to create tests for things that do not have (or require) a UI, let's turn your attention to testing the UI of an app. In this section you'll create a script to automatically test your user interface, using the Instruments tool included with Xcode.

Why would you want a UI automation script? There are a few good reasons:

- Creating a UI automation script is really easy, compared with writing a lot of code in an Objective-C unit test to drive the UI.

- You can easily create a set of scripts to run all parts of your UI.
- You need to stress-test your app and see how it performs.
- You can connect multiple instruments together to watch for memory leaks, etc. in Instruments.
- It's like building a robot!

## What is UI Automation?

UI Automation is an Apple technology that lets you automate UIKit-based applications. UI Automation scripts simulate user interaction with your apps. You can simulate just about anything you can think of, including taps, drags, pinches, swipes, flicks, long presses, two-finger taps, double taps, taps with options, rotations, backflips, and scrolling! Just kidding about the backflips – that comes out in iOS 7. ☺

The key class to all of this interaction is **UIAElement**. If you're serious about UI Automation, be sure to bookmark the following:

<http://developer.apple.com/library/ios/#documentation/ToolsLanguages/Reference/UIAElementClassReference/UIAElement/UIAElement.html>

UI Automation scripts work by accessing a hierarchy of objects, all of which are subclasses of **UIAElement**. Every application, when run in the Automation Instrument, has this view hierarchy created. Accessing specific parts of this hierarchy is how you automate your application.

One easy way to find the **UIAElement** representing a view – say, a custom **UICollectionViewCell** – is by setting the accessibility label to a known value. **UIAElement** has a helper method, `-name`, that will return the view with the given accessibility label. You will see an example of this below.

Here are some of the key players you'll use writing UI Automation scripts:

- **UIATarget:** This is the system under test, the target running on either your device or in the Simulator.
- **UIAApplication:** This gives you access to the application-level elements in the view hierarchy. (The active app from the target.)
- **UIAWindow:** This gives you access to and control of your application's main window elements. (Your app's main window.)
- **UIALogger.logPass:** The mechanism for passing a test!
- **UIALogger.logFail:** This is the mechanism for failing a UI Automation script. Green is good, red is bad!
- **UIALogger.logMessage:** This one logs messages.

- **<UIAElement>.logElementTree:** As you will see later, this is simply amazing. It will print a screenshot of every view in the hierarchy. It also prints a log of the view hierarchy and is the key to finding things in your scripts.

## Accessibility and UI Automation

Just about every UIKit class can have accessibility turned on and an accessibility label set. If you do these two things, not only will your app be more accessible to users with disabilities, but you will also be able to lookup the object by its label in your UI Automation script.

**Note:** To learn more about making your apps accessible to users with disabilities, check out Chapter 25, "Accessibility."

In this section, you will programmatically set the accessibility label of your `UICollectionViewCells` to be a character's name, and then use that label to tap it by name!

One thing that might surprise you is that UI Automation scripts are written in...gasp... Javascript!



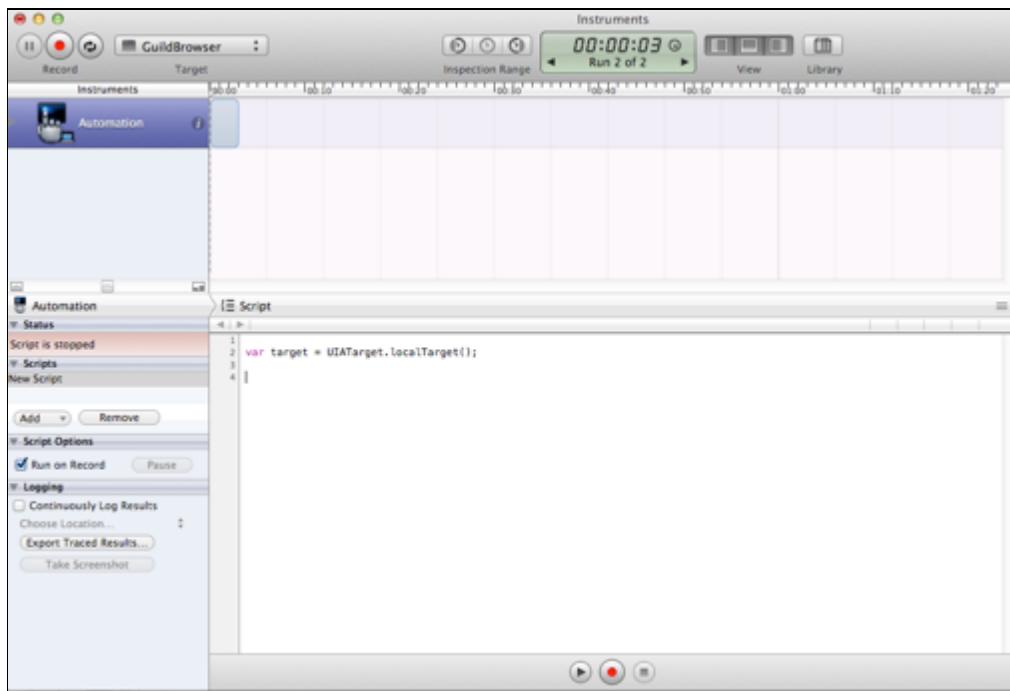
Yes, Javascript! Hey, don't stop reading! It's not that bad. ☺

In the latest version of Xcode, there is an editor built right into Instruments for UI Automation, let's check that out now.

In the GuildBrowser app, make sure the current device is set to iPad 6.0 Simulator, and select **Product\Profile** from the menu to launch the Instruments application. In the template chooser, select **Automation** and then choose **Profile**.

The GuildBrowser app should have launched in the Simulator. Switch to Instruments, and you will see that it is now profiling your application, which right now isn't doing much. Click the red stop button or press **⌘-R**. This stops the trace and changes the button label to **Record**.

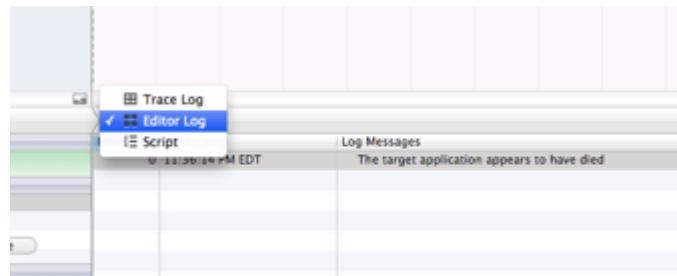
Let's add a script. Click the **Add** button (on the left sidebar under the Scripts section) and choose **Create...** Your screen should look like this:



Notice that the target is set to the GuildBrowser app (on the top left next to the three buttons). When running your scripts, it's best to run them via the Record / ⌘-R button, as this will launch the app in a known starting condition. My personal experience has been that you will use the Play button (at the bottom of the window) until it bites you and then remember to use the Record button! ☺

If you run the app, then you'll notice that the bottom right-hand section changes to the Trace Log view. You can switch back to the Script editor by clicking on the Trace Log label (or Editor Log if you are in that mode) and choosing Script.

This isn't the best UI in the world – here is a screenshot of how you switch between the different view modes:



Every time you run the script it will switch to the Editor Log if you used the Play button, and the Trace Log if you ran it via Record.

But before you get going on scripting, remember when I said accessibility makes it easier for your scripts? Let's go ahead and modify the `charactercell` class to have its accessibility labels set programmatically.

Open **CharacterCell.m** and in `configureCellWithCharacter:` after the line:

```
self.character = selectedCharacter;
```

Add this:

```
self.accessibilityLabel = _character.name;
self.isAccessibilityElement = YES;
```

This simply sets the `accessibilityLabel` to be the character's name, and allows you to use scripting to tap the cell by character name!

OK, now it's time to do some scripting!

From Xcode, launch your app into Instruments to make sure you have the latest build by using **Product\Profile** or **⌘-I**. The app should launch in the Simulator. You might get an error message from Instruments that "the simulated application quit." If so, just click OK and the trace should be running again. Stop the trace by clicking on **Stop** or **⌘-R**, and switch from the Instruments log viewer to the script editor.

Replace the contents of the script editor with this code:

```
// 1
var target = UIATarget.localTarget();
var app = target.frontMostApp();
var window = app.mainWindow();
var navBar = window.navigationBar();

//2
// make sure we see the correct guild on app start
var testGroup = "verify guild is correct"

// 3
UIALogger.logStart(testGroup)

// pause for app to start and load data from network
target.delay(5);

var currentGuildName = navBar.staticTexts()[0].name();

// 4
if (currentGuildName != "Dream Catchers") {
 UIALogger.LogError("The wrong guild is showing: currentGuildName
 == " + currentGuildName);
```

```
 UIALogger.logFail(testGroup);
} else {
 UIALogger.logMessage("We found the correct guild yay!");
 // 5
 // take a screenshot
 target.captureScreenWithName("guild");

 // 6 debug view hierarchy
 target.logElementTree();
 UIALogger.logPass(testGroup);
}
```

This first script is pretty simple – it launches the app and verifies that the navigation bar in the app has the correct title. But it's a great place to start.

Here's the breakdown:

1. This is going to be standard code for all of your UI Automation scripts. Creating these variables will save you from chaining together all of these to traverse the view hierarchy – as in `UIATarget.localTarget().frontMostApp().mainWindow()` to get to the main window. The `UIATarget` is the root object to access the view hierarchy. Next you get your app, which will be the front-most one, and finally you get the window from the app.
2. Think of each test as a “group” – giving it a name helps when looking at the log output (which needs all the help it can get!). All of your tests should follow this pattern:

```
var testGroup = "this is what I'm testing";

UIALogger.logStart(testGroup);

// your tests

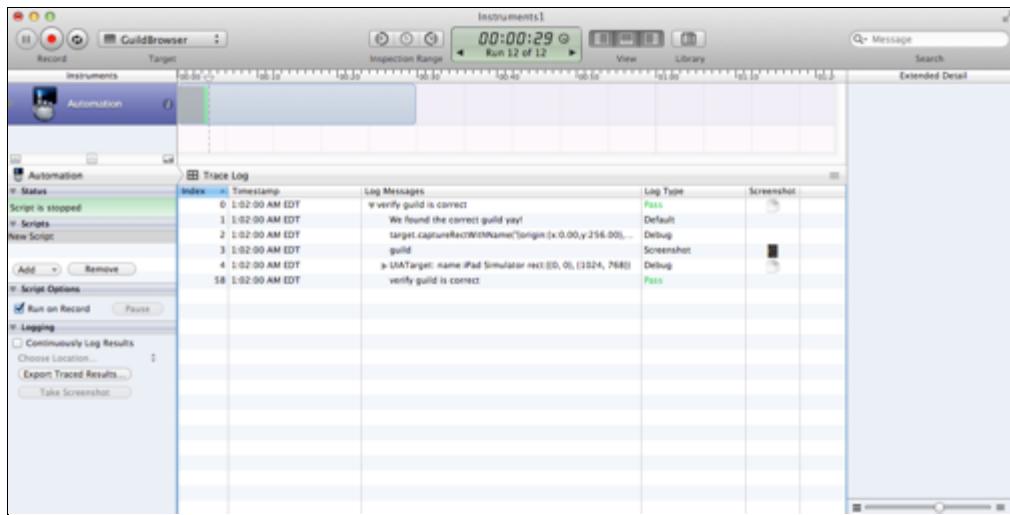
// then either
UIALogger.logFail(testGroup)
// OR
UIALogger.logPass(testGroup)
```

3. Now you start the test group, which adds an entry in the log viewer. Next you get the name of the guild by accessing the navigation bar `staticTexts` array of `UIAStaticText` objects. Since you know your navigation bar only has the title, you can get the first one.
4. Now it's time for the actual test.
5. This is pretty neat – you can take a screenshot at any time within your script, which is especially useful if you're doing something interesting like changing views, drilling into a navigation bar, etc.

6. As mentioned earlier, calling `target.logElementTree()` will add an entry into your test output showing you the entire view hierarchy. You can make this call on any `UIAElement`. It's a lifesaver when trying to figure out how to select something.

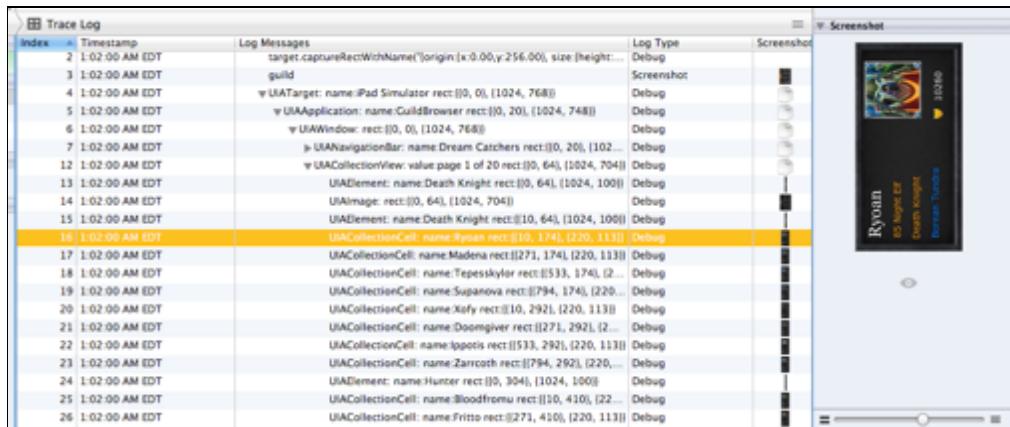
For example, when you called `navBar.staticTexts()` above, if you had added a `navBar.logElementTree()` before that, then you would see the values for `staticTexts` printed out. You would see the output after running the script, and this would allow you to drill into the entire view hierarchy of the app to figure out how to get to a specific element.

Run the test now from Instruments using the Record button or ⌘-R, and you should see the app start in the Simulator. Now take a look at the Trace Log viewer in Instruments. It should look like this:



Green is good! Notice that the screenshot is in the output also.

Now let's look at the debug output from `logElementTree()` by expanding the fourth entry, `UIATarget`. Notice that you can click on each level and see the screenshot to the right. Notice also that all of the `UIACollectionCells` have a `name` value set to the character's name. This is because of the code you added above to the `CharacterCell` class.



Because each cell has the character name set, you can use that name to tap a cell. In fact, that sounds like another great test – clicking on a particular character by name, verifying that the character detail screen loads correctly, and then closing it.

Add the following code to the end of the existing script:

```
// 1
testGroup = "CharacterDetail Screen Test";
UIALogger.logStart(testGroup);
UIALogger.logMessage("Selecting Xofy");

// 2
var collectionView = window.collectionViews()[0];

// see what we currently look like
target.captureScreenWithName("app loaded screenshot");

UIALogger.logMessage("Kicking a gnome!");

// 3
character = collectionView.cells().firstWithName("Xofy");
character.tap();
character.waitForInvalid();

target.captureScreenWithName("Xofy - Character Detail");

// 4
var toolbarTitle = window.toolbar().staticTexts()[0].name()

if (toolbarTitle != "Bloodsail Admiral Xofy") {
 UIALogger.LogError("We are kicking the wrong gnome");
 UIALogger.logFail(testGroup);
} else {
 UIALogger.logMessage("Woot boot a gnome!");
 UIALogger.logPass(testGroup);
}

// now close character detail screen
target.frontMostApp().toolbar().buttons()["Close"].tap();
```

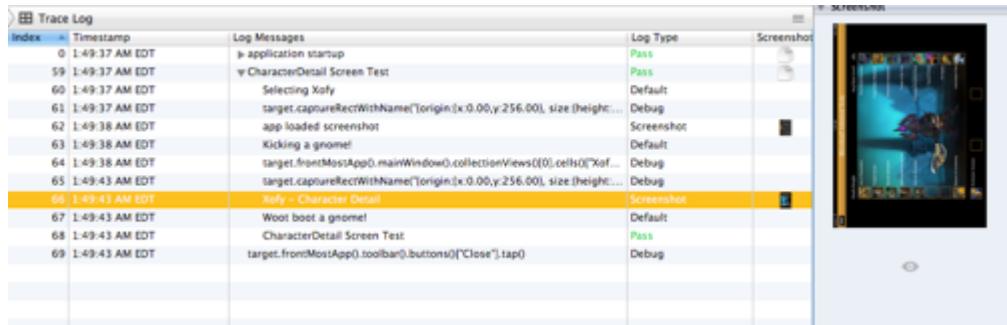
This test case is pretty similar to the previous one, but this time, you have to actually make the test do stuff too! Here's what happens:

1. Set up the test case, entering a new test group name and calling `logStart`. This makes things much neater and easier to read in the Trace Log viewer.
2. Get a reference to the `UICollectionView`, knowing you only have one in the app.

3. Here is where the accessibility change you made to the `CharacterCell` object helps out. `firstWithName` is a special function that will return the first `UICollectionViewCell` that, in this case, has an accessibility label with the name "Xofy". That cell is then tapped and the code waits for the cell to be invalidated (meaning that the next screen has loaded) before proceeding further.

4. To make sure you have actually loaded the character detail screen, you verify that the title matches by looking at the toolbar title property.

Run the script by clicking on Record (or ⌘-R), and your output should be similar to the following. Note that when you select the cell, the screenshot shows the correct character detail page:



Now let's add one more test case to your script. Let's make sure that when a user types in a new guild name, it will load correctly.

Add the following to the end of the automation script:

```
testGroup = "ChangeGuild";
UIALogger.logStart(testGroup);
UIALogger.logMessage("Selecting a new guild");

// 1
navBar.rightButton().tap();
target.frontMostApp().mainWindow().popover().textFields()[0].setValue("Fishsticks")
// 2
target.tap({x:322.00, y:252.00});
target.delay(4);

currentGuildName = navBar.staticTexts()[0].name();

if (currentGuildName != "Fishsticks") {
 UIALogger.LogError("The wrong guild is showing");
 UIALogger.logFail(testGroup);
} else {
 UIALogger.logMessage("We found the correct guild yay!");
 target.captureScreenWithName("guild");
```

```
 UIALogger.logPass(testGroup);
 }
```

A lot is happening here for so few lines of code:

1. Use the view hierarchy to get the `rightButton` from the `navBar` and tap it. This causes the popover to show up, and you set the value by accessing the view hierarchy, knowing that the first text field is the guild name field.
2. To dismiss the popover, you click outside of it on the app, and add a delay to make sure everything loads up properly. The rest of the code here is the same as in your first test case for loading the app.

Run the code, and accept my congratulations! You have now added automated UI testing to your app! You should see all three of the tests running in sequence, with each group passing.

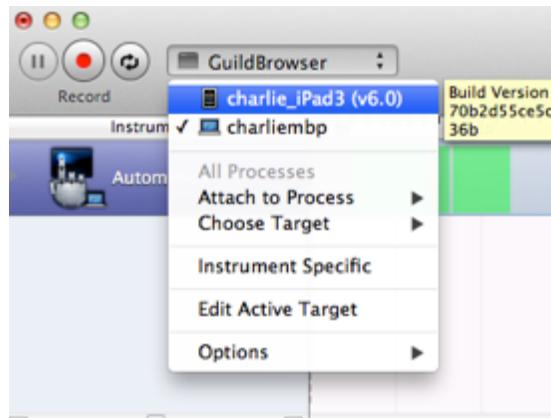
## UI Automation on a device

It's also possible to run a script to automate your app deployed to an iOS device. Let's see that in action. First go back to your project in Xcode and get the latest build out to your device.

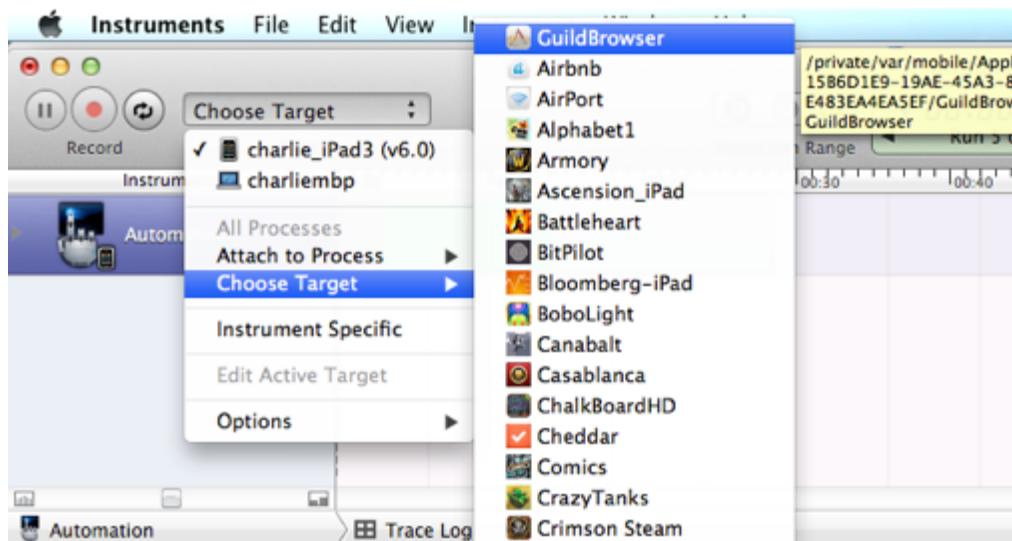
**Note:** This assumes you can deploy to your device and that you have an iPad2 or later.

Switch the scheme to **GuildBrowser\<your iOS Device>** and run the app to deploy it. It helps to connect your device now, too. Make sure everything is working properly, stop the app and then go back to the script in Instruments.

Select a new target by clicking on the dropdown that says GuildBrowser and choosing your device. It should look like this:



Now click on the dropdown again. It will say Choose Target, so go to **Choose Target\GuildBrowser**, which should be at the top of the list. I know it's tempting, but don't choose Bobo. ☺



Now run the script by pressing Record / ⌘-R and you should see the app running on its own on your device. I think that's really cool!

While you have your device loaded up, now is the time to use the Record button at the bottom of the script editor. Click the button and the app will load onto the device. Now start using the app and notice that you get snippets of code automatically loaded into the editor. This is a great way to get started with scripts! I have found they make a lot more sense, however, after you have written a few from scratch.

## Where to go from here?

If you've made it through both of these chapters, you should feel very proud – you now know more about automated, unit, and UI testing than 90% of iOS developers! ☺

You now have the skills you need to set up a robust automated testing system for your apps. You know how to:

- Set up a central continuous integration server that pulls down your code checked into Git, automatically builds your apps, and runs your unit tests.
- And speaking of unit tests – you know how to create tests for both simple and complex cases that involve mock objects or user interface testing.
- And best of all – once all this is complete, you know how to automatically upload the resulting build to TestFlight for deployment to your testers!

Adding automated testing to your apps will help you identify bugs quickly and have confidence that your app still works correctly as you add new features. It makes working as a team much easier, because with a good testing process, you can trust that your team's code will build cleanly and run correctly even when you don't know

exactly what your teammates are up to. And if someone does break the build, you'll know right away who it was!

I hope you enjoyed these bonus chapters, and I hope to see many of you use automated testing in your apps in the future!



# 25

# Chapter 25: Accessibility

By Felipe Laso Marsetti

Welcome to a very unique chapter, Accessibility for iOS.

Though not new to iOS 6, Accessibility is something that has not yet been covered or “tutorialized” before at [raywenderlich.com](http://raywenderlich.com). Luckily for you (and iOS users with disabilities), we’re here to fill the gap and help you make your apps more accessible to everyone.

When developing apps, most developers understand the importance of an eye-popping UI, eliminating bugs, making apps iPhone- and iPad-compatible (Universal), internationalization and translation to other languages, the icon and default images, social network incorporation, and so on.

Rarely, however, do developers consider Accessibility a core feature or requirement for their apps. It’s a shame, really, considering how easy it is to add Accessibility support – Apple has done most of the job for you, as you’ll see!

But more importantly, you can help change someone’s life by making your app accessible to as many iOS users as possible, regardless of their condition or disability.

This chapter will get you acquainted with the tools iOS provides out of the box, as well as what you as a developer need to do to add Accessibility support for your apps.

The chapter will start by introducing the Accessibility features available in iOS, and explain how to use and enable them. From there, you will take the **Fun Facts** project built in Chapter 11, “Beginning Social Framework,” and add support for Accessibility.

Are you ready? Awesome! Let’s get started. ☺

## Accessibility and iOS

You may not know it, but iOS includes some very cool Accessibility features such as:

- **VoiceOver:** This is Apple's screen reading technology that allows users to hear an audible description of UI elements so they can work with their device without having to see the screen.
- **Screen Zoom:** iOS includes a built-in method to zoom in on the screen on any app (Settings\General\Accessibility\Zoom), which is great for users with low visibility. It's also great for developers when you want to easily zoom in to take a look at some detail on a piece of UI such as an icon or a drop shadow. ☺
- **Mono Audio:** iOS also includes a method to route both audio channels into a both earbuds, an aid for users with hearing loss.
- **Much, much more:** That's just the beginning – iOS also includes inverting screen colors, support for hearing aids, custom vibrations, assistive touch, and a whole lot more!

The great news is most of the features above work fine just "out of the box" without you having to do anything – except for VoiceOver. VoiceOver is the area that you have most control over as a developer and where you will be spending most of your time in this chapter.

To get a better feeling for what VoiceOver is, try it out!

**Note:** Accessibility functionality works only on the device. So you need to develop on device for this chapter.

On your device, go to Settings\General\Accessibility\VoiceOver, and turn it on (and make sure Speak Hints is on as well).



Once you turn it on, you'll see that your device will start reading the screen to you! It also modifies touch behavior – when you tap the screen, first it selects an item

(so you it can read to you what you just tapped), and then if you tap it again it's as if a tap occurs. Similarly, you now drag three fingers to scroll.

Play around with a bit so you can get used to how it works. For fun, you can even close your eyes and see if you can navigate your iPhone with your eyes closed!

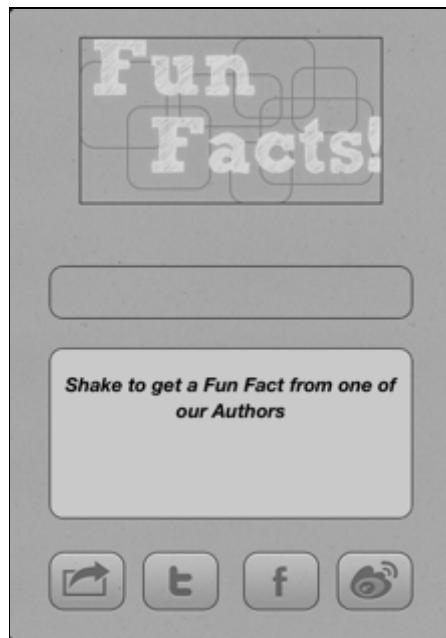
Once you've finished playing around with VoiceOver and feel comfortable with it, come back to get started with the project and learn how you can make sure VoiceOver works properly with your apps!

## Testing the current Fun Facts app

If you've already completed Chapter 11, "Beginning Social Framework," you can use the Fun Facts project you created there as the starting point for this chapter. Otherwise, you can find a starter copy of the project among the resources for this chapter.

Before looking at the classes that make up the project, run the app to see what it does, even if you're already familiar with the project. Be sure to have VoiceOver and Speak Hints enabled via the Settings app.

For a bit of fun, and to better understand why accessibility can be so important to users, close your eyes and try to understand what Fun Facts does with just the spoken hints you receive.



How was this experience for you? Were you able to understand what the app does with just the spoken hints from VoiceOver?

Don't worry, I didn't understand much either, nor did I know what to do with the app. This is why a proper implementation of Accessibility is so important!

Now that you've played with the app for a bit, let me explain what it does for those who haven't been through Chapter 11.

Fun Facts gives you random facts about the authors and editors of *iOS 6 by Tutorials* when you shake the device. You can then share these facts via Twitter, Facebook or Sina Weibo, or use an activity view controller to print, copy, save to camera roll, etc.

If you were using VoiceOver with spoken hints while running the app, you may have noticed some of the following:

- Tapping the buttons at the bottom of the screen simply gives you the spoken hint "icon <name> button," where the "icon <name>" part is the icon used for the button.
- Tapping the author image gives no spoken feedback.
- The author name and Twitter username are spoken as-is. No feedback or context is given to these items.
- There is no explanation as to what the app does, how to use it or any other context given.
- The user is never told to shake the device in order to get a new fact.
- No feedback is given when the user has shaken the device and has obtained a new fact.

As you can see, there is a lot of work that can be done to make this app truly accessible!

## Making Fun Facts accessible

Let's take this one item at a time, and start by fixing the Fun Facts image view. Right now, if you tap it nothing happens – but it would be nice if it gave a hint about how to use the app, along with the name of the person shown in the image view if appropriate.

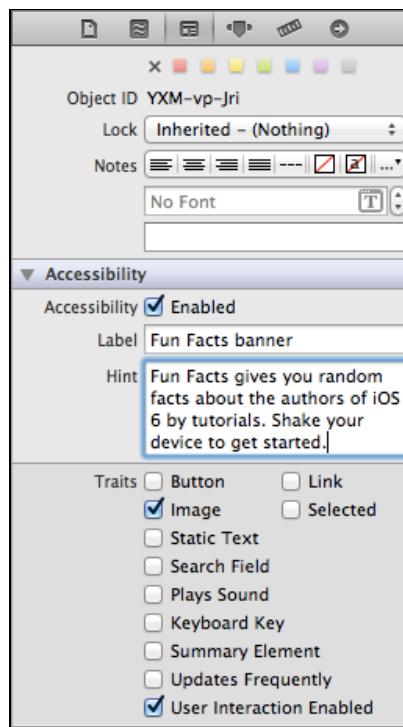
The good news is it's pretty easy to make this happen – it's as easy as changing a couple settings in the Storyboard editor!

Let's try it out. Open **MainStoryboard.storyboard** and select the `UIImageView` that holds the banner and author images. With the image view selected, open the Utilities pane and switch over to the Identity inspector.

You may have to scroll down a bit to see that there is a section titled Accessibility with a bunch of features that, most of the time, are enough to add the proper accessibility support in your apps.

Make sure the **Accessibility Enabled** checkbox is selected, then add the following **Label** text: "Fun Facts banner". Now switch to the **Hint** section and enter the following text: "Fun Facts gives you random facts about the authors of iOS 6 by Tutorials. Shake your device to get started."

Finally, for **Traits**, make sure **Image** and **User Interaction Enabled** are both selected.



Run the app, and notice how before you even tapped anything, VoiceOver went ahead and spoke the app's name, the label for the image view as well as its hint, all without you having to enter a single line of code. How cool was that?

You can also tap the image view and the label and hint will be spoken once again. So far so good!

If you incorporate this into your habitual workflow – putting an item in Interface Builder and giving it the appropriate Accessibility settings – you'll be well on your way to expanding your audience, making your app enjoyable to more people and earning a very loyal group of followers. And don't forget to advertise accessibility as a feature of your app!

There's a problem though. Shake the app so a random fun fact is displayed – the image view should now show an image of an author or editor. If you tap the image view, it still says "Fun Facts banner" as its text. That's not what you want – it should read the author or editor's name instead.

There's no good way to do this in Interface Builder though as it depends on the author or editor that is displayed – so it's time to go to code!

## Accessibility attributes

Everything you can do in Interface Builder with, you can also do in code. Let's go over each of the attributes you can set on a UIView that deal with accessibility.

The two most important attributes you can configure on your views for Accessibility are:

```
@property(nonatomic, assign) BOOL isAccessibilityElement
@property(nonatomic, retain) NSString *accessibilityLabel
```

`isAccessibilityElement` simply indicates whether VoiceOver should see this element or not – by default, most controls in UIKit are set to YES. If you have your own custom control, you might want to manually set this to yes.

`accessibilityLabel` is a text representation of the object that VoiceOver will read to the user – it's the same thing you currently set as "Fun Facts Label" in Interface Builder. This sounds like the thing you need to change – so let's try it out!

Switch over to **SocialViewController.m**, go to `motionEnded:withEvent:`, and add the following line at the end of the `if` statement:

```
self.authorImageView.accessibilityLabel = name;
```

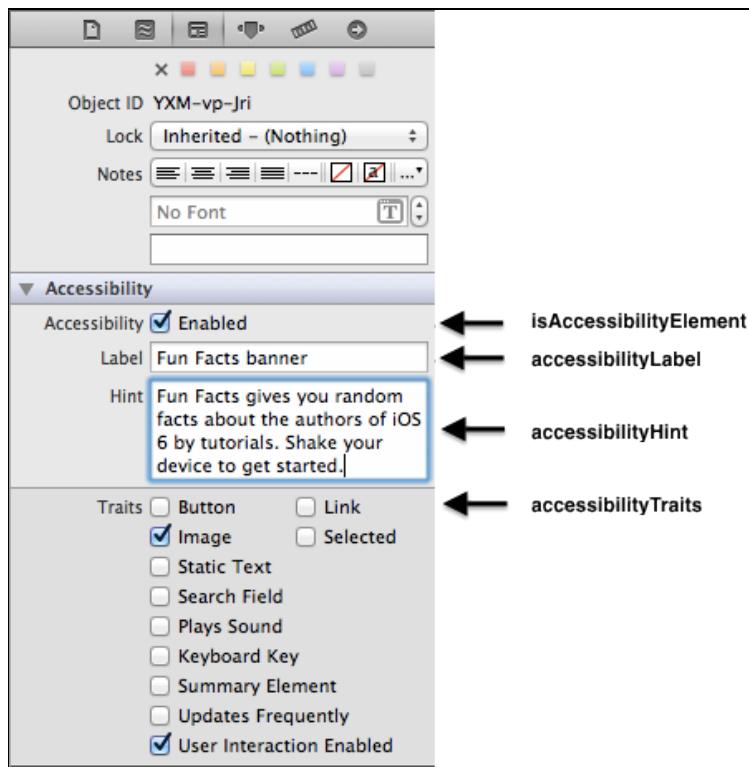
Build and run, shake for a random fact, and then tap the image. It will now read the author's name, followed by "image" and the hint. Nice!

There are two more accessibility properties to cover before you move on though:

```
@property(nonatomic, retain) NSString *accessibilityHint
@property(nonatomic, assign) UIAccessibilityTraits
accessibilityTraits
```

`accessibilityHint` is the same as the "Hint" value you set earlier in Interface Builder. This is an optional hint that provides more information and context to the user.

`accessibilityTraits` is a bitmask of integers corresponding to the checkboxes in Interface Builder that let you define the behavior of the element, as you can see below:



One thing to keep in mind is that if you have your own custom `UIControl` subclass, there probably isn't any Accessibility configuration or any default values. So remember to manually set the appropriate properties , like the below:

```
control.isAccessibilityElement = YES;
control.accessibilityLabel = @"My Control";
```

If the values you need for Accessibility are somehow dynamic or calculated at runtime, then you need to override the relevant methods to provide the proper Accessibility behavior, like this for example:

```
- (BOOL)isAccessibilityElement
{
 return YES;
}

- (NSString *)accessibilityLabel
{
 if (myCondition)
 {
 return @"If Condition Label";
 }

 return @"Default Label";
}
```

For the most part, setting up these attributes for your elements (whether custom or native) is all you need to do. Even if you do have to override some methods, it really is simple and takes very little work.

## Accessibility notifications

There are cases where you want to send a voice announcement to the user to let them know something that's happened with the app. For example, in this app it would be great if you could tell the user after they shake the screen that a new fun fact is displayed.

For these situations, you can use Accessibility Notifications. There are many different notifications for accessibility, but the two most important ones you will use in your applications are:

1. `UIAccessibilityScreenChangedNotification`: Use this when the contents of the screen change so that VoiceOver can reset itself.
2. `UIAccessibilityLayoutChangedNotification`: Use this when some items change onscreen, so that VoiceOver can update itself.

Let's try this out. Go to `SocialViewController.m` and add the following line at the end of the `if` statement in `motionEnded:withEvent::`:

```
UIAccessibilityPostNotification(
 UIAccessibilityLayoutChangedNotification, self.factTextView);
```

The first parameter is the notification to post, and the second parameter varies depending on the notification used. This particular notification takes a string or accessibility element that VoiceOver will then speak. You use the text view here so VoiceOver will read the new fun fact to the user.

**Note:** To find out what the second parameter means for different notifications, check out the `UIAccessibility` Protocol Reference for full details. Also note that passing the Accessibility element (to become the focused element) as a parameter of `UIAccessibilityLayoutChangedNotification` is new to iOS 6.

Build and run the app, and now after you shake the app it will read the latest fun fact!

## Boisterous buttons

Let's continue making this app Accessible by fixing up the four buttons at the bottom of the app.

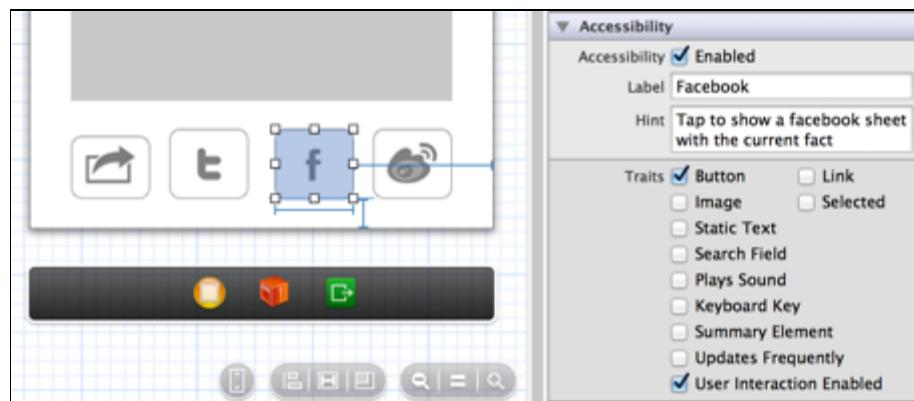
Select the leftmost button (the Twitter button) in Interface builder and set the Accessibility settings as follows:



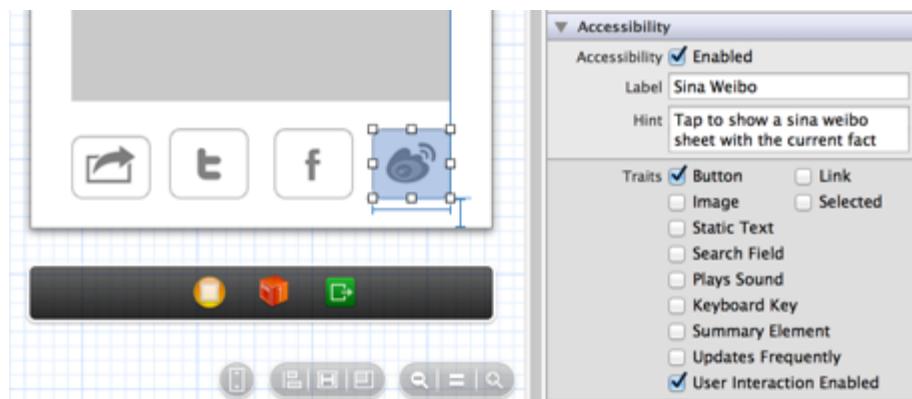
Then the Twitter button:



The Facebook button:



And the Sina Weibo button:



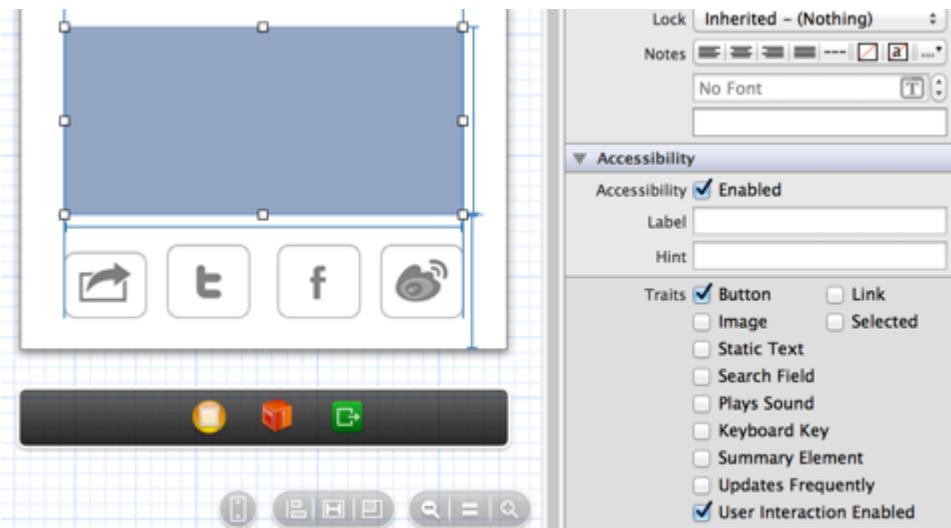
Build and run the app. Instead of simply saying “button” when you tap on any of the four buttons, VoiceOver should now read out the label, followed by “button” and then the hint.

Thanks to Interface Builder and the simplicity of the Accessibility API, it’s super-easy to add the appropriate information for the elements within your applications, whether they are default UIKit controls or custom controls.

## Talkative text

Running the app, you may have noticed a few more glitches and imperfections in the Accessibility implementation. Tapping somewhere in the text view selects just certain portions of text to be read out loud.

Fortunately, it’s very easy to fix this. Inside Interface Builder, select the text view, go to the Identity Inspector’s Accessibility section and enable the “Button” trait:



By setting the text view as a button, it makes it so that when you tap it all of the text is read (instead of only a certain portion).

Also annoying is what’s spoken when you tap the “Fun Fact From” label. Having that label’s text spoken gives no context or useful information whatsoever. To fix

this, go to **SocialViewController.m** and add the following line inside the `if` statement of `motionEnded:withEvent::`:

```
self.factTitleLabel.accessibilityLabel = [NSString
stringWithFormat:@"%@", self.factTitleLabel.text, name];
```

Run the app again to see if these changes worked. Ahh, yes, the author's fact is now read in its entirety, and tapping the "Fun Fact From" label causes VoiceOver to speak the label's text, followed by the author's name.

Perfect!

## Finishing touches

All that remains is setting up the author name, Twitter handle and fact information. You'll do this in code, so replace the code for the `if` statement inside `motionEnded:withEvent:` in **SocialViewController.m** with the following:

```
if (motion == UIEventSubtypeMotionShake)
{
 self.deviceWasShaken = YES;

 NSUInteger authorRandSize = self.authorsArray.count;
 NSUInteger authorRandomIndex = (arc4random() %
((unsigned)authorRandSize));

 NSDictionary *authorDictionary =
self.authorsArray[authorRandomIndex];

 NSArray *facts = authorDictionary[AuthorFactsKey];
 NSString *image = authorDictionary[AuthorImageKey];
 NSString *name = authorDictionary[AuthorNameKey];
 NSString *twitter = authorDictionary[AuthorTwitterKey];

 NSUInteger factsRandSize = facts.count;
 NSUInteger factsRandomIndex = (arc4random() %
((unsigned)factsRandSize));

 self.authorImageView.image = [UIImage imageNamed:image];
 self.authorImageView.accessibilityLabel = name;

 self.factTextView.text = facts[factsRandomIndex];

 self.factTitleLabel.hidden = NO;
 self.factTitleLabel.accessibilityLabel = [NSString
stringWithFormat:@"%@", self.factTitleLabel.text, name];
```

```
 self.nameLabel.text = name;
 self.nameLabel.isAccessibilityElement = YES;
 self.nameLabel.accessibilityLabel = [NSString
stringWithFormat:@"Author Name: %@", name];

 self.twitterLabel.text = twitter;
 self.twitterLabel.isAccessibilityElement = YES;
 self.twitterLabel.accessibilityLabel = [NSString
stringWithFormat:@"Twitter Username: %@", twitter];

UIAccessibilityPostNotification(UIAccessibilityLayoutChangedNotification, self.factTextView);
}
```

This code simply sets the accessibilityLabel for fact, name, and twitter labels appropriately.

One more time, run the app to make sure VoiceOver behaves as expected.

Great success! The app now provides useful labels and hints for the onscreen elements, as well as custom text to be spoken out loud for some elements in order to provide better feedback and context.

## iOS 6 Accessibility enhancements

In this section, you're going to learn about some new Accessibility enhancements that were introduced in iOS 6 by adding them to Fun Facts!

There are four new features that you'll add into the app: magic tap, a new accessibility notification, ordering container VoiceOvers, and a new accessibility trait. Let's get started!

### Magic Tap

One of the awesome new things available in Accessibility with iOS 6 is something called **Magic Tap**. A Magic Tap occurs when a user double-taps with two fingers. Accessibility now lets you control what happens when the Magic Tap occurs.

Here are some examples of what Magic Tap does in some of the built-in apps included with iOS:

- **Clock:** Starts/Stops the stopwatch.
- **Camera:** Takes a picture.
- **iPod:** Pauses the music if music is playing.
- **Phone:** Automatically answers/hangs up a phone call.

To catch the Magic Tap and perform your own custom actions, you must override the following method:

```
- (BOOL)accessibilityPerformMagicTap
```

Right now, double-tapping with two fingers (the Magic Tap) in Fun Facts does nothing. But what if, when a Magic Tap occurs, a new fact was randomly generated, in case the user has motion disabilities and can't perform a shake gesture? That would be pretty cool!

**Note:** Alternatively, users could use the Assistive Touch feature in Accessibility Settings, that allows users to perform a shake gesture by tapping an onscreen button.

There is a bit of refactoring to be done to make the Magic Tap generate a random fact. All of the code inside the `if` statement in `motionEnded:withEvent:` needs to be extracted into its own method. This will avoid code duplication and will keep your code nice and DRY.

**Note:** DRY stands for **Don't Repeat Yourself**, which is programmer jargon for extracting code that repeats often and putting it in a single method.

In case you aren't there already, open **SocialViewController.m** and remove all the code inside the `if` statement of `motionEnded:withEvent:`, but make sure you leave the following line:

```
self.deviceWasShaken = YES;
```

Now add a new method, which will contain all the lines you just deleted:

```
- (void)generateRandomFact
{
 NSUInteger authorRandSize = self.authorsArray.count;
 NSUInteger authorRandomIndex = (arc4random() %
((unsigned)authorRandSize));

 NSDictionary *authorDictionary =
self.authorsArray[authorRandomIndex];

 NSArray *facts = authorDictionary[AuthorFactsKey];
 NSString *image = authorDictionary[AuthorImageKey];
 NSString *name = authorDictionary[AuthorNameKey];
 NSString *twitter = authorDictionary[AuthorTwitterKey];
```

```

 NSUInteger factsRandSize = facts.count;
 NSUInteger factsRandomIndex = (arc4random() %
((unsigned)factsRandSize));

 self.authorImageView.image = [UIImage imageNamed:image];
 self.authorImageView.accessibilityLabel = name;

 self.factTextView.text = facts[factsRandomIndex];

 self.factTitleLabel.hidden = NO;
 self.factTitleLabel.accessibilityLabel = [NSString
stringWithFormat:@"%@: %@", self.factTitleLabel.text, name];

 self.nameLabel.text = name;
 self.nameLabel.isAccessibilityElement = YES;
 self.nameLabel.accessibilityLabel = [NSString
stringWithFormat:@"Author Name: %@", name];

 self.twitterLabel.text = twitter;
 self.twitterLabel.isAccessibilityElement = YES;
 self.twitterLabel.accessibilityLabel = [NSString
stringWithFormat:@"Twitter Username: %@", twitter];

UIAccessibilityPostNotification(UIAccessibilityLayoutChangedNotification, self.factTextView);
}

```

Back in `motionEnded:withEvent:`, add a call to `generateRandomFact` inside the `if` statement:

```
[self generateRandomFact];
```

Finally, implement `accessibilityPerformMagicTap` as follows:

```

- (BOOL)accessibilityPerformMagicTap
{
 [self generateRandomFact];

 return YES;
}

```

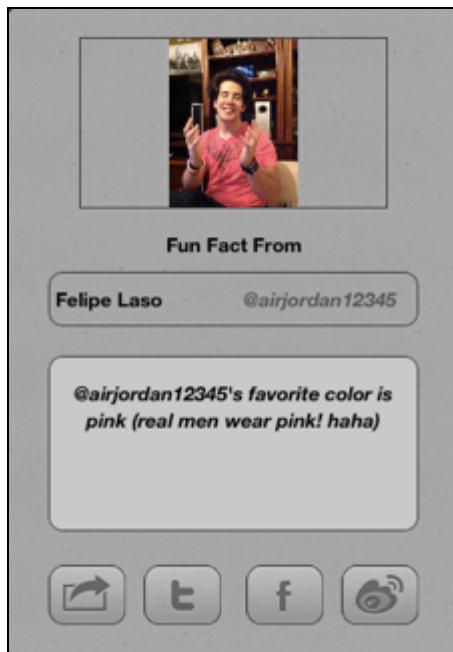
Give the app another run and perform the Magic Tap by double-tapping with two fingers. Notice how the fact changes when you perform the Magic Tap, another small detail that can go a long way towards improving your app's accessibility.

**Note:** You can implement `accessibilityPerformMagicTap` on an individual object, inside the app delegate, or on a parent in the view hierarchy. In the latter case, VoiceOver will run through the view hierarchy and catch each object that it finds.

## Fun Facts complete!

You're finally done making Fun Facts accessible!

Here is an image of the finished project, complete with Accessibility. I know, it doesn't look different in any way. It's all "under the hood," as some might say. But you and I know it's a much better app. ☺



## Where to go from here?

As you can tell from reading this chapter, Apple has put a huge amount of effort into making the OS accessible to users with disabilities. As an app developer, so should you – especially since it requires such little effort!

Most of the time you can simply set some accessibility attributes in Interface Builder and you're done – and when that isn't enough, you have a robust API to help you out. The amount of customer appreciation and loyalty you will gain in return by making your apps accessible far outweighs the value of the time you'll spend.

But don't just think about Accessibility as a marketing and promotion tool; think about the lives you may be changing or enhancing thanks to the added support in your apps.

Now that you've seen how Accessibility works, it's time for you to try it out. Take any project you're currently working on or an app that you may have in the App Store and add Accessibility to it. Pick one that has some challenging problems for you to resolve, like:

- View-less controls
- Custom drawing
- Dynamic UIs
- Lots of on screen elements
- Animations
- Dynamic labels and components that need to be set up programmatically

At first, it may be a process of trial and error to get the results you expect, but after you get the hang of it you'll have no trouble whatsoever incorporating Accessibility into your daily workflow.

Also, don't forget to check out the **Accessibility Programming Guide** available from Apple, as well as the developer forums. In addition, you should check out some of the comments and feedback in the App Store for apps with excellent Accessibility support, as well as feedback you may receive when you add support for Accessibility in your own applications.

Just as many developers can't live without version control systems or unit testing, they shouldn't be able to live without Accessibility. It's time you made good use of it and brought a lot more smiles to some very special customers of yours! ☺

# Chapter 26: Secrets of Info.plist

By Matt Galloway

By now, as an iOS developer, you will no doubt have seen the file called **Info.plist** (or sometimes **YourProjectName-Info.plist**) many times. You may have edited it directly, or you may have used Xcode's project details panel to edit it.

Either way, I wager you have been left wondering what some of the options mean, or what options are available you might not know about. Or maybe you've spotted an option that you think looks interesting, but you were afraid to use it because you didn't understand it!

Well, this bonus chapter outlines *all* of the options available to an iOS 6.0 app via its Info.plist file – that's right, all of them! – and explains why you might want to use each one.

## The basics of Info.plist

Before we dive into the specific content of Info.plist, let's take a higher-level view and see what it's all about.

To begin with, Info.plist uses a file format that's been around since before iOS was even invented – Apple's **property list** format.

A property list is an UTF-8 encoded XML file designed to contain the most commonly-used data types, such as:

- Dictionary (`NSDictionary`)
- Array (`NSArray`)
- String (`NSString`)
- Number – int, float, bool (`NSNumber`)
- Date (`NSDate`)
- Data (`NSData`)

In the case of Info.plist, most values are strings, numbers, arrays or dictionaries.

## What does an Info.plist do?

Info.plist exists to supply iOS with some important information about an app, bundle or framework. It designates things such as how an app should launch, how it can be localized, the name of the app, the icon to display, and much more.

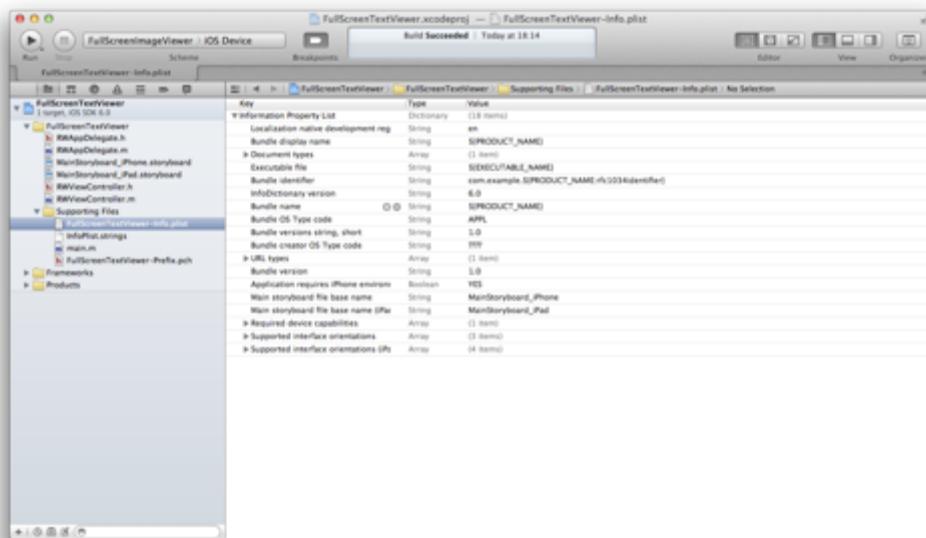
On iOS, an app's Info.plist is loaded into the operating system's internal data structures when the app is installed, so that other parts of the system know it exists. For instance, the fact that your app handles a certain file type, or the icon to show when you do a search on your device – this information needs to be available to the OS somehow, and Info.plist is how Apple chose to provide it.

That means it's an easy way for you to tell iOS some information about how you'd like your app configured.

## Editing an Info.plist

There are two ways you can edit the Info.plist file in your project. Pull up a project of yours in Xcode so that you can experiment with using both ways.

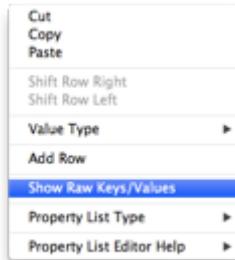
Your first option is to use the standard property list editor. Just find your Info.plist in the project navigator in Xcode and single-click on it. You will then see the various options, or keys, in the main editor window, like this:



Here's a cropped version so you can see those keys better:

|                                           |            |                                                |
|-------------------------------------------|------------|------------------------------------------------|
| ▼ Information Property List               | Dictionary | (20 items)                                     |
| Localization native development region    | String     | en                                             |
| Bundle display name                       | String     | FullScreenText                                 |
| Executable file                           | String     | \$(EXECUTABLE_NAME)                            |
| ► Icon files (iOS 5)                      | Dictionary | (2 items)                                      |
| Bundle identifier                         | String     | com.example.\$[PRODUCT_NAME:rfc1034identifier] |
| InfoDictionary version                    | String     | 6.0                                            |
| Bundle name                               | String     | FullScreenText                                 |
| Bundle OS Type code                       | String     | APPL                                           |
| Bundle versions string, short             | String     | 1.0                                            |
| Bundle creator OS Type code               | String     | ????                                           |
| Bundle version                            | String     | 1                                              |
| Application requires iPhone environment   | Boolean    | YES                                            |
| Application supports iTunes file sharing  | Boolean    | YES                                            |
| Main storyboard file base name            | String     | MainStoryboard_iPhone                          |
| Main storyboard file base name (iPad)     | String     | MainStoryboard_iPad                            |
| Icon already includes gloss effects       | Boolean    | YES                                            |
| Status bar is initially hidden            | Boolean    | NO                                             |
| Status bar style                          | String     | Gray style (default)                           |
| ► Supported interface orientations        | Array      | (3 items)                                      |
| ► Supported interface orientations (iPad) | Array      | (4 items)                                      |

Notice that all of the keys are fairly descriptive, but it's worth pointing out that behind those descriptions are Apple-defined strings (which you'll hear more about in the rest of this chapter). You can switch the view to show the raw keys by right-clicking anywhere in the list and selecting "**Show Raw Keys/Values**":



You'll see that the description for each field changes to a "raw/predefined constant" value like this:

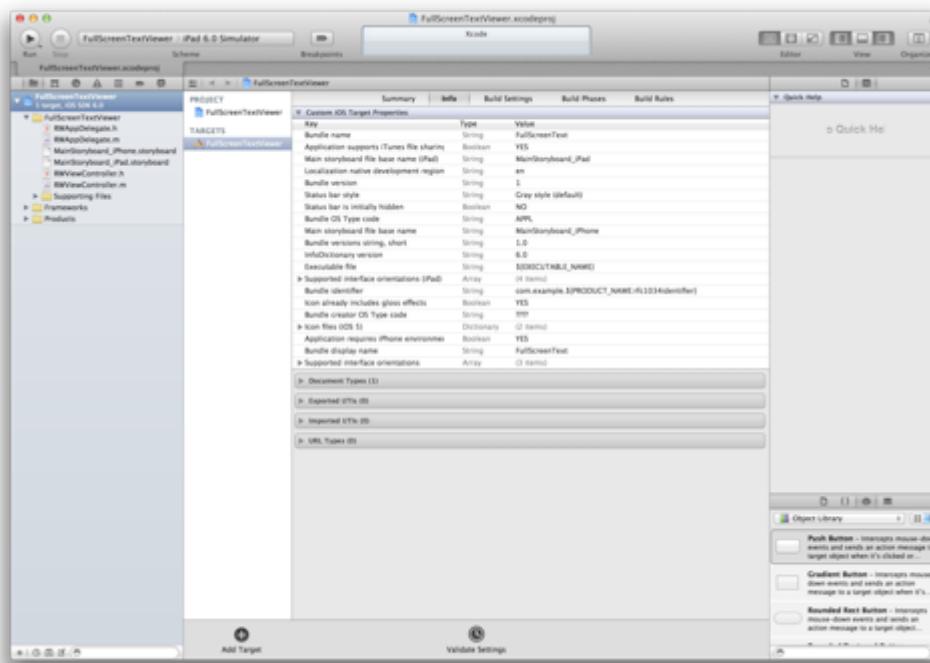
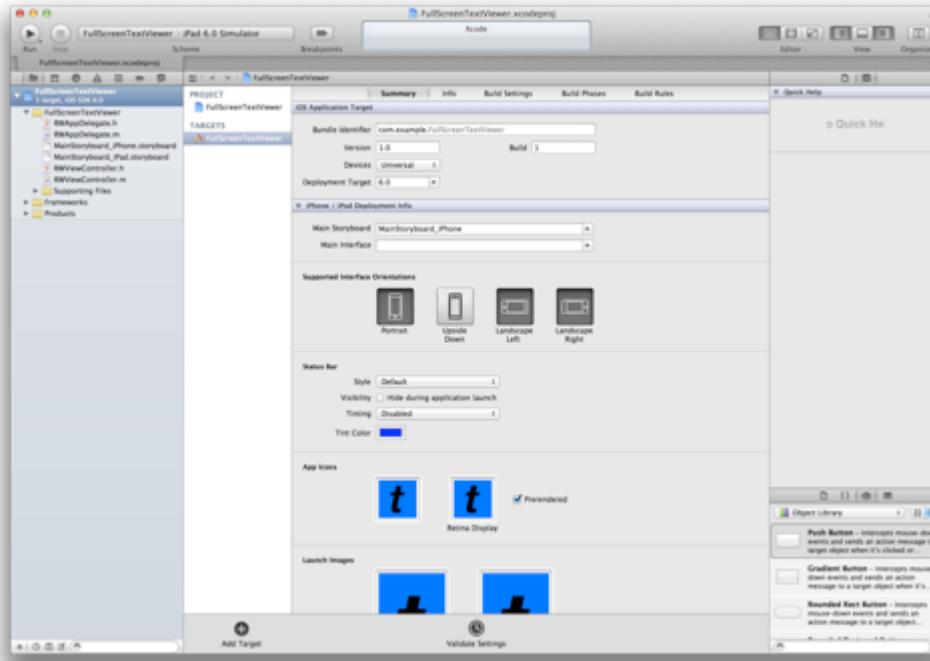
| ▼Information Property List              |            |                                               |
|-----------------------------------------|------------|-----------------------------------------------|
|                                         | Dictionary | (20 items)                                    |
| CFBundleDevelopmentRegion               | String     | en                                            |
| CFBundleDisplayName                     | String     | FullScreenText                                |
| CFBundleExecutable                      | String     | \${EXECUTABLE_NAME}                           |
| ► CFBundleIcons                         | Dictionary | (2 items)                                     |
| CFBundleIdentifier                      | String     | com.example.\${PRODUCT_NAME}fc1034identifier} |
| CFBundleInfoDictionaryVersion           | String     | 6.0                                           |
| CFBundleName                            | String     | FullScreenText                                |
| CFBundlePackageType                     | String     | APPL                                          |
| CFBundleShortVersionString              | String     | 1.0                                           |
| CFBundleSignature                       | String     | ???                                           |
| ► CFBundleVersion                       | String     | 1                                             |
| LSRequiresiPhoneOS                      | Boolean    | YES                                           |
| UIFileSharingEnabled                    | Boolean    | YES                                           |
| UIMainStoryboardFile                    | String     | MainStoryboard_iPhone                         |
| UIMainStoryboardFile~ipad               | String     | MainStoryboard_iPad                           |
| UIPrerenderedIcon                       | Boolean    | YES                                           |
| UIStatusBarHidden                       | Boolean    | NO                                            |
| UIStatusBarStyle                        | String     | UIStatusBarStyleDefault                       |
| ► UISupportedInterfaceOrientations      | Array      | (3 items)                                     |
| ► UISupportedInterfaceOrientations~ipad | Array      | (4 items)                                     |

It's often easier to work with the Info.plist with the "raw" view because sometimes it's hard to find the correct "long descriptive" string in the dropdown (and sometimes they don't even exist in the dropdown!) In this chapter, you will see the list of potential Info.plist entries listed by their raw values, not the long descriptive names.

Your second option for viewing the Info.plist is to use Xcode's built-in editor. This makes life much easier by providing a pretty UI for many of the options, meaning that you don't need to remember all the options yourself.

To get to this handy UI, click on the project root at the top of the project navigator in Xcode. You'll be presented with a view with five tabs across the top. The two for editing the Info.plist file are **Summary** and **Info**.

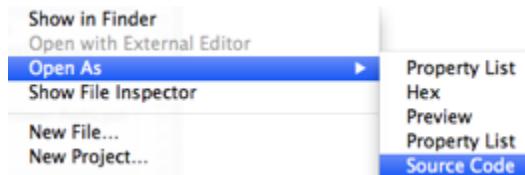
In your own project, you should see screens similar to the following.



When you change the settings in these screens, it is actually editing the Info.plist file for you behind the scenes.

## The structure of an Info.plist

As mentioned previously, under the hood a property list is an XML file that adheres to an Apple-defined schema. You can view an Info.plist in XML form by right clicking it and selecting Open As\Source Code.



You can edit a property list in the visual property list editor or in XML, whichever you're more comfortable with. In the rest of this chapter, we will list the property list elements in XML form, but if you prefer using the visual property list editor you can still use that – just select the appropriate data types from the dropdowns.

If you choose to edit the XML directly, be careful – it's easy to make a mistake! Be sure to check everything is OK by opening it as a property list again after editing.

The root object of a property list can either be an array or a dictionary, and the overall structure looks like this in XML form:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>SomeString</key>
 <string>This is some text</string>
 <key>SomeNumber</key>
 <integer>42</integer>
 <key>SomeDate</key>
 <date>1985-01-24T12:00:00Z</date>
 <key>SomeBoolean</key>
 <false/>
 <key>SomeData</key>
 <data>ASNFZ6vN7wA=</data>
 <key>SomeArray</key>
 <array>
 <string>Item 0</string>
 </array>
 <key>SomeDictionary</key>
 <dict>
 <key>AnItem</key>
 <string>Item 0</string>
 </dict>
</dict>
```

```
</plist>
```

Take a moment to read through that and see how all the various data types are encoded. Most are self explanatory; a date is formatted using the ISO 8601 standard, and data is Base64-encoded.

An Info.plist has a dictionary as its root, containing key-value mapping from system-defined keys to the value for your app. For example, one of the keys is called **CFBundleName**, and it defines the title that is shown under your app's icon on the iOS home screen.

### Device-specific keys

Usually, the key used in an Info.plist to identify a value is a constant string value. An example is **CFBundleName**, as mentioned above.

If you use the raw key name, then this will be the value used on every device. But if you want, you can specify different values for different devices.

An example of why you might want to do this is if you use the key to specify the app's main storyboard file. You will very likely want a different file for iPhone than for iPad, and this is made possible by using device-specific keys.

To form a device-specific key, you append either **~iphone** or **~ipad** to the key. So when specifying the main storyboard file for the app, you would use **UIStoryboardFile~iphone** and **UIStoryboardFile~ipad**. If a device-specific key does not exist, then the OS falls back to the generic key (e.g., **UIStoryboardFile**).

### Localizing values

Some keys in an Info.plist can be localized, meaning that iOS will pick an appropriate value based on the region specified in the device Settings.

For example, you might want to set a different display name for each supported language. To do this, you would create a file called **InfoPlist.strings** and include it in your project. Then you would localize it in the same way as you would any other strings file, and for each localization, set the key to what you want it to be.

## Unlocking the keys

You are probably wondering what all the different keys/options in an Info.plist mean, right? There are around 50 of them as of iOS 6.0!

Some of these options control display of the app when it's installed, some control what happens when the app is launched, some control various things within the system frameworks, and so on. For the rest of this chapter, we'll break the options down into sections to help explain what each one does.

To illustrate some of the options, I will refer to an app I've supplied called **FullScreenTextViewer**. A starter version of the project should be available with the resources for this chapter.

Open FullScreenTextViewer and familiarize yourself with it. It is a fairly simple app that displays text. You will add some features as you go through this chapter, like being able to open the app with a custom URL, opening text files from emails, changing various display options, and more.

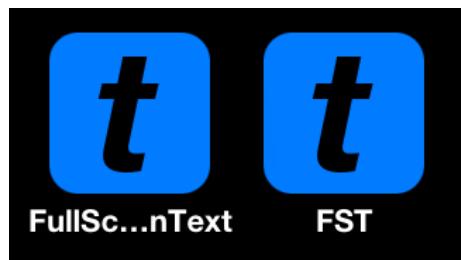
But enough with the background, let's get onto the options!

## Controlling the app name

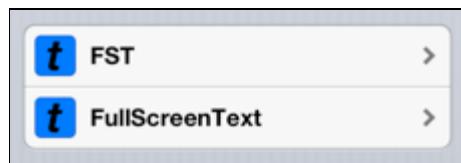
### CFBundleName (String)

This key specifies the name of your app as iOS sees it. The main place where your app name will be displayed is in SpringBoard (the iOS home screen).

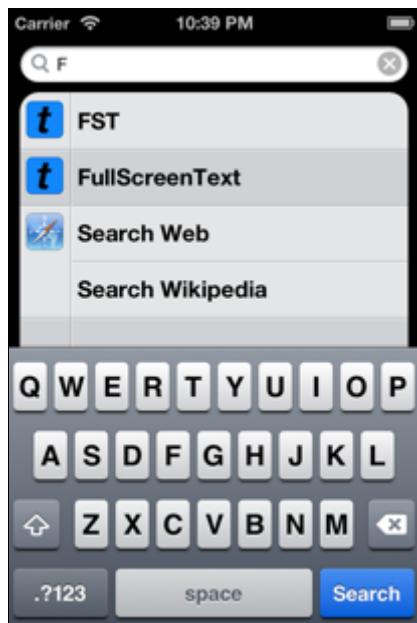
If the name is too long, then an ellipsis is put in the middle, making it look ugly. Nobody wants an ugly app name, so be sure to check out the display name in action. For example, look at the following two display names:



Your app name is also displayed in the Settings app (if it has any Settings):



...and in Spotlight (when the user searches for something):



...as well as in any other place where iOS displays your app's name. Be sure to check out all the various places, so you know an ugly version isn't hiding somewhere!

Example:

```
<key>CFBundleName</key>
<string>CoolApp</string>
```

### CFBundleDisplayName (String)

It's often a good idea to localize your app name. For example, consider an app called "Calculator." It would be very nice if it were localized to show "Calculatrice" in French or "Rechner" in German.

`CFBundleDisplayName` is how you achieve that, by using the `InfoPlist.strings` file to specify a value of the key for each language, as appropriate.



Do note that if you aren't going to localize your bundle, it's not a good idea to include this key in your Info.plist, since having it in there incurs a performance penalty. So add this key only if you want localization support for the display name.

Also note that the actual localization (the different names based on different languages) comes from the InfoPlist.strings file, and not from the value for this key.

Example:

```
<key>CFBundleDisplayName</key>
<string>CoolApp</string>
```

### Example using the sample project

To see these two keys in action, open the **FullScreenTextViewer** project and find the **InfoStrings.plist** file in the project navigator. There will be an arrow on the left of the file. Click that and it will open up a list of the localizations for that file, like this:



I've added English and French. Open each file separately and edit the `CFBundleName` and `CFBundleDisplayName` properties in each file.

Then build and run the app and press the home button, and you'll see what you entered appear under the app icon. If you change the region of your device or the Simulator from English-speaking to French-speaking, you will also see the different display names for each language.

## App identity

### CFBundleIdentifier (String)

iOS has to have some way to distinguish between apps. It could use the app names, but that's not a very good way because it's quite possible you might have two apps with the same name.

This is where `CFBundleIdentifier` comes in – it's a string containing a unique identifier. It is derived from your App ID, which you create in the iOS Provisioning Portal.

An App ID has two parts: the team identifier (a 10 character hex string generated by apple) followed by the bundle identifier. You want to put the bundle identifier into this field.



The bundle identifier is generally be in reverse domain notation using a domain that you own. For example, my company owns `swipestack.com`, so for an app called `HelloWorld`, I would use the bundle identifier `com.swipestack.HelloWorld`.

**Note:** An App ID is what defines an app, not its name. This means that you can change the name of your app and upload a new version to the App Store, and the name of your app will change – all that matters is your `CFBundleIdentifier` / App ID needs to be the same!

Alternatively, if you want to create a completely new version of your app with the same source code, you just need to change the `CFBundleIdentifier` / App ID to something else.

Your App ID is encoded into your provisioning profile, and when you upload your app to iTunes Connect, Apple verifies that your `CFBundleIdentifier` matches the bundle ID portion of your App ID in your provisioning profile.

If you look at the raw value for this key when you create a new project in Xcode, you will see something like this:

```
com.example.${PRODUCT_NAME:rfc1034identifier}
```

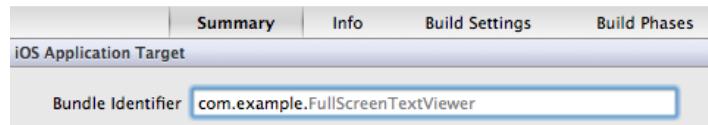
This is using a *macro* that Xcode has defined. First, `PRODUCT_NAME` is a variable defined in every Xcode project, set to the value in the “Product name” box when you created the project. The `rfc1034identifier` portion then formats that string to contain no illegal characters (spaces, for instance).

However, you could change this value if you wanted to remove the product name portion completely and specify a custom value. You might want to do this if you set up the app ID in the developer portal to a value other than your Xcode project’s name. Usually I change this manually to make sure it’s set up properly.

Example:

```
<key>CFBundleIdentifier</key>
<string>com.example.myawesomeapp</string>
```

Note that you can edit the bundle identifier visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. However, it will not allow you to edit the last part (it defaults to the project name). If your project name doesn’t match the last part of your bundle ID, you’ll need to edit the Info.plist directly.



## App version

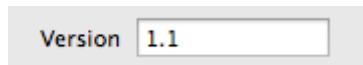
### CFBundleShortVersionString (String)

The value of this key is a string specifying the version of your app as you would see it in the App Store. For example: version 1.0 or 2.3. This value (or the `CFBundleVersion`) must increase between App Store releases.

Example:

```
<key>CFBundleShortVersionString</key>
<string>2.6</string>
```

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You’ll find a field for version there:



## CFBundleVersion (Integer)

The value of this key is an integer that you can increment for every release, be it a testing release or an App Store release. Most commonly, this key is set automatically using a build script (often utilizing **agvtool**), and if you're using SVN version control, then a good choice is the SVN revision at build time. This value (or the CFBundleShortVersionString) must increase between App Store releases.

Example:

```
<key>CFBundleVersion</key>
<string>12345</string>
```

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You'll find a field for Build there:



## Example using the sample project

To see these two keys in action, open the **FullScreenTextViewer** project and navigate to the Info.plist editor. The two keys you are looking for are called “**Bundle versions string, short**” and “**Bundle version**.“ Change them to whatever you want, and then build and run the app.

Tap on the **About** button at the top right and you will see an alert showing the app version as you have specified it. It will look something like this:



## App icons

The procedure for setting your app's icon has changed twice over the course of iOS's existence, so this is commonly an area of confusion for developers. I'll start with the oldest way and work up to the newest.

### CFBundleIconFile (String – DEPRECATED in 3.2)

This is the oldest way to specify your app's icon. The value required here is a single string telling iOS the name of your app's icon to be found in the main app bundle. This is not flexible, and since there are now many different icon sizes required for apps, this key was deprecated in iOS 3.2.

Example:

```
<key>CFBundleIconFile</key>
<string>Icon.png</string>
```

### UIPrerenderedIcon (Boolean)

This key has been around since the first iOS SDK and tells the OS whether or not to add the shine/glossy effect that you see on app icons.

If this key is set to **true**, it indicates that the icon should be displayed as-is, without any shine effect, i.e., it is *prerendered*. If it's set to **false**, then the shine effect will be added.

Here's an example of the FullScreenTextViewer icon with and without shine effect:



Example:

```
<key>UIPrerenderedIcon</key>
<true/>
```

### CFBundleIconFiles (Array – DEPRECATED in 5.0)

This key was introduced to cope with the increasing number of icon sizes that were required when the iPad and retina screens came into being. The value for this key should be an array of strings, each of which is an icon file name matching one of the following sizes:

- iPhone main app icon (57 x 57 & 114 x 114 retina).
- iPad main app icon (72 x 72 & 144 x 144).
- iPhone Spotlight search and settings icons (29 x 29 & 58 x 58).
- iPad Spotlight search icon (50 x 50 & 100 x 100).
- iPad settings icon (29 x 29 & 58 x 58).

You can use any file name for the icons and iOS is clever enough to work out the correct icon based on its size, using the most appropriate one accordingly.

In addition, you can specify the retina icons individually (they will have the **@2x** suffix), but if you specify each icon *without the file extension*, then iOS will cleverly and automatically look for the retina icon for you. For example, you could enter just **Icon** and **Icon-iPad** for the file names rather than **Icon.png** and **Icon-iPad.png**, and it would work fine.

You do not have to specify every single icon size, as iOS will automatically scale the icon closest in size if it can't find the exact match. However, Apple recommends you supply at least the main app icons for both iPhone and iPad, if your app supports both devices.

Example:

```
<key>CFBundleIconFiles</key>
<array>
 <string>Icon</string>
 <string>Icon-iPad</string>
 <string>Icon-iPhone-spotlight</string>
 <string>Icon-iPad-spotlight</string>
</array>
```

## CFBundleIcons (Dictionary)

This key was added in iOS 5.0 to enable developers to supply a Newsstand icon along with a normal app icon. Both are required for Newsstand apps, if you want the app to run on both devices that support Newsstand and on those that do not.

Inside the dictionary, the key **CFBundlePrimaryIcon** specifies the normal icon. The value of this key is yet another dictionary! Yes, that's right, you're down three levels of dictionaries now – deep in the depths of icons! Within this dictionary are two keys:

- **CFBundleIconFiles**: The same as the top-level key of the same name, which specifies the icon files for the normal icon. See the section above for full details.
- **UIPrerenderedIcon**: The same as the top-level key of the same name, used to specify if these icons are pre-rendered or if they should have a shine effect added to them. See the section above for full details.

At the second level of the dictionary, to specify the Newsstand icon you need to add the **UINewsstandIcon** key, which points to another dictionary containing the following keys:

- **CFBundleIconFiles**: Again, the same as the top-level key of the same name. See the section above for full details.
- **UINewsstandBindingType**: Specifies the type of binding to add to the icon. Bindings make Newsstand look more like a real newsstand by adding a familiar look to each of the icons (Note: all Newsstand apps appear within a special folder on iOS). The binding can be one of two types, designed to make the icon look like either a magazine or a newspaper:
  - `UINewsstandBindingTypeMagazine`
  - `UINewsstandBindingTypeNewspaper`
- **UINewsstandBindingEdge**: Specifies the edge where the binding effect is added to the icon. There are three options that are self-explanatory:
  - `UINewsstandBindingEdgeLeft`
  - `UINewsstandBindingEdgeRight`
  - `UINewsstandBindingEdgeBottom`

Here is a screenshot of the property list editor once all these fields have been filled in:

| Key                             | Type       | Value                 |
|---------------------------------|------------|-----------------------|
| Icon files (iOS 5)              | Dictionary | (2 items)             |
| Primary Icon                    | Dictionary | (2 items)             |
| Icon files                      | Array      | (6 items)             |
| Item 0                          | String     | Icon                  |
| Item 1                          | String     | Icon-iPad             |
| Item 2                          | String     | Icon-iPhone-spotlight |
| Item 3                          | String     | Icon-iPad-spotlight   |
| Item 4                          | String     | Icon.png              |
| Item 5                          | String     | Icon@2x.png           |
| Icon already includes gloss eff | Boolean    | YES                   |
| Newsstand Icon                  | Dictionary | (3 items)             |
| Icon files                      | Array      | (1 item)              |
| Item 0                          | String     | Icon-Newsstand        |
| Binding edge                    | String     | Left                  |
| Binding type                    | String     | Magazine              |

Example:

```

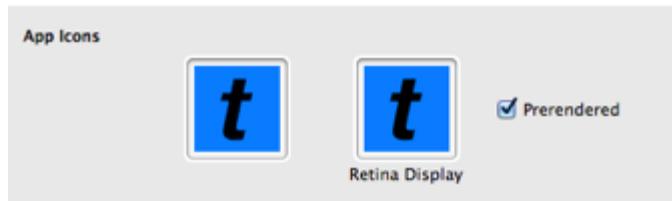
<key>CFBundleIcons</key>
<dict>
 <key>CFBundlePrimaryIcon</key>
 <dict>
 <key>CFBundleIconFiles</key>
 <array>
 <string>Icon</string>
 <string>Icon-iPad</string>
 <string>Icon-iPhone-spotlight</string>
 <string>Icon-iPad-spotlight</string>
 </array>
 </dict>
</dict>

```

```
<key>UIPrerenderedIcon</key>
<false/>
</dict>
<key>UINewsstandIcon</key>
<dict>
<key>CFBundleIconFiles</key>
<array>
<string>Icon-Newsstand</string>
</array>
<key>UINewsstandBindingType</key>
<string>UINewsstandBindingTypeMagazine</string>
<key>UINewsstandBindingEdge</key>
<string>UINewsstandBindingEdgeLeft</string>
</dict>
</dict>
```

**Note:** Remember that you should use the keys matching the iOS versions that you support. For example, if you support 4.0 and up, then you should include both `CFBundleIconFiles` and `CFBundleIcons`.

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You'll find a section to set the App Icons (and whether they are prerendered or not) there:



## Controlling initial app launch

### UILaunchImageFile (String)

By default (no pun intended, you'll see!) the launch image, sometimes referred to as the "splash screen," is named **Default.png**. This is the image displayed while the app is launching.

However, you can set this key to specify any file of your choice instead of `Default.png`. You don't have to set this key, but it can help with organization of files. Plus, it causes less confusion among your graphic designers when you ask for an image called `Default.png`!

iPad chooses different splash screens depending on the orientation in which the app is launched. On the iPad, the name you specify for this key is automatically suffixed with **-Portrait** and **-Landscape** for their respective launch orientations.

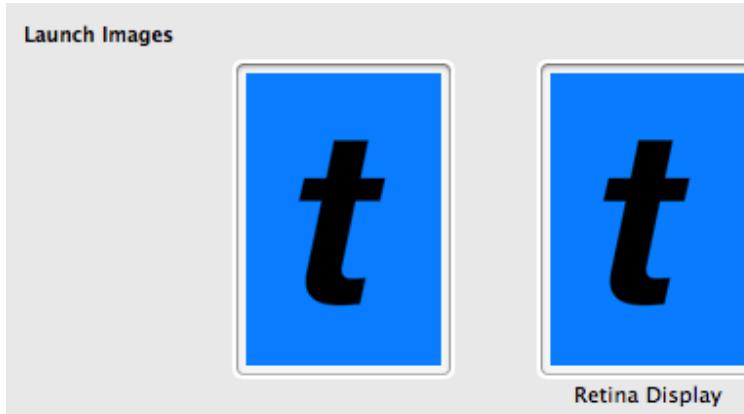
This key is particularly useful when combined with device-specific key settings, such that you can specify the filename for iPad to be different than that for iPhone. For example, if you specify the iPad key as **LaunchImageiPad**, then your files for both orientations would be:

- **LaunchImageiPad-Portrait.png**
- **LaunchImageiPad-Landscape.png**

Example:

```
<key>UILaunchImageFile</key>
<string>LaunchImageiPhone</string>
<key>UILaunchImageFile~ipad</key>
<string>LaunchImageiPad</string>
```

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You'll find a section for setting the launch images there:



### NSMainNibFile (String)

This is an important key if you use NIBs. It specifies the NIB file that will be used to create the initial window and associated objects when your app launches (if the fourth parameter of `UIApplicationMain` is nil).

Xcode creates a NIB called **MainWindow.xib** by default when you create a new project without Storyboards enabled. So you probably won't need to change this key. However, sometimes you might be starting a project from scratch – that's a good case where you might need to set this yourself.

Sometimes you might want to have a different MainWindow.xib for the iPhone and the iPad. You can specify a device-specific key for the iPad (i.e., **NSMainNibFile~ipad**), which would then load a file other than the default, as you can see below:

```
<key>NSMainNibFile</key>
<string>MainWindow_iPhone</string>
<key>NSMainNibFile~ipad</key>
<string>MainWindow_iPad</string>
```

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You'll find a field for setting the main interface (XIB file) there:



### UIMainStoryboardFile (String)

This is an important key if you use storyboards. It specifies the storyboard file that will be loaded during app launch to create the initial interface.

Again, Xcode creates a storyboard called **MainStoryboard.storyboard** by default when you create a new project, so you probably won't need to change this key unless you are starting a project from scratch.

However, you may find it useful to create different storyboard for the iPhone and iPad. You can specify a device-specific key for the iPad (i.e., **UIMainStoryboardFile~ipad**) or for the iPhone (i.e. **UIMainStoryboardFile~iphone**), as you can see below:

```
<key>UIMainStoryboardFile</key>
<string>MainStoryboard_iPhone</string>
<key>UIMainStoryboardFile~ipad</key>
<string>MainStoryboard_iPad</string>
```

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You'll find a field for setting the Main Storyboard there:



## Reason strings for accessing user information

It is becoming more and more common for software manufacturers to ask permission from the user before allowing apps to access the user's information. In previous versions of iOS, permission was only needed to access the user's current

location. With iOS 6.0, four more sets of data require permission from the user for an app to gain access. The full list is now:

- Calendars
- Contacts
- Location
- Photos
- Reminders

Apple has kindly allowed us developers to provide some text that is displayed along with the permission request, so that users can understand *why* an app needs access to their data.



This is your chance to explain to the user why the experience will be better if they say yes, so users aren't tempted to just say no to everything. ☺ These strings are all found in the Info.plist under the following keys.

### **NSCalendarsUsageDescription (String)**

The first time an app tries to access the user's calendar, an alert pops up asking for permission. If you give this key a value, then that value is shown inside the alert.

For example, say you gave the string the value, "To show your appointments." The alert will end up looking like this:



Example:

```
<key>NSCalendarsUsageDescription</key>
<string>To show your appointments.</string>
```

### NSContactsUsageDescription (String)

The first time an app tries to access the user's contacts, an alert pops up asking for permission. As with the previous key, whatever you provide as the value of this key will be shown inside the alert.

For example, say you set the key to "To make calls to your friends." The alert will then look like this:



Example:

```
<key>NSContactsUsageDescription</key>
<string>To make calls to your friends.</string>
```

### NSLocationUsageDescription (String)

The first time an app tries to access the user's location using either the `CLLocationManager` API or by turning on user tracking in an `MKMapView`, an alert pops up asking for permission. Whatever you set as the value of this key will be shown inside the alert.

For example, say you set the string to "To help you find your way." The alert will then look like this:



Example:

```
<key>NSLocationUsageDescription</key>
<string>To help you find your way.</string>
```

### NSPhotoLibraryUsageDescription (String)

The first time an app tries to access the photos stored in the user's photo library, an alert pops up asking for permission. Whatever you set as the value of this key will be shown inside the alert.

For example, say you set the string to "To upload your photos to share with friends." The alert will then look like this:



Example:

```
<key>NSPhotoLibraryUsageDescription</key>
<string>To upload your photos to share with friends.</string>
```

### NSRemindersUsageDescription (String)

The first time an app tries to access the user's reminders, an alert pops up asking for permission. Whatever you set as the value of this key will be shown inside the alert.

For example, say you set the string to "To help organize your files." The alert will then look like this:



Example:

```
<key>NSRemindersUsageDescription</key>
<string>To help organize your life.</string>
```

## Custom URLs and document types

There are a couple of ways you can make your app interact with other apps. One is to provide a set of URL schemes that your app supports, so that just as `http://` opens the browser, `yourapp://` will open your app.

The second way is to supply a list of documents that the app supports opening. Then when the user requests to open a file of that type, they will be asked if they want to launch your app.

I'll explain both of these methods in the following sections.

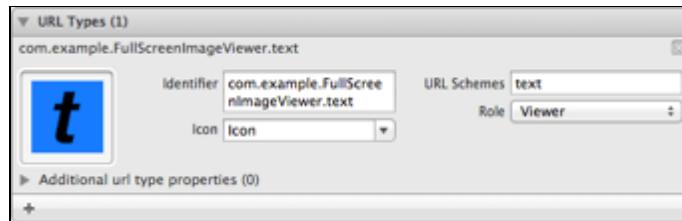
### CFBundleURLTypes (Array)

This key enables you to supply an array of URL schemes that your app supports. The array contains dictionaries, one for each type of URL scheme that you support, with the following keys:

- **CFBundleURLSchemes:** This should be an array of strings, one for each URL scheme supported in this type. Most often you will supply just one for each type, but if it makes sense to group schemes into types because they are similar, then you can do that.
- **CFBundleTypeRole:** Specifies the role your app plays when opened with this type of URL. The options are **Editor**, **Viewer**, **Shell** or **None**. It doesn't particularly matter what you put here, so most apps will choose Viewer.
- **CFBundleURLIconFile:** Specifies the filename of the icon to be used for this type of URL.
- **CFBundleURLName:** This should be a unique string used by iOS to distinguish between different URL types. You should use the same reverse domain name approach that you used for `CFBundleIdentifier`. For example, `com.example.HelloWorld.urlscheme`.

There is also an easy way to add URL types using the Xcode graphical Info.plist editor. To find this, click on your project root at the top of the project navigator and select the target in the box that appears on the right.

Then select the Info tab and scroll to the bottom, where you will find a "URL Types" box. Expand that, and you will find a nice, graphical way of adding a URL type. It will look like this:



Example:

```
<key>CFBundleURLTypes</key>
<array>
 <dict>
 <key>CFBundleTypeRole</key>
 <string>Viewer</string>
 <key>CFBundleURLIconFile</key>
 <string>Icon</string>
 <key>CFBundleURLName</key>
 <string>com.example.FullScreenImageViewer.text</string>
 <key>CFBundleURLSchemes</key>
 <array>
 <string>text</string>
 </array>
 </dict>
</array>
```

## CFBundleDocumentTypes (Array)

This key enables you to state which types of documents your app can handle opening. This means, for example, that if you support text documents, then if a user has a text file sent to them via email, the Mail app will let them open the file directly in your app.

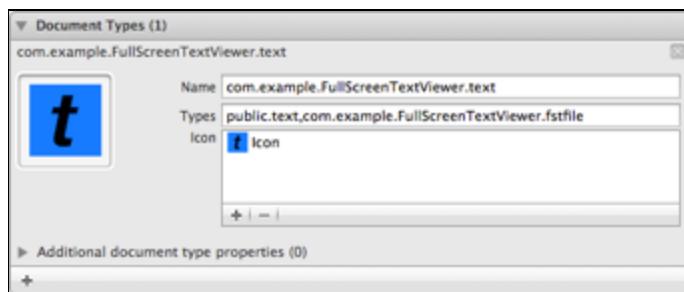
Each value in the array is a dictionary containing the following keys:

- **CFBundleTypeIconFiles:** This is an array of icon files defined in the same way as icon files in the `CFBundleIconFiles` key.
- **CFBundleTypeName:** This should be a unique string used by iOS to distinguish between different URL types. So you should use the same reverse domain notation that you used for `CFBundleIdentifier`. For example, `com.example.HelloWorlddoctype`.

- **LSItemContentTypes:** This defines a list of Uniform Type Identifiers (UTIs) supported with this document type. I explain more about these in the next section.
- **LHandlerRank:** This is a way to specify your app's rank for a given document type. There are a few options for this, but the main two you might want to choose from are **Owner**, meaning that the app is the primary creator, and **Alternate**, meaning that the app is not the main creator but does handle this document type.

There is also an easy way to add document types using the Xcode graphical Info.plist editor. To find this, click on your project root at the top of the project navigator and then select the target in the box that appears on the right.

Then select the Info tab and scroll to the bottom, where you'll find a "Document Types" box. Expand that, and you will find a nice graphical way of adding a document type. It will look like this:



Example:

```
<key>CFBundleDocumentTypes</key>
<array>
 <dict>
 <key>CFBundleTypeIconFiles</key>
 <array>
 <string>Icon</string>
 </array>
 <key>CFBundleTypeName</key>
 <string>com.example.FullScreenTextViewer.text</string>
 <key>LSItemContentTypes</key>
 <array>
 <string>public.text</string>
 </array>
 </dict>
 <key>LHandlerRank</key>
 <string>Owner</string>
</array>
```

## UTEportedTypeDeclarations (Array)

Uniform Type Identifiers (UTIs) are what iOS uses to identify different types of data, whether it's in a file, on a pasteboard, or any other blob of data. You can read more about UTIs here:

[https://developer.apple.com/library/ios/#documentation/FileManagement/Conceptual/understanding\\_utis/understand\\_utis\\_conc/understand\\_utis\\_conc.html](https://developer.apple.com/library/ios/#documentation/FileManagement/Conceptual/understanding_utis/understand_utis_conc/understand_utis_conc.html)

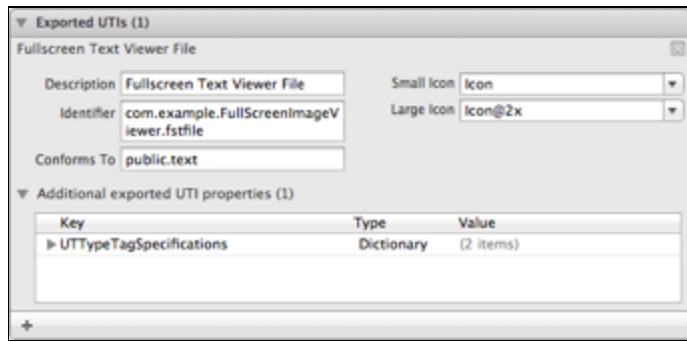
Here are the basics:

There are a set of system-defined types such as `public.image`, `public.text`, and `public.movie`. But you can also add your own for-use-in-document types (see `CFBundleDocumentTypes` above). To do this, the `UTEportedTypeDeclarations` key should be populated with an array containing a dictionary representing each UTI. Within this dictionary should be the following keys:

- **UTTypeIdentifier:** This is the identifier that you would put in the `LSItemContentTypes` field of a `CFBundleDocuments` types dictionary. It should be a unique identifier for this UTI. Usually, for custom UTIs, it is a reverse domain-style identifier. For example: `com.example.HelloWorld.customtype`.
- **UTTypeTagSpecification:** The value for this key should be a dictionary of key-value mappings describing the various filename extensions and MIME types that this UTI refers to. To specify a filename extension, you would add a key to the dictionary called `public.filename-extension` whose value was the extension. For example: `myfile`. To specify a MIME type, you would add a key called `public.mime-type` whose value was the MIME type. For example: `application/x-myfile`.
- **UTTypeConformsTo:** UTIs are arranged into hierarchies. For example, `public.jpeg`'s parent is `public.image` whose parent is `public.data`. This should be an array containing a list of which UTIs are the parents.
- **UTTypeDescription:** This is a text description. For example, JPEG would be "JPEG Image File."
- **UTTypeSize64IconFile:** The filename of a 64x64 icon to use when displaying files of this type.

There is also an easy way to add exported UTIs using the Xcode graphical Info.plist editor. To find this, click on your project root at the top of the project navigator and then select the target in the box that appears on the right.

Then select the Info tab and scroll to the bottom, where you will find an "Exported UTIs" box. Expand that, and you will find a nice graphical way of adding a UTI. It will look like this:



Example:

```

<key>UTEportedTypeDeclarations</key>
<array>
 <dict>
 <key>UTTypeConformsTo</key>
 <array>
 <string>public.text</string>
 </array>
 <key>UTTypeDescription</key>
 <string>Fullscreen Text Viewer File</string>
 <key>UTTypeIdentifier</key>
 <string>com.example.FullScreenImageViewer.fstfile</string>
 <key>UTTypeSize64IconFile</key>
 <string>Icon</string>
 <key>UTTypeTagSpecification</key>
 <dict>
 <key>public.filename-extension</key>
 <string>fst</string>
 <key>public.mime-type</key>
 <string>application/x-fullscreeentext</string>
 </dict>
 </dict>
</array>

```

## UTImportedTypeDeclarations (Array)

To allow your app to open a file of a type that it doesn't specifically own, you need to do what is known as importing that UTI, and this is the key you use.

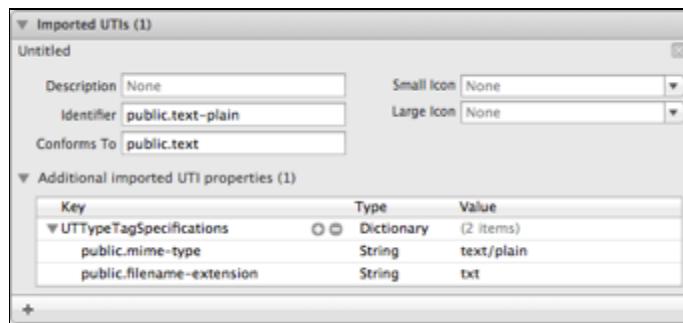
The key's value is an array just like `UTEportedTypeDeclarations`, but this time containing the UTIs that are not owned by your app but that the app can open. For example, you may import the plain text UTI known as `public.text-plain`.

You can find a list of the system-defined UTIs that you might want to import here:

[https://developer.apple.com/library/ios/#documentation/Miscellaneous/Reference/UTIRef/Articles/System-DeclaredUniformTypeIdentifiers.html#/apple\\_ref/doc/uid/TP40009259-SW1](https://developer.apple.com/library/ios/#documentation/Miscellaneous/Reference/UTIRef/Articles/System-DeclaredUniformTypeIdentifiers.html#/apple_ref/doc/uid/TP40009259-SW1)

There is also an easy way to add imported UTIs using the Xcode graphical Info.plist editor. To find this, click on your project root at the top of the project navigator, and select the target in the box that appears on the right.

Then select the Info tab and scroll to the bottom, where you will find an “Imported UTIs” box. Expand that, and you will find a nice graphical way of adding a UTI. It will look like this:



Example:

```
<key>UTImportedTypeDeclarations</key>
<array>
 <dict>
 <key>UTTypeConformsTo</key>
 <array>
 <string>public.text</string>
 </array>
 <key>UTTypeIdentifier</key>
 <string>public.text-plain</string>
 <key>UTTypeTagSpecification</key>
 <dict>
 <key>public.filename-extension</key>
 <string>txt</string>
 <key>public.mime-type</key>
 <string>text/plain</string>
 </dict>
 </dict>
</array>
```

## Example using the sample project

To see the URL and document type handling in action, open the **FullScreenTextViewer** project and navigate to the graphical Info.plist editor. Then add the following URL type:

| Field              | Value                                  |
|--------------------|----------------------------------------|
| <b>Identifier</b>  | com.example.FullScreenImageViewer.text |
| <b>URL Schemes</b> | Text                                   |
| <b>Icon</b>        | Icon.png                               |
| <b>Role</b>        | Viewer                                 |

This defines a URL such that your app will be opened when a URL such as this is opened:

```
text://www.raywenderlich.com
```

To test it out, run the app, then press the home button and open Safari. Type in the above URL to the address bar and tap Go. Then watch the app open and display the contents of [www.raywenderlich.com](http://www.raywenderlich.com)!



Next, I'll show you how to support opening text files and a custom file type with extension **fst** (standing for Full Screen Text!). First, add the following custom *exported* UTI:

| Field              | Value                                     |
|--------------------|-------------------------------------------|
| <b>Description</b> | Full Screen Text Viewer File              |
| <b>Identifier</b>  | com.example.FullScreenImageViewer.fstfile |

|                                                                |                                                                                                        |
|----------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <b>Conforms To</b>                                             | public.text                                                                                            |
| <b>Small Icon</b>                                              | Icon.png                                                                                               |
| <b>Additional:<br/>UTTypeTagSpecification<br/>(Dictionary)</b> | <pre>public.mime-type =&gt;     application/x-fullscreentext public.filename-extension =&gt; fst</pre> |

Then add the following *imported* UTI:

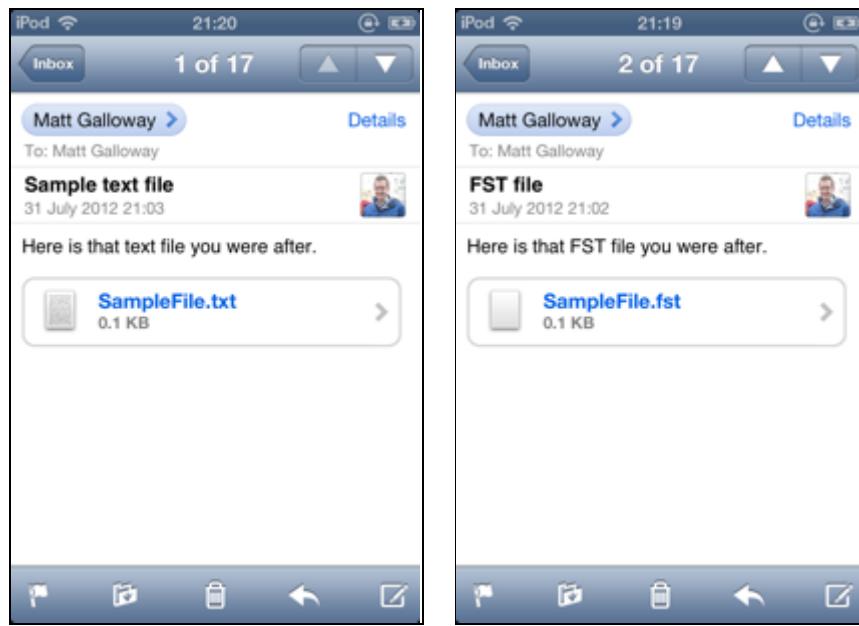
| Field                                                          | Value                                                                                |
|----------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <b>Description</b>                                             | Plain Text File                                                                      |
| <b>Identifier</b>                                              | public.text-plain                                                                    |
| <b>Conforms To</b>                                             | public.text                                                                          |
| <b>Additional:<br/>UTTypeTagSpecification<br/>(Dictionary)</b> | <pre>public.mime-type =&gt;     text/plain public.filename-extension =&gt; txt</pre> |

And finally, add the following document type:

| Field        | Value                                                 |
|--------------|-------------------------------------------------------|
| <b>Name</b>  | com.example.FullScreenTextViewer.text                 |
| <b>Types</b> | public.text, com.example.FullScreenTextViewer.fstfile |
| <b>Icon</b>  | Icon.png                                              |

Now send yourself a couple of emails with some text file attachments, one named **SampleFile.txt** and one named **SampleFile.fst**. Use your favorite text editor to create both files, and write whatever you want in them: a bawdy limerick, a to-do list, or just a bunch of random characters.

When you're done, build and run the app, then press the home button and open your email. You should see something like this for each email:



If you then tap on the files within each email, you will be taken to a new screen. At the top-right will be an action button. Tap that and you will see a list of actions that you can take with the file – this is the new Activity View Controller as discussed in Chapter 11, “Beginning Social Framework”.

One of the options in the list will be “Open in FullScreenText”:



When you tap on the “Open in FullScreenText” button, you will find that your app opens and the text from that file is displayed! Woop!

## Working with iCloud

### NSUbiquitousDisplaySet (String)

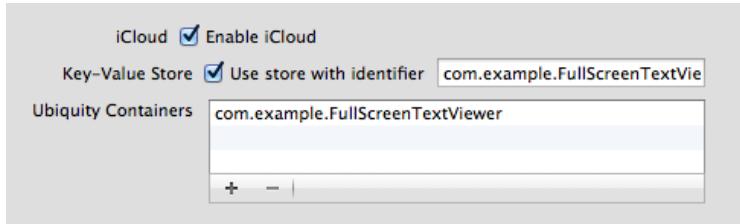
iCloud uses this string as the key indicating your app's file store. Think of it as the folder in iCloud where your app's files are stored.

You can set this key to the same value across multiple apps of your own, such that the same data files can be shared between apps. This would be useful if, for example, you created a "lite" and "pro" version of your app and wanted the data to be shared between them.

Example:

```
<key>NSUbiquitousDisplaySet</key>
<string>com.example.CoolApp</string>
```

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You'll find a section for the iCloud options there:



For more information on iCloud, check out "Beginning and Intermediate iCloud" in *iOS 5 by Tutorials*.

## Core OS

You should generally not need to modify any of the keys in this section, as Apple regulates them for you, but to satisfy your curiosity and be a true plist professor, read on!

### LSRequiresiPhoneOS (Boolean)

This key is required for iOS apps to be launched on a device, and it *must* be set to **true**. It doesn't make much sense in the context of apps that are purely for iOS, but it does when you consider that the Info.plist has come from the Mac, and so LSRequiresIPhoneos is a key that specifies that this app is for iOS rather than Mac.

Example:

```
<key>LSRequiresiPhoneOS</key>
<true/>
```

## CFBundlePackageType (String)

In iOS, you deal only in apps, whereas on the Mac there are other types of items to which an Info.plist can refer such as frameworks. For apps, the string `APPL` is required for this key. The good news is, you don't need to worry about adding it yourself, as it's already included by default when you create a new project in Xcode.

Example:

```
<key>CFBundlePackageType</key>
<string>APPL</string>
```

## CFBundleExecutable (String)

An app on iOS (and Mac for that matter) is packaged as a `.app` file. Within that file are all your resources (images, sound files, NIBs, etc.) and a file referred to as the executable. This key is a string that specifies the name of that executable file.

Don't worry, though, because Xcode adds this for you by default, and uses a template replacement variable `{EXECUTABLE_NAME}` that is automatically set to the output that Xcode creates when compiling. To put it another way... just leave this setting alone!

Example:

```
<key>CFBundleExecutable</key>
<string>${EXECUTABLE_NAME}</string>
```

## CFBundleInfoDictionaryVersion (String)

It's possible that the structure of the Info.plist will change over time, and there needs to be a way to tell the OS what version of the file it's currently working with. That's what this key does.

At the time of writing, the Info.plist is version 6.0. Note that this does not refer to the iOS version – it is a mere coincidence that the current version of iOS matches the current version of the Info.plist.

You should not ever need to change this value, because if the version ever did get bumped, then no doubt Apple would provide a mechanism for automatic upgrading of the file for us.

Example:

```
<key>CFBundleInfoDictionaryVersion</key>
<string>6.0</string>
```

## CFBundleSignature (String)

This key is a bit of a throwback to the Mac, and not particularly relevant on iOS. However, it is still present. It is meant as a key to distinguish apps, but now that there is the `CFBundleIdentifier` key, this key is redundant.

Xcode sets this up for you, defaulting to the string, “????”. You should not ever need to change this.

Example:

```
<key>CFBundleSignature</key>
<string>????</string>
```

## Localization

### CFBundleLocalizations (Array)

The system determines which language regions your app supports by looking at the **Iproj** folders supplied with your application. These are individual folders providing the specific assets (strings files, NIBs, etc.) for each language.

However, you can override which languages the system thinks you support by using this key. If you do include this key, then its value takes precedence over which Iproj folders you supplied.

The value of this key should be an array of strings, one for each supported region. These strings can either be language names (e.g., English, French, Spanish) or region identifiers using ISO 639-1 language designators (e.g., `en` for English, `fr` for French, `es` for Spanish), and optional ISO 3166-1 regional qualifiers (e.g., `GB` for Great Britain, `us` for United States and `ca` for Canada).

For example, to support both French and Canadian French, you would give `fr_FR` and `fr_CA`.

Example:

```
<key>CFBundleLocalizations</key>
<array>
 <string>en_GB</string>
 <string>en_US</string>
 <string>Spanish</string>
</array>
```

### CFBundleDevelopmentRegion (String)

When you develop your app, you will generally be writing it in your native localization. The value of this key is a string indicating the localization region (e.g.,

`en_us` for US English or `es` for Spanish) to be used as the default if the user's requested region does not have an asset for the required resource.

Usually, you will set this to your native region, but you only need to set it if you are localizing your app.

Example:

```
<key>CFBundleDevelopmentRegion</key>
<string>en_GB</string>
```

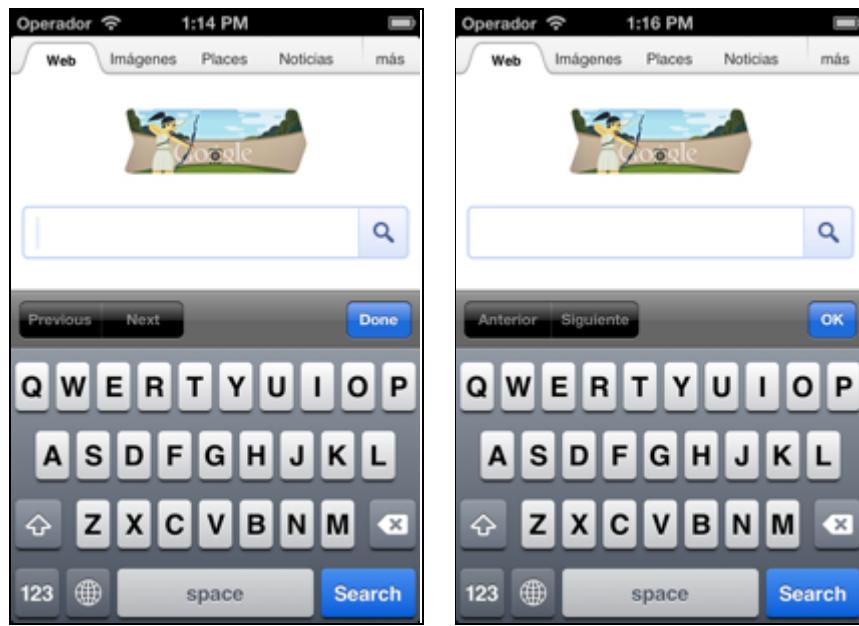
### CFBundleAllowMixedLocalizations (Boolean)

Certain parts of the system frameworks are already localized for you. A good example of this is in a web view, where the buttons that become attached to the keyboard when entering details in a web text field (Previous, Next and Done) have already been localized by Apple.

However, by default the system will only show the localizations for the regions that you *specifically* support in your app. If you support English and Spanish, say, then only those localizations will be automatically supported by the web view. If the user is set to a French localization, then they will get the strings for the default region (as specified by your `CFBundleDevelopmentRegion`, explained earlier).

To get around this is, set the value of the `CFBundleAllowMixedLocalizations` key to **true**. Then system frameworks will support all localizations that they themselves support. An example of this is shown below.

In the example, the system region is set to Spanish. The images show an app running that supports only English. In the left screenshot, mixed localizations is turned off, while in the right screenshot, it's turned on. Notice how the Previous, Next and Done buttons are localized on the right, but not on the left.



Example:

```
<key>CFBundleAllowMixedLocalizations</key>
<true/>
```

## User Interface

### UISupportedInterfaceOrientations (Array)

The value of this key is an array of orientations that your app supports. The options are:

- **UIInterfaceOrientationPortrait**: The device is in portrait with the home button at the bottom.
- **UIInterfaceOrientationPortraitUpsideDown**: The device is in portrait with the home button at the top.
- **UIInterfaceOrientationLandscapeLeft**: The device is in landscape with the home button on the left.
- **UIInterfaceOrientationLandscapeRight**: The device is in landscape with the home button on the right.

Note that each of your view controllers can limit this further, if required. You might want one of your view controllers to only appear in portrait, for example, even though the rest of the app supports landscape as well.

This key is a good example where you might want to set device-specific values in cases where you want a different set of orientations on iPhone than on iPad.

Example:

```
<key>UISupportedInterfaceOrientations</key>
<array>
 <string>UIInterfaceOrientationPortrait</string>
</array>
<key>UISupportedInterfaceOrientations~ipad</key>
<array>
 <string>UIInterfaceOrientationPortrait</string>
 <string>UIInterfaceOrientationLandscapeLeft</string>
 <string>UIInterfaceOrientationLandscapeRight</string>
</array>
```

Note that you can edit this visually by clicking on your project in the Project Navigator, selecting your target, and selecting the Summary tab. You can easily click on the orientations your app supports there (dark grey means supported):



### UIInterfaceOrientation (String)

If your app supports only one orientation, then you should set this key to indicate that fact. This will ensure that the status bar is in the correct position when the app is launched, rather than starting with it in one place and then animating it around.

For example, if you support just landscape mode, then you should pick either landscape-left or landscape-right (either will do) and use that for this key. Then when the app is launched, the display will be put in the correct orientation immediately.

Example:

```
<key>UIInterfaceOrientation</key>
<string>UIInterfaceOrientationLandscapeRight</string>
```

### UIStatusBarHidden (Boolean)

This key indicates whether or not the status bar is initially hidden during app launch. It is most common to set this key to the same state you use in your main interface. However, you are free to then use `UIApplication's setStatusBarHidden:animated:` method to show and hide the status bar visibility after app launch.

Example:

```
<key>UIStatusBarHidden</key>
<true/>
```

## UIStatusBarStyle (String)

This key sets the initial style of the status bar during launch. In general, you want to set this to the same style as your initial app interface. You can choose from one of the following values, corresponding to the `UIStatusBarStyle` enumeration values with the same names:

- **UIStatusBarStyleDefault**
- **UIStatusBarStyleBlackTranslucent**
- **UIStatusBarStyleBlackOpaque**

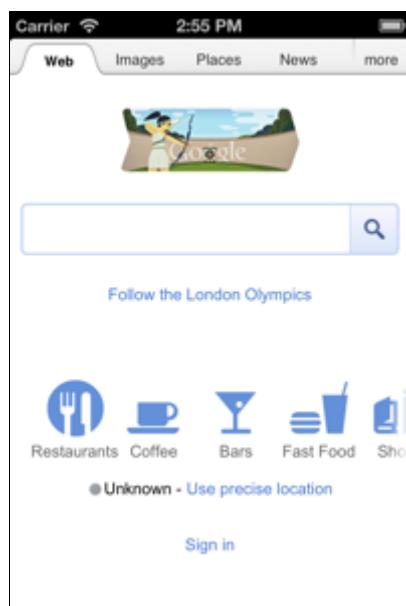
Example:

```
<key>UIStatusBarStyle</key>
<string>UIStatusBarStyleBlackTranslucent</string>
```

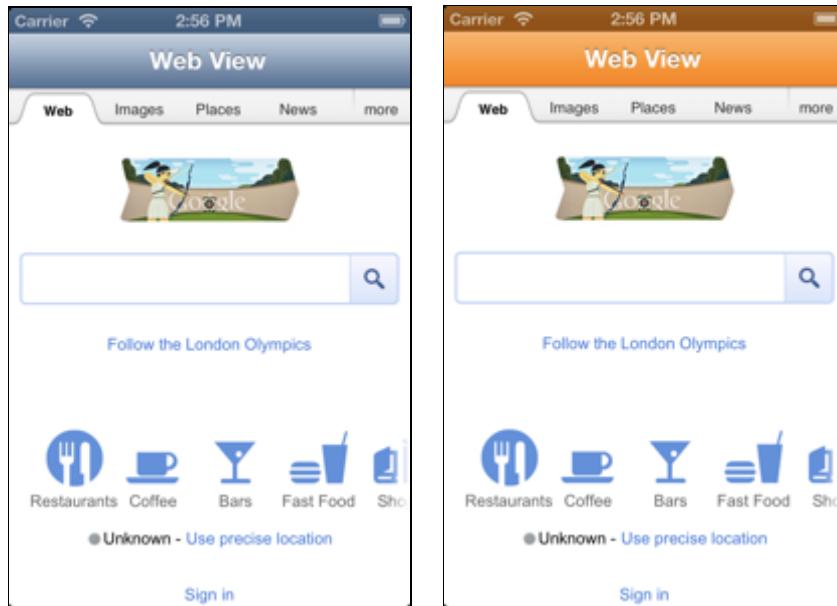
## UIStatusBarTintParameters (Dictionary)

The status bar got a bit of a makeover with iOS 6.0. It will now change color to best match your app's color scheme if its style is set to **UIStatusBarStyleDefault**. If its style is **UIStatusBarStyleBlackTranslucent** or **UIStatusBarStyleBlackOpaque**, then it won't automatically match your app color scheme.

This tinting is best demonstrated with some examples. The first screenshot below shows the status bar in the default style in an app that has no navigation bar. Notice how the status bar has been automatically set to black color – probably because that's what looks best with this interface.



The following two screenshots show the same app, but this time with a navigation bar. The left screenshot shows the navigation bar with no specific tint color, i.e., it uses the standard color. The right screenshot shows the navigation bar with an orange tint.



Notice how the status bar takes on a color similar to the navigation bar, such that it fits in with the look and feel of the app. In fact, the color it uses to tint itself is the average color of the bottom pixel of the navigation bar. It will even perform this tinting if you set an image as the background of the navigation bar. Neat, hey!

But you may be wondering what this has to do with the `UIStatusBarTintParameter` key, which is what this section is about. Since the OS has no way of knowing what navigation bar will be shown initially, this key specifies the style of that navigation bar.

The value of `UIStatusBarTintParameter` should be a dictionary with one key called **`UINavigationBar`** whose value is also a dictionary. Inside that dictionary are four keys:

- **Style:** This refers to the `barStyle` property of the navigation bar, i.e., one out of:
  - `UIBarStyleDefault`
  - `UIBarStyleBlack`
- **Translucent:** This refers to the `translucent` property of the navigation bar.
- **TintColor:** This specifies the tint color of the navigation bar. It is itself a dictionary containing three keys, `Red`, `Green` and `Blue`, whose values are the components of the tint color in the range 0.0 to 1.0. For example, red would be a red value of 1.0 and green and blue values of 0.0.
- **BackgroundImage:** If your navigation bar has a background image, then put the filename of the image here.

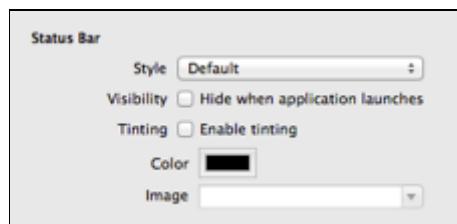
If these are all set up correctly, then the status bar will be shown at launch exactly as it will when the app has finished loading.

Example:

```
<key>UIStatusBarTintParameters</key>
<dict>
 <key>UINavigationBar</key>
 <dict>
 <key>Style</key>
 <string>UIBarStyleDefault</string>
 <key>TintColor</key>
 <dict>
 <key>Blue</key>
 <real>0.016740924636483221</real>
 <key>Green</key>
 <real>0.0</real>
 <key>Red</key>
 <real>0.71015200407608692</real>
 </dict>
 <key>Translucent</key>
 <false/>
 </dict>
</dict>
```

## Status bar parameters in the Xcode GUI

All of these status bar parameters can be set from Xcode's Info.plist editor. To find it, click on your project root at the top of the project navigator and then select the target in the box that appears on the right. Then select the Summary tab and scroll to the section called "Status Bar." It will look like this:



You can set all the parameters for the initial status bar here. The Color and Image parameters refer to the tint color and background image of the navigation bar in your initial UI, respectively.

## App control

### UIRequiredDeviceCapabilities (Array or Dictionary)

iOS devices come in many flavors – some have a GPS, some don't. Some have a camera, some don't, and yet others have two cameras. You get the picture.

Some apps require the presence of certain device features (or specific hardware) in order to function properly. For example, it would be no good running a video conferencing app on a device that had no camera and no microphone!

That's where the `UIRequiredDeviceCapabilities` key comes in. It allows you to specify which features a device must have, or which features a device must *not* have. To give a list of the features a device must have, you can use an array as the value of this key. The array will contain a list of strings representing the features (more on those in a second).

If you want to request that certain features must *not* be present, then you have to use the dictionary approach. In this case, provide a dictionary containing the features as keys, and each value as **true** for each feature that must be present, and **false** for each feature that must *not* be present.

Here are the various features and their meanings, broken into categories:

#### General

|            |                                                      |
|------------|------------------------------------------------------|
| telephony  | Presence of telephony services, i.e., the Phone app. |
| wifi       | Presence of a WiFi chip to connect to networks.      |
| sms        | The ability to send and receive SMS messages.        |
| gamekit    | Presence of the GameCenter app.                      |
| microphone | Presence of a microphone.                            |

#### Camera

|                     |                                                      |
|---------------------|------------------------------------------------------|
| still-camera        | Presence of a camera capable of taking still images. |
| auto-focus-camera   | Presence of a camera with auto-focus capability.     |
| front-facing-camera | Presence of a front-facing camera.                   |

|              |                                                            |
|--------------|------------------------------------------------------------|
| camera-flash | Presence of a flash for use with picture and video taking. |
| video-camera | Presence of a camera capable of recording videos.          |

## Device motion & location

|                   |                                                                                                   |
|-------------------|---------------------------------------------------------------------------------------------------|
| accelerometer     | Presence of an accelerometer used for rough device orientation detection.                         |
| gyroscope         | Presence of a gyroscope used for device rotation rate detection and better orientation detection. |
| magnetometer      | Presence of a magnetometer used for detecting magnetic heading like a compass.                    |
| location-services | Ability to perform location services for location detection.                                      |
| gps               | Presence of a GPS device for fine-grained location detection.                                     |

## Processor

|            |                                                       |
|------------|-------------------------------------------------------|
| armv6      | The CPU should understand the v6 ARM instruction set. |
| armv7      | The CPU should understand the v7 ARM instruction set. |
| opengles-1 | The GPU should be capable of using OpenGL ES 1.1.     |
| opengles-2 | The GPU should be capable of using OpenGL ES 2.0.     |

## Bluetooth

|              |                                                           |
|--------------|-----------------------------------------------------------|
| peer-peer    | Presence of peer-to-peer Bluetooth connectivity.          |
| bluetooth-le | The Bluetooth hardware is capable of the low energy mode. |

Example array (`armv7` and `gps` both required):

```
<key>UIRequiredDeviceCapabilities</key>
<array>
 <string>armv7</string>
 <string>gps</string>
</array>
```

Example dictionary (`armv7` required, but must not have `gps`):

```
<key>UIRequiredDeviceCapabilities</key>
<dict>
 <key>armv7</key>
 <true/>
 <key>gps</key>
 <false/>
</dict>
```

## UIBackgroundModes (Array)

Normally when the home button is pressed, the currently-running app is put into the background and suspended. This means that it won't get scheduled on the CPU, and so cannot perform any more tasks until it is opened again and resumes functionality.

Some apps, however, need to run in the background to function properly, such as audio-playing apps or voice-over-IP apps. Imagine if your favorite Internet voice-calling app suddenly hung up when you pressed the home button – that would be a terrible user experience!

Apple has allowed these apps to run in the background, and to enable this behavior, you set the value of this key to an array containing strings of all the background modes your app requires. The ones you can choose from are as follows:

- **audio**: The app uses the audio frameworks to play and/or record.
- **location**: The app requires access to the user's location in the background. This should only be used when fine-grained location updates are required.
- **voip**: The app provides a voice-over-IP service and therefore requires Internet connectivity and audio in the background. You do not need to also request the audio mode if you request this. Apps with this mode selected are also auto-started when the device is turned on, so that a connection with the server can be established.
- **newsstand-content**: The app uses the Newsstand APIs to download and process content in the background. This allows an app to be woken when a push comes in to indicate that a new issue is available for download.

- **external-accessory**: The app communicates with an external device using the External Accessory framework.
- **bluetooth-central**: The app communicates with an external device using the CoreBluetooth framework.

These background modes should be requested with caution, as apps using them will be subject to extra scrutiny during review for the App Store. So, only use the ones you absolutely need!

Example:

```
<key>UIBackgroundModes</key>
<array>
 <string>audio</string>
</array>
```

### MKDrectionsApplicationSupportedModes (Array)

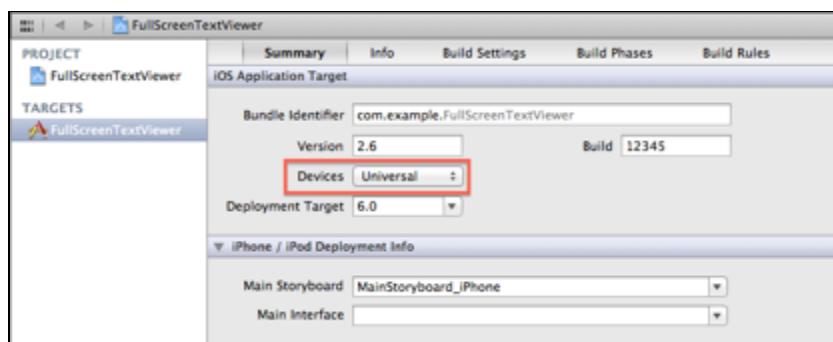
This is a new key added in iOS 6. It allows you to indicate that your app provides routing information for a specific area and transportation mode (such as car, bus, train, plane, etc.).

Example:

```
<key>MKDrectionsApplicationSupportedModes</key>
<array>
 <string>MKDrectionsModePlane</string>
</array>
```

### UIDeviceFamily (Number or Array)

Xcode adds this key automatically, so you should not add it yourself. It signifies what devices are supported – iPhone, iPad or both. To edit this key, use the selection box in the project summary editor, as pictured below.



## UIAppFonts (Array)

iOS comes with many fonts already, but sometimes enough is not enough and you need that one extra font. This key is your chance to add those non-standard fonts so your app can use them.

If you have a font (or fonts) you want to include, first add them to your project and ensure that they are included in the app bundle. Then set the value of this key to an array containing the names of the font files.

Example:

```
<key>UIAppFonts</key>
<array>
 <string>MyAwesomeFont.ttf</string>
</array>
```

## UIApplicationExitsOnSuspend (Boolean)

Since iOS 4.0, an iOS app will move to the background by default when the home button is pressed, rather than being terminated as in previous iOS versions. If this key is set to **true** then the app is terminated instead of being moved to the background.

You may want to include this key if you decide that your app needs to terminate completely when the home button is pressed. But please use with caution, as users these days expect that an app will resume from where they left off when it's re-launched.

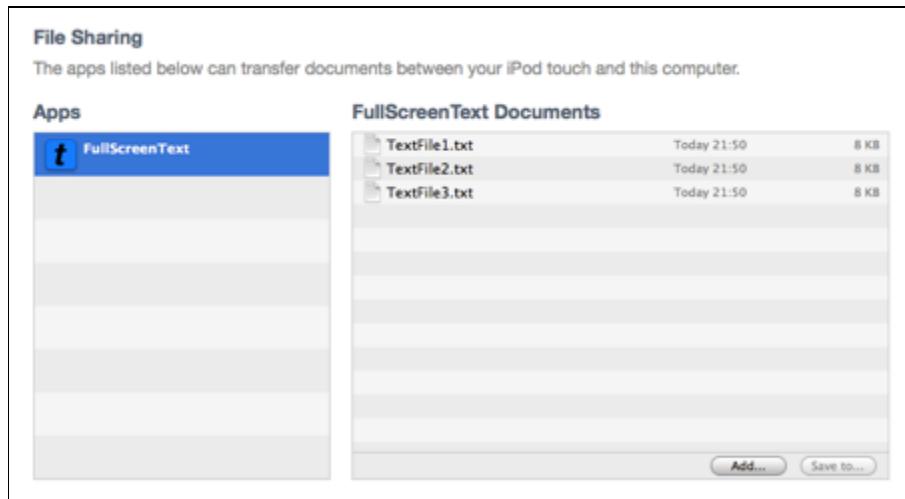
Example:

```
<key>UIApplicationExitsOnSuspend</key>
<false/>
```

## UIFileSharingEnabled (Boolean)

If you would like users to be able to use the iTunes file-sharing option to transfer files to and from your app's documents directory, then you need to set the value of this key to **true**. If you do, then users will be able to use iTunes to drag and drop files into the `<Your_App_Home>/Documents` directory, and even copy them out as well. This feature could provide a document-editing app a means of getting documents into the app, for example.

The screen within iTunes when this is enabled looks like this:



Example:

```
<key>UIFileSharingEnabled</key>
<false/>
```

For more information on using File Sharing within your apps, check out this tutorial:  
<http://www.raywenderlich.com/1948/how-integrate-itunes-file-sharing-with-your-ios-app>

### UINewsstandApp (Boolean)

If your app is a Newsstand app, then you should set this value to **true**. This key is used by SpringBoard to decide whether to show the app in the Newsstand folder.

If you designate your app as a Newsstand app in iTunes Connect, then when you upload your binary, the value of this key must be set to **true**. Otherwise verification of the bundle will fail.

Example:

```
<key>UINewsstandApp</key>
<true/>
```

For more information about Newsstand, check out the "Beginning and Intermediate Newsstand" chapters in *iOS 5 by Tutorials*.

### UIRequiresPersistentWiFi (Boolean)

iOS will, by default, close the WiFi connection on the device if there is no activity for 30 minutes. If you set the value of this key to **true**, then that behavior is overridden and the connection is never closed while your app is open.

You might set this key, for example, if your app is a VoIP app where there could be inactivity for more than 30 minutes, but the connection needs to stay open for the app to function correctly.

Example:

```
<key>UIRequiresPersistentWiFi</key>
<true/>
```

### UISupportedExternalAccessoryProtocols (Array)

This key specifies the protocols supported for communication with attached hardware accessories. As the name indicates, it's used with external accessories, and so you may not need to work with this key very often. But if you do, it simply points to an array that contains a list of communications protocols supported by your external accessory.

Example:

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
 <string>com.example.device.comms</string>
</array>
```

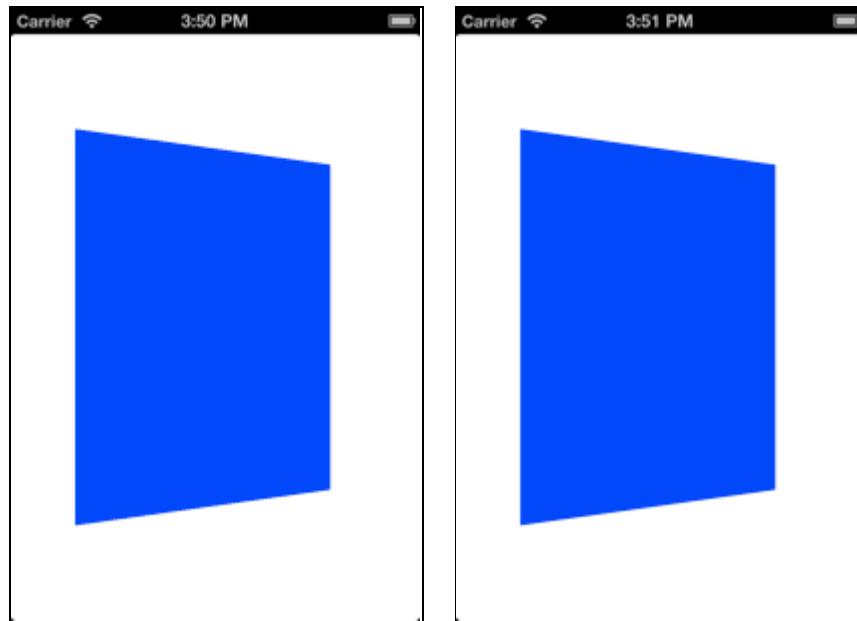
## Advanced view control

### UIViewEdgeAntialiasing (Boolean)

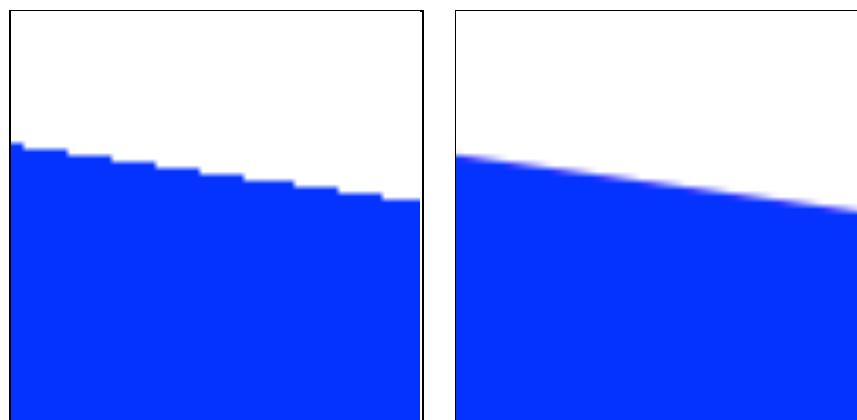
Your views are drawn by the iOS rendering system (CoreAnimation) without edge anti-aliasing by default. This is because it's a lot faster to not deal with antialiasing.

But this means that if you use 3D transformations on your views, then you will see jagged edges. To specify that edges should be anti-aliased during rendering, set the value of this key to **true**. Note that your app will run slower, because the system has to look at what is below every view in order to do the anti-aliasing.

The screenshots below show the effect of edge anti-aliasing. Both screenshots show a view whose transform is set to rotate 45° about the y-axis, which makes it appear as shown. Edge anti-aliasing is turned off in the left image and turned on in the right image.



It's hard to see the difference in these still images, so here is a zoomed view of the edges showing the effect much more clearly:



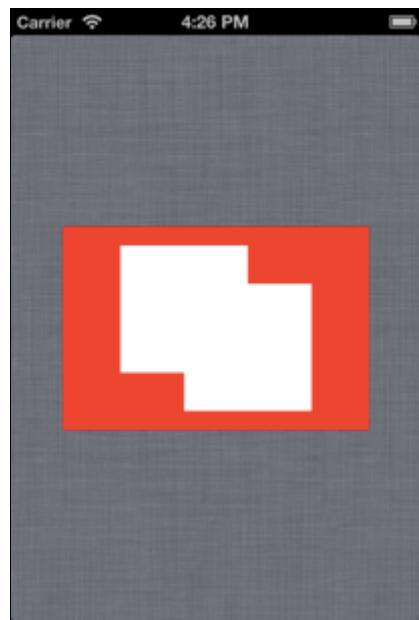
**Note:** Turning on edge anti-aliasing can severely affect performance. Ensure that you test your app thoroughly for performance on a range of devices (so you can see how it performs on older/slower devices as well as on newer/faster ones) if you turn it on.

Example:

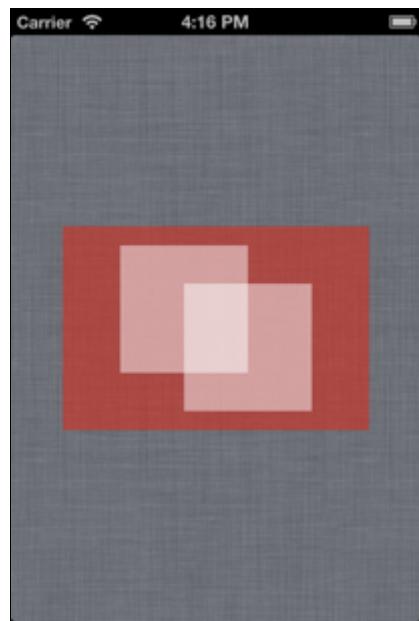
```
<key>UIViewEdgeAntialiasing</key>
<false/>
```

### UIListGroupOpacity (Boolean)

When you set the opacity of a view, any sub-views of that view are rendered with the same opacity, but are rendered afterwards. The effect this has on rendering can be seen in the following example, where a red-colored view has two white sub-views arranged like so:



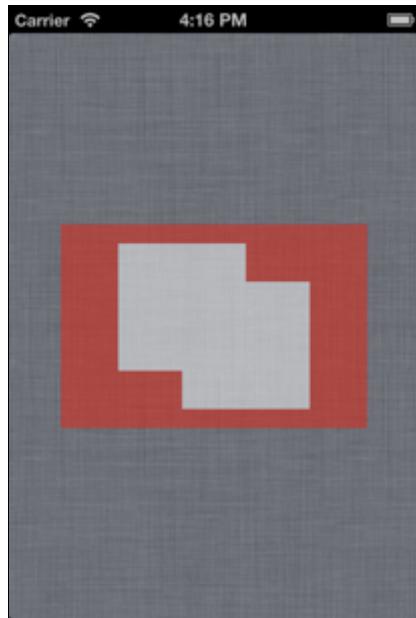
Now if the red view is set to opacity 0.5 while keeping the white views at opacity 1.0, the following default behavior is seen:



The effect is perhaps surprising, but if you think about the order of rendering, it isn't. The red view is rendered first with opacity 0.5 and then the white views are

both rendered with opacity 0.5 on top. This is what happens with **UIViewGroupOpacity** set to **false**.

But the above behavior is probably not what a developer would require in this situation. The developer would probably be after this instead:



Here the entire red view and two white sub-views have been rendered in one go, and *then* the opacity has been applied. This is the result with **UIViewGroupOpacity** set to **true**.

Example:

```
<key>UIViewGroupOpacity</key>
<false/>
```

## Where to go from here?

Believe it or not, that's it – you've learned about all of the possible keys you can set in Info.plist (at least at the time of writing this chapter)!

If you've read this chapter straight through from the beginning, you now know an awful lot about Info.plist, and should feel confident wielding it to do your bidding!

I hope that this chapter has helped you discover options you'd never heard of before and better understand those that might have seemed mysterious. Now I suggest that you go and make better apps by fine-tuning all those little details that so often get forgotten. But no longer by you, of course! ☺

# 27

# Chapter 27: Conclusion

You have done it – you have finally reached the end of this massive tome!

We hope you had a great adventure (and lots of fun) throughout this book! If you were cherry-picking chapters according to your own interests and projects, we hope that you learned a lot and that it helped you start your projects on the right foot. And if you read this entire book from cover to cover – take a bow, you are a beast! ☺

You now have experience with all of the major new APIs in iOS 6, and should be ready to use these cool new technologies in your own apps. If you’re like us, learning this stuff has got you brimming with ideas, and we can’t wait to see what you come up with!

If you have any questions or comments as you go along, please stop by our forums at <http://www.raywenderlich.com/forums>.

Thank you again for purchasing this book. Your continued support is what makes the tutorials, books, and other things we do at raywenderlich.com possible, so we truly appreciate it!

Best of luck with your iOS adventures,

- Adam, Charlie, Matt, Jacob, Ali, Matthijs, Felipe, Marin, Brandon, and Ray from the raywenderlich.com Tutorial Team