

Fully updated
for iOS 6!

iOS 5 by tutorials

SECOND EDITION

By the raywenderlich.com Tutorial Team

Steve Baranski, Adam Burkepile, Jacob Gundersen, Matthijs Hollemans,
Felipe Laso Marsetti, Cesare Rocchi, Marin Todorov, and Ray Wenderlich

iOS 5 By Tutorials

Second Edition

By the raywenderlich.com Tutorial Team

Steve Baranski, Adam Burkepile, Jacob Gundersen, Matthijs Hollemans,
Felipe Laso Marsetti, Cesare Rocchi, Marin Todorov, and Ray Wenderlich

Copyright © 2011, 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.....	6
Chapter 2: Beginning ARC.....	15
Chapter 3: Intermediate ARC	74
Chapter 4: Beginning Storyboards	116
Chapter 5: Intermediate Storyboards.....	174
Chapter 6: Beginning iCloud	218
Chapter 7: Intermediate iCloud.....	251
Chapter 8: Beginning OpenGL ES 2.0 with GLKit	354
Chapter 9: Intermediate OpenGL ES 2.0 with GLKit	387
Chapter 10: Beginning UIKit Customization	412
Chapter 11: Intermediate UIKit Customization	432
Chapter 12: Beginning Twitter.....	456
Chapter 13: Intermediate Twitter	469
Chapter 14: Beginning Newsstand	515
Chapter 15: Intermediate Newsstand.....	523
Chapter 16: Beginning UIPageViewController.....	540
Chapter 17: Intermediate UIPageViewController	555
Chapter 18: Beginning Turn Based Gaming	587
Chapter 19: Intermediate Turn Based Gaming	645
Chapter 20: Beginning Core Image.....	670
Chapter 21: Intermediate Core Image	692
Chapter 22: UIViewController Containment.....	720
Chapter 23: Working with JSON.....	736

Chapter 24: UIKit Particle Systems	750
Chapter 25: Using the iOS Dictionary.....	763
Chapter 26: New AddressBook APIs.....	776
Chapter 27: New Location APIs.....	793
Chapter 28: New Game Center APIs.....	810
Chapter 29: New Calendar APIs	821
Chapter 30: Using the New Linguistic Tagger API.....	848
Chapter 31: Conclusion	864

Dedications

To Kristine, Emily, and Sam.

Steve Baranski

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

- Adam Burkepile

To my boys, John and Eli, may I be as cool as you two.

- Jacob Gundersen

To all the coders out there who are making the world a better place through their software.

- Matthijs Hollemans

To my amazing mother Patricia, and my beautiful nephew Leonardo, I love you guys with all my heart. I'd also like to thank Ray for making me a part of his team and helping boost my career to the next level. Finally I'd like to dedicate this to Steve Jobs. God bless you and rest in your peaceful iCloud!

- Felipe Laso Marsetti

To my parents, my friends around the globe, and my inspirers. Also thanks to Ray for inviting me in.

-Marin Todorov

To the iOS Tutorial Team - together we have made something truly amazing!

-Cesare Rocchi and Ray Wenderlich

Chapter 1: Introduction

When the raywenderlich.com Tutorial Team first started looking into iOS 5, we were amazed by the wealth of new libraries, new APIs, and new features available. iOS 5 was one of the biggest upgrades to iOS yet, containing tons of cool new stuff you can start using in your apps – even in iOS 6 and beyond!

But as we were researching iOS 5, we realized there wasn't a lot of high quality sample code, tutorials, and documentation to help developers (such as ourselves!) get up to speed quickly on all of these new features.

So we decided to team up and solve that problem by writing this book. Our goal was to create the definitive guide to help intermediate and advanced iOS developers learn the new iOS 5 APIs in the quickest and easiest way – via tutorials!

The Tutorial Team takes pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun. And we don't want to just skim the surface of a subject – we want to really dig into it, so you can truly understand how it works and apply the knowledge directly in your own apps.

If you've enjoyed the tutorials we've written in the past at raywenderlich.com, you're in for a treat. The tutorials we've written in this book are some of our best yet – and this book contains detailed technical knowledge you simply won't be able to find anywhere else.

So if you're eager to learn what iOS 5 introduced for you, you're in the right place. Sit back, relax, and prepare for some fun and informative tutorials!

Introducing the second edition

It's been a year since we originally wrote *iOS 5 by Tutorials*, and we are happy to announce that you are holding a brand new second edition of the book, fully revised for iOS 6!

We went through each and every chapter of this book and updated the code so that everything works on iOS 6, uses the latest and greatest technologies and API calls where necessary, and uses Modern Objective-C syntax like auto-synthesis and literals. We also made a ton of other improvements and updates along the way!

And the best part, is even though this second edition took a huge amount of effort on our part, we are releasing this as a free update to our PDF customers! This is our way of saying thanks for purchasing the PDF – you are what makes all of the tutorials, books, and Starter Kits we make on raywenderlich.com possible!

This book has been designed to act as a good companion to its sister book, *iOS 6 by Tutorials*. This book focuses on all of the new APIs that came out in iOS 5 such as ARC, Storyboards, and iCloud. Similarly, *iOS 6 by Tutorials* focuses on all of the new APIs that came out in iOS 6 such as Auto Layout, Collection Views, and Passbook. If you have them both, you have it all – over 2,300 pages of up-to-date high quality tutorials!

Wondering what's changed in the second edition? Here's the full list!

- Updated all chapters to be compatible with iOS 6, including using new APIs where possible and adding notes about relevant updates for iOS 6.
- Updated all chapters to be compatible with the latest version of Xcode (4.5 at the time of this update).
- Updated all chapters to use Modern Objective-C syntax (auto-synthesis, literal syntax, etc.).
- Updated many chapters to use Storyboards instead of XIBs.
- Updated chapters to use most recent version of imported libraries, such as Cocos2D.
- Added additional clarifications and useful notes to many chapters.
- Streamlined many chapters by removing instructions that were not relevant to the main topic at hand (such as creating user interfaces), instead moving that to a starter project. The chapters should now be more focused on the main topic.

Note: This book requires the latest version of Xcode, which is 4.5 at the time of writing this book. If you are running an older version of Xcode, you should either update your version of Xcode (our suggestion), or use an older version of *iOS 5 by Tutorials* (links available on the private PDF forums).

Also, as mentioned above all chapters in this book have been updated to use Modern Objective-C syntax. If some of this looks unfamiliar to you and you are still "stuck in the old days", you should check out Chapter 2 in *iOS 6 by Tutorials*, "Modern Objective-C Syntax".

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 firm up their knowledge about technologies introduced in iOS 5.

If you're a complete beginner, you can follow along with this book as well, because the tutorials are always in a step-by-step process, but there may be some missing gaps in your knowledge. You might want to go through the *iOS Apprentice* series available on the raywenderlich.com store before you go through this book.

How to use this book

This book is huge! We wanted to provide you guys with detailed top quality content so you could really understand each topic, so included a ton of material for you.

You're welcome to read through this book chapter by chapter of course (the chapters have been arranged to have a good reading order), but we realize that as a developer your time is limited so you might not have time to go through the entire book.

If that is the case for you, we recommend you just pick and choose the subjects you are interested in or need for your projects, and jump directly to those chapters. Most tutorials are self-contained so there will not be a problem if you go through them in a different order.

Not sure which ones are the most important? Here's our recommended "core reading list": Beginning ARC, Beginning Storyboards, Beginning iCloud, and Beginning UIKit Customization. That covers the "big 4" topics of iOS 5, and from there you can dig into other topics that are particularly interesting to you.

Book overview

iOS 5 introduced a ton 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:

Beginning and Intermediate ARC

ARC stands for Automatic Reference Counting, and is a fancy way of saying, "remember all that memory management code you used to have to write? Well, you don't need that anymore!"

This is a huge new feature, and will save you a lot of memory management headaches and make your code easier to write and read. In this book, not only will you learn about how to use ARC, but also how it works under the hood, how to port your old projects to ARC, how to use third party libraries that haven't been

converted to ARC, how to handle subtle problems and issues with ARC, and much more!

Beginning and Intermediate Storyboards

In the old days, you would generally have one XIB per view controller. This worked well, except it wasn't very easy to visualize the flow of your app without resorting to third party diagramming tools.

iOS 5 introduced a new way to design view controllers – the Storyboard editor! You can now design the visual look of multiple view controllers in one place, and easily visualize the transitions between view controllers.

What's more, Storyboards can save you a ton of time, because they introduce cool new features such as creating table view cells directly within the editor. In this book, you'll dive deep into Storyboards and learn how you can use all the major new features in your apps.

Beginning and Intermediate iCloud

Before iOS 5, if you wanted to share data between devices, you would have to write your own web service, or integrate with a third party API like Dropbox. This was usually quite error prone and a lot of work!

Now with iCloud, there's built-in functionality provided by Apple that lets you save your app's data in the cloud, and easily synchronize it between your apps on different devices. Customers are going to start expecting your apps to have this feature, so in this book you'll get to dive in and get some hands-on experience working with this cool new technology!

Beginning and Intermediate OpenGL ES 2.0 with GLKit

If you've been wanting to get into OpenGL ES 2.0 programming but have been a little bit scared by the complexity, the new GLKit framework has made things a lot easier for you to get started. Experienced OpenGL developers will love GLKit too, because you can use it to remove a ton of boilerplate code from your apps and make transitioning from OpenGL ES 1.0 to OpenGL ES 2.0 much easier.

In this book, you'll dive into the new `GLKView`, `GLKViewController`, `GLKBaseEffect`, `GLKTextureLoader`, and `GLKMath` APIs – in a manner easy to follow, whether you are a complete beginner or an advanced developer!

Beginning and Intermediate UIKit Customization

To have a successful app on the App Store these days, your app has to look good. Almost everyone wants to customize the default look of the UIKit controls, but in the old days you had to resort to some strange workarounds to make this work.

With iOS 5, this has become a lot easier, so in this book you're going to dig into some practical examples of customizing just about every control you might want in UIKit!

Beginning and Intermediate Twitter Integration

iOS 5 made it a lot easier to use Twitter within your app, with built-in Twitter integration. In these chapters, you'll learn how to use these APIs from your apps to send Tweets easier than ever! Plus, you will learn how to use the Twitter API directly so you can access the full power of the API within your app!

Beginning and Intermediate Newsstand

iOS 5 introduces a special folder on your home screen just for periodicals. The folder is special because your app's icon can update to reflect your latest content – just like a real magazine or newspaper!

In these chapters, you'll learn about the benefits of distributing your content via Newsstand, learn how to modify your app in order to appear in Newsstand, and learn how to deliver dynamic content to your app.

Beginning and Intermediate Core Image

Core Image is a powerful new framework that lets you easily apply filters to images, such as modifying the vibrance, hue, or exposure. And the best part is it uses the GPU to run the filters, so they are extremely fast!

In these chapters, you'll learn how to use the Core Image framework to easily apply cool filters to your images. You'll also learn how to use Core Image to compose filters, mask images, and even perform face detection!

Beginning and Intermediate Turn-Based Gaming

One of the coolest new features of Game Center in iOS 5 is the new turn-based gaming feature. This makes it extremely easy to create multiplayer turn-based games where you take a turn, asynchronously wait for a friend to take his/her turn, and get a notification when it's your turn again.

In these chapters, you'll dive into a fun example of how you can integrate turn-based gaming into a simple game, and then dig further to see how you can create your own custom UI!

Beginning and Intermediate UIPageViewController

Ever since Apple introduced iBooks, developers have been yearning for a way to get that cool page curl animation in their apps. In the past, some people have developed libraries to simulate this effect, but now it's baked in to iOS 5 and above!

In these chapters, you'll learn how this new view controller works, and how to use it to make a simple photo album app.

Other New iOS 5 APIs

The other chapters cover the biggest new features in iOS 5, but there are a ton of other handy features you should know about.

So we added a ton of bonus chapters to the book as well! In these bonus chapters, you'll learn about almost every new iOS 5 API not already covered – things like View Controller Containment, JSON parsing, Address Book and Location APIs, and much more!

Even though these APIs are smaller than the topics from the other chapters, we've worked hard to present them in real life projects, which you can directly benefit from – and maybe even extend into your own apps.

So if you want to fully round out your iOS 5 knowledge, be sure to check out these gems!

Book source code and forums

When you downloaded this book, you should have received a zip file that contains both this PDF, along with a bunch of folders containing source code and resources for the chapters.

Some of the chapters in this book have starter projects or required resources, so you'll definitely want to keep these on hand as you go through the chapters!

We also have public forums set up where you can ask questions about the book, or submit any errata you may find. You can find the forum here:

- <http://www.raywenderlich.com/forums/viewforum.php?f=16>

Also, since you bought the PDF, you also have access to a private forum where you can always find the latest download link for the PDF, and download any updates in the future. You likely already have access to this forum and should see it in the list of forums above.

If you do not already have access to this forum, you can sign up here:

- <http://www.raywenderlich.com/store/ios-5-by-tutorials/forum-signup>

And if you have any troubles signing up, you can just email me directly at ray@raywenderlich.com.

We hope to see you on the forums! ☺

Acknowledgements

We would like to thank several people for their assistance 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!

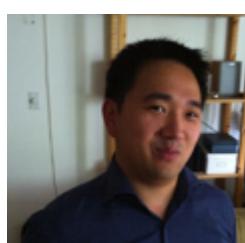
- **Everyone at Apple:** For developing an amazing set of APIs, constantly inspiring us to improve our apps and skills, and making it possible for many developers to have their dream jobs as app developers!
- **Andrea Coulter:** For doing all the painstaking formatting and layout of the first version of this book!
- **Vicki Wenderlich:** For designing the book cover, formatting and layout of the second edition of this book, and for much of the lovely artwork and illustrations in the book.
- **Mike Daley:** For helping out in our investigations of GLKit.
- **Adam Burkepile, Richard Casey, Adam Eberbach, Brandon Lassiter, Marcio Valenzuela, and Nick Waynik:** For being excellent forum moderators and generously donating their time and experience to the iOS community.
- **Steve Jobs:** You were an inspiration to all of us, and without you this book (or our careers!) wouldn't be possible. We are immensely grateful.
- 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! ☺



Steve Baranski is the founder of Komorka Technology, a creator of original & commissioned mobile apps based in Oklahoma City. When not making apps or spending time with his family, he can usually be found on a bicycle of some sort.



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.



Jake Gundersen 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>.



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](#) or on his blog at <http://iFe.li>.



Cesare Rocchi is a UX designer and developer. He runs Studio Magnolia (<http://www.studiomagnolia.com>), an interactive studio that creates compelling web and mobile applications. You can find him on Twitter as [@_funkyboy](#) and LinkedIn as cesarerocchi. When off duty he enjoys snowboarding and beach tennis.



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>



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 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: Beginning ARC

By Matthijs Hollemans

The most disruptive change in iOS 5 is the addition of Automatic Reference Counting, or ARC for short. ARC is a feature of the new LLVM 3.0 compiler and it completely does away with the manual memory management that all iOS developers love to hate.

Using ARC in your own projects is extremely simple. You keep programming as usual, except that you no longer call `retain`, `release` and `autorelease`. That's basically all there is to it.

With Automatic Reference Counting enabled, the compiler will automatically insert `retain`, `release` and `autorelease` in the correct places in your program. You no longer have to worry about any of this, because the compiler does it for you. I call that freaking awesome. In fact, using ARC is so simple that you can stop reading this tutorial now. ;-)

But if you're still skeptical about ARC – maybe you don't trust that it will always do the right thing, or you think that it somehow will be slower than doing memory management by yourself – then read on. The rest of this chapter will dispel those myths and show you how to deal with some of the less intuitive consequences of enabling ARC in your projects.

In addition, you'll get hands-on experience with converting an app that doesn't use ARC at all to using ARC. You can use these same techniques to move your existing iOS projects over to ARC, saving yourself tons of memory headaches!

Memory Management is... Yuck

You're probably already familiar with manual memory management, which basically works like this:

- If you want to keep an object around you need to retain it, unless it already was retained for you.

- If you want to stop using an object you need to release it, unless it was already released for you (with autorelease).

As a beginner you may have had a hard time wrapping your head around the concept but after a while it became second nature and now you'll always properly balance your retains with your releases. Except when you forget.

The principles of manual memory management aren't hard but it's very easy to make a mistake. And these small mistakes can have dire consequences. Either your app will crash at some point because you've released an object too often and your variables are pointing at data that is no longer valid, or you'll run out of memory because you don't release objects enough and they stick around forever.

The static analyzer from Xcode is a great help in finding these kinds of problems but ARC goes a step further. It avoids memory management problems completely by automatically inserting the proper retains and releases for you!

It is important to realize that ARC is a feature of the Objective-C compiler and therefore all the ARC stuff happens when you build your app. ARC is not a runtime feature (except for one small part, the weak pointer system), nor is it *garbage collection* that you may know from other languages.

All that ARC does is insert retains and releases into your code when it compiles it, exactly where you would have – or at least should have – put them yourself. That makes ARC just as fast as manually managed code, and sometimes even a bit faster because it can perform certain optimizations under the hood.

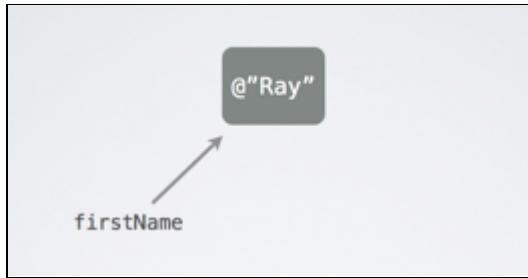
Pointers Keep Objects Alive

The new rules you have to learn for ARC are quite simple. With manual memory management you needed to retain an object to keep it alive. That is no longer necessary; all you have to do is make a pointer to the object. As long as there is a variable pointing to an object, that object stays in memory. When the pointer gets a new value or ceases to exist, the associated object is released. This is true for all variables: instance variables, synthesized properties, and even local variables.

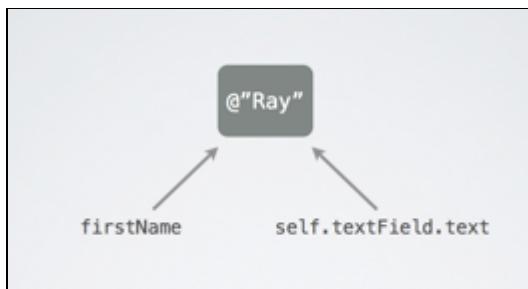
It makes sense to think of this in terms of ownership. When you do the following,

```
NSString *firstName = self.textField.text;
```

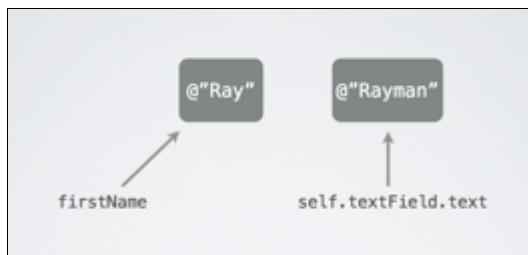
the `firstName` variable becomes a pointer to the `NSString` object that holds the contents of text field. That `firstName` variable is now the owner of that string object.



An object can have more than one owner. Until the user changes the contents of the `UITextField`, its `text` property is also an owner of the string object. There are two pointers keeping that same string object alive:



Moments later the user will type something new into the text field and its `text` property now points at a new string object. But the original string object still has an owner – the `firstName` variable – and therefore stays in memory.



Only when `firstName` gets a new value too, or goes out of scope – because it's a local variable and the method ends, or because it's an instance variable and the object it belongs to is deallocated – does the ownership expire. The string object no longer has any owners, its retain count drops to 0 and the object is deallocated.

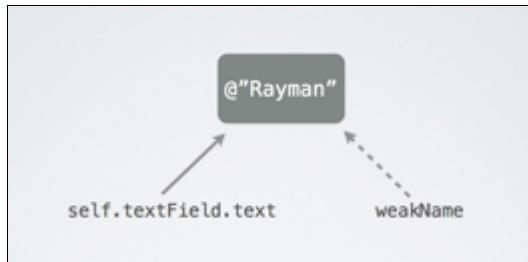


We call pointers such as `firstName` and `textField.text` "strong" because they keep objects alive. By default all instance variables and local variables are strong pointers.

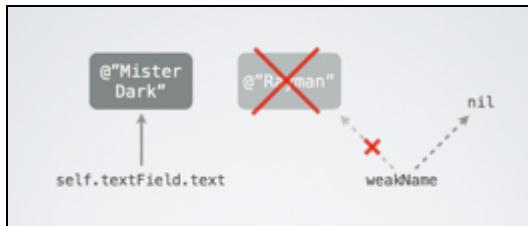
There is also a “weak” pointer. Variables that are weak can still point to objects but they do not become owners:

```
__weak NSString *weakName = self.textField.text;
```

Note that the `__weak` symbol is spelled with two leading underscore characters.



The `weakName` variable points at the same string object that the `textField.text` property points to, but it is not an owner. If the text field contents change, then the string object no longer has any owners and is deallocated:

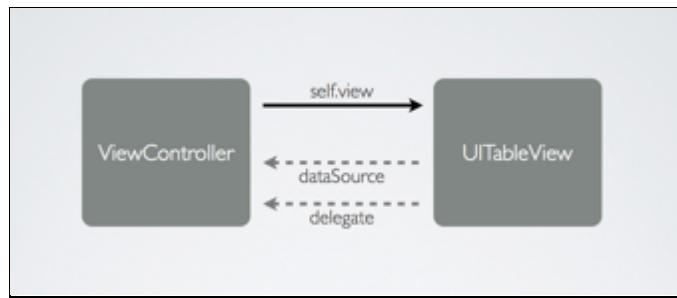


When this happens, the value of `weakName` automatically becomes `nil`. It is said to be a “zeroing” weak pointer.

Note that this is extremely convenient because it prevents weak pointers from pointing to deallocated memory. This sort of thing used to cause a lot of bugs – you may have heard of the term “dangling pointers” or “zombies” – but thanks to these zeroing weak pointers, that is no longer an issue! The variable either points to a valid object, or is `nil`, but it never references memory that has been freed.

You probably won’t use weak pointers very much. They are mostly useful when two objects have a parent-child relationship. The parent will have a strong pointer to the child – and therefore “owns” the child – but in order to prevent ownership cycles, the child only has a weak pointer back to the parent.

A typical example of this is the delegate pattern. Your view controller may own a `UITableView` through a strong pointer. The table view’s data source and delegate pointers point back at the view controller, but are weak. You’ll read more about this later in the chapter.



Note that the following isn't very useful:

```
__weak NSString *str = [[NSString alloc] initWithFormat:...];
 NSLog(@"%@", str); // will output "(null)"
```

There is no owner for the `NSString` object (because `str` is weak) and the object will be deallocated immediately after it is created. Xcode gives a warning when you attempt to do this because it's probably not what you intended to happen ("Warning: assigning retained object to weak variable; object will be released after assignment").

You can use the `__strong` keyword to signify that a variable is a strong pointer:

```
__strong NSString *firstName = self.textField.text;
```

But because variables are strong by default this is a bit superfluous.

Properties can also be strong and weak. The notation for properties is:

```
@property (nonatomic, strong) NSString *firstName;
@property (nonatomic, weak) id <MyDelegate> delegate;
```

ARC is great and will really remove a lot of clutter from your code. You no longer have to think about when to retain and when to release, just about how your objects relate to each other. The question that you'll be asking yourself is: who owns what?

For example, it was impossible to write code like this before:

```
id obj = [array objectAtIndex:0];
[array removeObjectAtIndex:0];
NSLog(@"%@", obj);
```

Under manual memory management, removing the object from the array would invalidate the contents of the `obj` variable. The object got deallocated as soon as it no longer was part of the array. Printing the object with `NSLog()` would likely crash your app. To work around this, you first had to do `[obj retain]` before removing the object from the array, and not forget to release it again later.

On ARC the above code works as intended. Because you put the object into the `obj` variable, which is a strong pointer, the array is no longer the only owner of the object. Even after you remove the object from the array, the object is still alive because `obj` keeps pointing at it.

I have seen the future and it is ARC

Automatic Reference Counting also has a few limitations. For starters, ARC only works on Objective-C objects. If your app uses Core Foundation or `malloc()` and `free()`, then you're still responsible for doing the memory management there. You'll see examples of this later in the tutorial. In addition, certain language rules have been made stricter in order to make sure ARC can always do its job properly. These are only small sacrifices; you gain a lot more than you give up!

Just because ARC takes care of doing `retain` and `release` for you in the proper places doesn't mean you can completely forget about memory management altogether. Because strong pointers keep objects alive, there are still situations where you will need to set these pointers to `nil` by hand, or your app might run out of available memory. If you keep holding on to all the objects you've ever created, then ARC will never be able to release them. Therefore, whenever you create a new object, you still need to think about who owns it and how long the object should stay in existence.

There is no doubt about it: ARC is the future of Objective-C. Apple encourages developers to turn their backs on manual memory management and to start writing their new apps using ARC. It makes for simpler source code and more robust apps. With ARC, memory-related crashes are a thing of the past. Believe it or not, ARC even combines well with C++. With a few restrictions you can also use ARC on iOS 4, which should only help to speed up the adoption.

As the developer community is entering a transitioning period from manual to automatic memory management you'll often come across code that isn't compatible with ARC yet, whether it's your own code or third-party libraries. Fortunately you can combine ARC with non-ARC code in the same project and this chapter will show you several ways how.

A smart developer tries to automate as much of his job as possible, and that's exactly what ARC offers: automation of menial programming work that you had to do by hand previously. To the author, switching is a no-brainer.

The Artists App

To illustrate how to use Automatic Reference Counting in practice, you are going to convert a simple app from manual memory management to ARC. The app, *Artists*, consists of a single screen with a table view and a search bar. When you type something into the search bar, the app employs the MusicBrainz API to search for musicians with matching names.

The app looks like this on the iPhone 5 simulator:



In their own words, MusicBrainz is “an open music encyclopedia that collects, and makes available to the public, music metadata”. They have a free XML web service that you can use from your own apps. To learn more about MusicBrainz, check out their website at <http://musicbrainz.org>.

You can find the starter code for the Artists app in this chapter’s folder. The Xcode project contains the following source files:

- **AppDelegate.h/.m**: The application delegate. Nothing special here, every app has one. It loads the view controller and puts it into the window.
- **MainViewController.h/.m/.xib**: The view controller for the app. It has a table view and a search bar, and does most of the work.
- **SoundEffect.h/.m**: A simple class for playing sound effects. The app will make a little beep when the MusicBrainz search is completed.
- **main.m**: The entry point for the app.

In addition, the app makes use of two third-party libraries. Your apps probably use a few external components of their own and it’s good to learn how to make these libraries play nice with ARC.

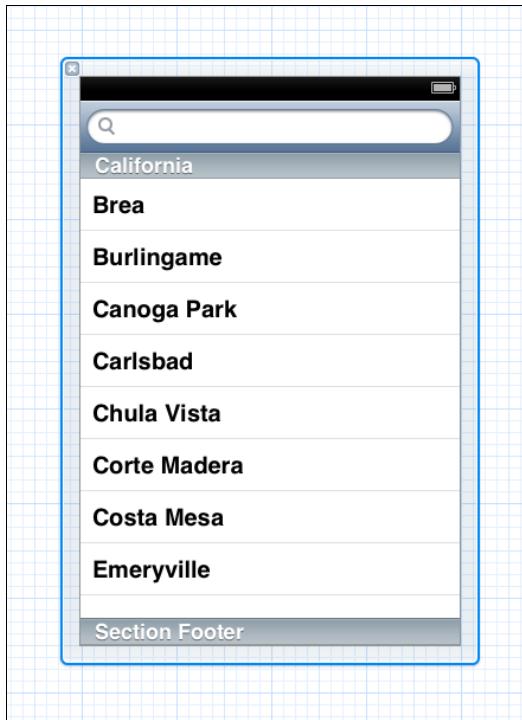
- **AFHTTPRequestOperation.h/.m**: Part of the AFNetworking library that makes it easy to perform requests to web services. The Artists project does not include the full library because you only need this one class. You can find the complete package at: <https://github.com/AFNetworking/AFNetworking/>
- **SVProgressHUD.h/.m/.bundle**: A progress indicator or “HUD” that will pop up on the screen while the search is taking place. You may not have seen “.bundle” files before. This is a special type of folder that contains the image files that are

used by SVProgressHUD. To view these images, right-click the .bundle file in Finder and choose the Show Package Contents menu option. For more info about this component, see: <https://github.com/samvermette/SVProgressHUD>

Note: Since the first edition of this book was published, both AFNetworking and SVProgressHUD were converted to use ARC, so if you insert the latest versions of these components into your ARC-enabled apps they should work without problems. However, for the purposes of this chapter you are intentionally using an older version so that you can learn how to convert other people's code to ARC by yourself. Over time, this will become less of a necessity as more and more libraries are switched to ARC by their maintainers.

A quick tour of the app

Let's quickly go through the code for the view controller so you have a decent idea of how the app works. `MainViewController` is a subclass of `UIViewController`. Its nib file contains a `UITableView` object and a `UISearchBar` object:



The table view displays the contents of the `searchResults` array. Initially this pointer is `nil`. When the user performs a search, the code fills up the array with the response from the MusicBrainz server. If there were no search results, the array is empty (but not `nil`) and the table says: "(Nothing found)". This all takes place in the usual `UITableView-DataSource` methods: `numberOfRowsInSection` and `cellForRowAtIndexPath`.

The actual search is initiated from the `searchBarSearchButtonClicked` method, which is part of the `UISearchBarDelegate` protocol.

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)theSearchBar
{
    [SVProgressHUD showInView:self.view status:nil
        networkIndicator:YES posY:-1
        maskType:SVProgressHUDMaskTypeGradient];
```

First, this method creates a new HUD and shows it on top of the table view and search bar, blocking any user input until the network request is done:



Then it creates the URL for the HTTP request, using the MusicBrainz API to search for artists:

```
NSString * urlString = [NSString stringWithFormat:
    @"http://musicbrainz.org/ws/2/artist?query=artist:%@&limit=20"
    , [self escape:self.searchBar.text]];

NSMutableURLRequest * request = [NSMutableURLRequest
    requestWithURL:[NSURL URLWithString:urlString]];
```

Note that the search text is URL-encoded using the `escape:` method to ensure that you're making a valid URL. Spaces and other special characters are turned into escape sequences such as `%20`.

```
NSDictionary * headers = [NSDictionary dictionaryWithObject:
    [self userAgent] forKey:@"User-Agent"];
```

```
[request setAllHTTPHeaderFields:headers];
```

This adds a custom User-Agent header to the HTTP request. The MusicBrainz API requires this. All requests should “have a proper User-Agent header that identifies the application and the version of the application making the request.” It’s always a good idea to play nice with the APIs you’re using, so the app constructs a User-Agent header that looks like:

```
com.yourcompany.Artists/1.0 (unknown, iPhone OS 5.0, iPhone Simulator,  
Scale/1.000000)
```

This formula was taken from another part of the AFNetworking library and placed into the `userAgent` method in the view controller.

Note: The MusicBrainz API has a few other restrictions too. Client applications must not make more than one web service call per second or they risk getting their IP address blocked. That won’t be too big of an issue for this app – it’s unlikely that a user will be doing that many searches – so the code does not take any particular precautions for this.

Given the newly constructed `NSMutableURLRequest` object, `AFHTTPRequestOperation` can perform the request:

```
AFHTTPRequestOperation *operation = [AFHTTPRequestOperation  
operationWithRequest:request  
completion:^(NSURLRequest *request,  
           NSHTTPURLResponse *response,  
           NSData *data, NSError *error)  
{  
    ...  
};  
  
[_queue addOperation:operation];
```

`AFHTTPRequestOperation` is a subclass of `NSOperation`, which means you can add it to an `NSOperationQueue` (in the `_queue` instance variable) and it will be handled asynchronously. Because the HUD blocks the screen, the app ignores any user input while the request is taking place.

You give `AFHTTPRequestOperation` a block that it invokes when the request completes. Inside the block the code first checks whether the request was successful (HTTP status code 200) or not. For this app we’re not particularly interested in why a request failed; if it does we simply tell the HUD to dismiss with a special “error” animation.

Note that the completion block is not necessarily executed on the main thread and therefore you need to wrap the call to SVProgressHUD in `dispatch_async()`.

```

if (response.statusCode == 200 && data != nil)
{
    . . .
}
else // something went wrong
{
    dispatch_async(dispatch_get_main_queue(), ^
    {
        [SVProgressHUD dismissWithError:@"Error"];
    });
}

```

Now for the interesting part. If the request succeeds, we allocate the `searchResults` array and parse the response. The response is XML so the app uses `NSXMLParser` to do the job.

```

self.searchResults = [NSMutableArray arrayWithCapacity:10];

NSXMLParser *parser = [[NSXMLParser alloc]
                      initWithData:data];
[parser setDelegate:self];
[parser parse];
[parser release];

[self.searchResults sortUsingSelector:
 @selector(localizedStandardCompare:)];

```

You can look up the logic for the XML parsing in the `NSXMLParserDelegate` methods, but essentially it just looks for elements named "sort-name". These contain the names of the artists. Those names are added as `NSString` objects to the `searchResults` array.

When the XML parser is done, we sort the results alphabetically, and then update the screen on the main thread:

```

dispatch_async(dispatch_get_main_queue(), ^
{
    [self.soundEffect play];
    [self.tableView reloadData];
    [SVProgressHUD dismiss];
});

```

That's it for how the app works. It's written using manual memory management and doesn't use any iOS 5 specific features. Now let's convert it to ARC.

Automatic Conversion

You are going to convert the Artists app to ARC. Basically this means you'll just get rid of all the calls to `retain`, `release`, and `autorelease`, but you'll also run into a few situations that require special attention.

There are three things you can do to make your app ARC-compatible:

1. Xcode has an automatic conversion tool that can migrate your source files.
2. You can convert the files by hand.
3. You can disable ARC for source files that you do not want to convert. This is useful for third-party libraries that you don't feel like messing with.

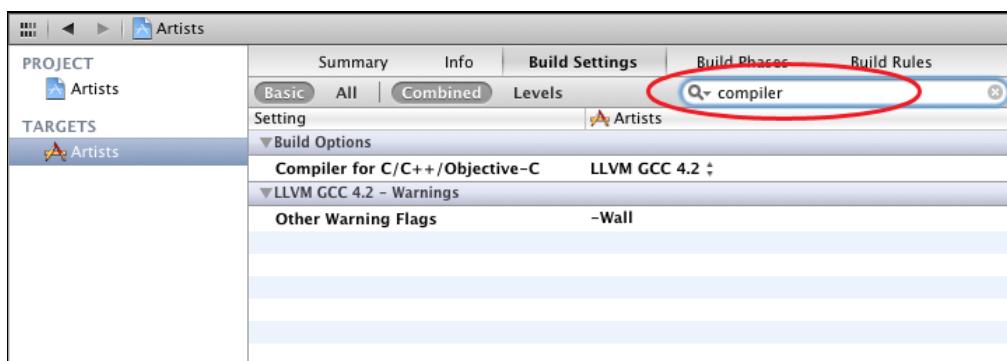
You will use all of these options on the Artists app, just to get an idea of how it all works.

In this section, you are going to convert the source files with Xcode's automated conversion tool, except for `MainViewController` and `AFHTTPRequestOperation`.

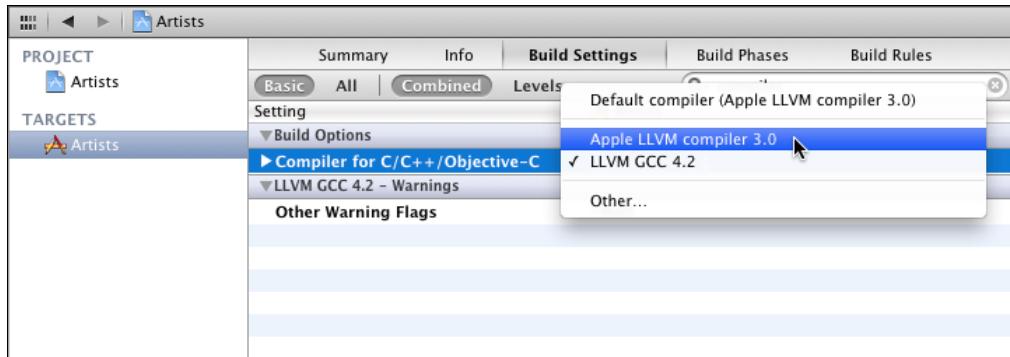
Note: Before you continue, you should make a copy of the project, as the ARC conversion tool will overwrite the original files. Xcode does offer to make a snapshot of the source files but just as a precaution it is wise to make a backup anyway.

ARC is a feature of the new LLVM 3.0 and 4.0 compilers. Your existing projects most likely use the older GCC 4.2 or LLVM-GCC compilers, so it's a good idea to switch the project to the new compiler first and see if it compiles cleanly in non-ARC mode.

Go to the **Project Settings** screen, select the **Artists target** and under **Build Settings** type "compiler" into the search box. This will filter the list to bring up just the compiler options:

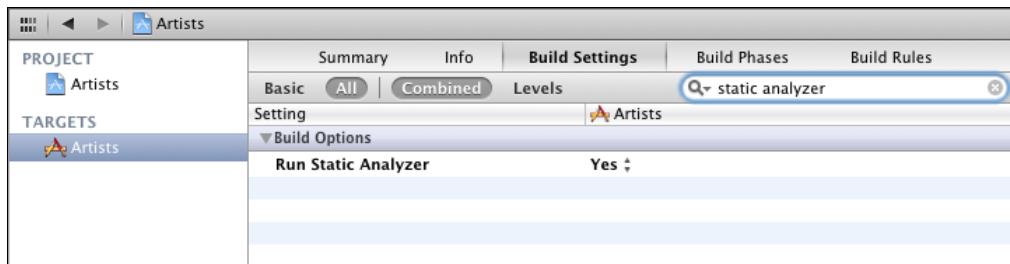


Make sure the **Compiler for C/C++/Objective-C** option says “Apple LLVM compiler” version 3.0 or higher.



Under the **Warnings** header, also set the **Other Warning Flags** option to **-Wall**. The compiler will now check for all possible situations that can cause problems. By default many of these warnings are turned off but it is useful to always have all warnings on and to treat them as fatal errors. In other words, if the compiler gives a warning it is smart to first fix the problem before continuing. Whether that is something you may want to do on your own projects is up to you, but during the conversion to ARC it is recommended that you take a good look at any issues the compiler may complain about.

For the exact same reason, also enable the **Run Static Analyzer** option under the **Build Options** header:



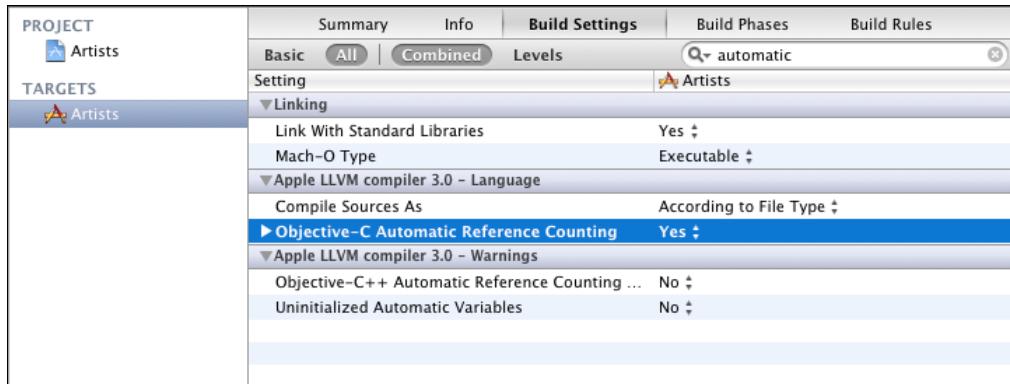
Xcode will now run the analyzer every time you build the app. That makes the builds a bit slower but for an app of this size that's barely noticeable.

Let's build the app to see if it gives any problems with the new compiler. First do a clean using the **Product\Clean** menu option (or Shift-Cmd-K). Then press **Cmd-B** to build the app. Xcode should give no errors or warnings, so that's cool. If you're converting your own app to ARC and you get any warning messages at this point, then now is the time to fix them.

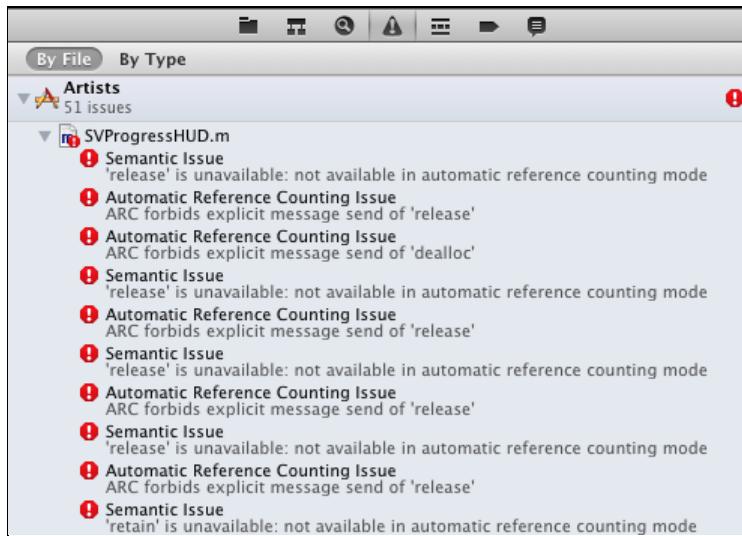
Trying out ARC

Just for the fun of it, let's switch the compiler to ARC mode and make it build the app again. You're going to get a ton of error messages but it's instructive to see what exactly these are.

Still in the Build Settings screen, switch to “All” to see all the available settings (instead of Basic, which only shows the most-often used settings). Search for “automatic” and set the **Objective-C Automatic Reference Counting** option to **Yes**. This is a project-wide flag that tells Xcode that you wish to compile all of the source files in your project using the ARC compiler.



Build the app again. Whoops, you will get a ton of errors:

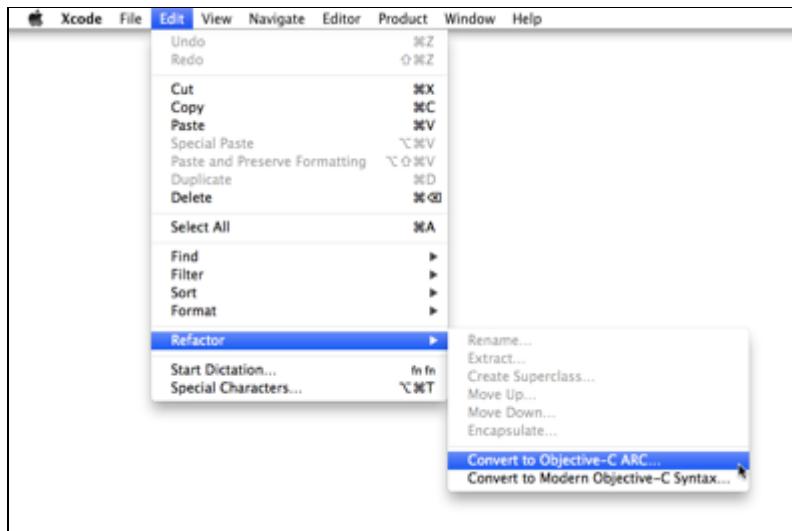


Clearly you have some migrating to do! Most of these errors are pretty obvious; they say you can no longer use `retain`, `release` and `autorelease`. You could fix all of these errors by hand but it's much easier to employ the automatic conversion tool.

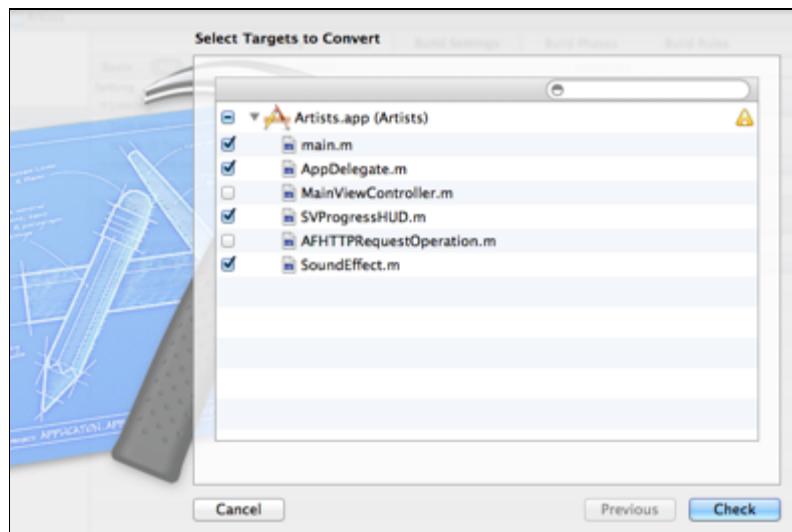
The tool will compile the app in ARC mode and rewrites the source code for every error it encounters, until the whole thing compiles cleanly.

Putting the conversion tool into action

From Xcode's menu, choose **Edit\Refactor\Convert to Objective-C ARC**.



A new window appears that lets you select which parts of the app you want to convert:



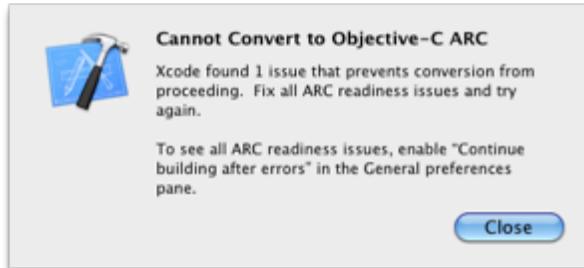
For the purposes of this tutorial, you don't want to do the whole app, so select only the following files:

- main.m
- AppDelegate.m
- SVProgressHUD.m
- SoundEffect.m

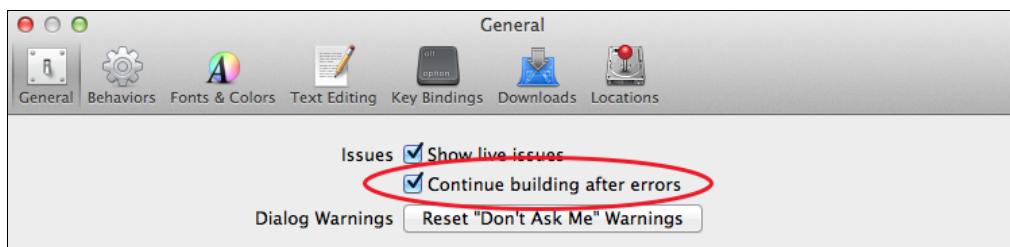
Note: The dialog shows a little warning icon indicating that the project already uses ARC. That's because earlier you've enabled the Objective-C Automatic Reference Counting option in the Build Settings and now the conversion tool

thinks this is already an ARC project. You can ignore the warning; it won't interfere with the conversion.

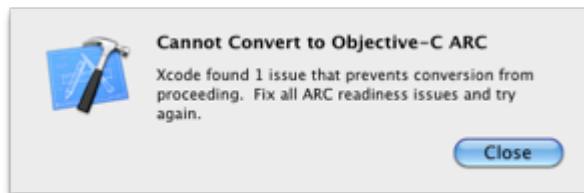
Press the Check button to begin. The tool first checks whether your code is in a good enough state to be converted to ARC. You did manage to build the app successfully with the new LLVM compiler earlier, but apparently that left something to be desired. Xcode gives the following error message:



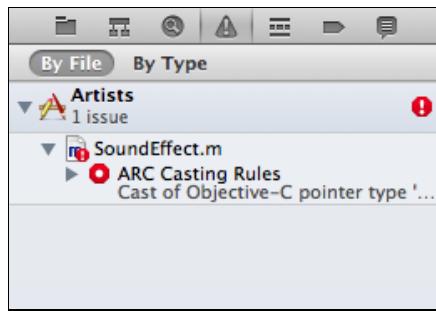
It complains about "ARC readiness issues" and that you should enable the "Continue building after errors" option. Open the **Xcode Preferences** window (from the menubar, under Xcode) and go to the **General** tab. Enable the option **Continue building after errors**:



Let's try again. Choose **Edit\Refactor\Convert to Objective-C ARC** and check the source files except for MainViewController.m and AFHTTPRequestOperation.m. Press Check to begin.



No luck, again you get an error message. The difference with before is that this time at least the compiler was able to identify all the issues that you need to fix before you can do the conversion. Fortunately, there is only one:



(You may have more errors in this list than are displayed here. Sometimes the conversion tool also complains about things that are not really “ARC readiness” issues.)

Get ready for ARC

The full description of the one ARC readiness issue that you found is:

Cast of Objective-C pointer type 'NSURL *' to C pointer type 'CFURLRef' (aka 'const struct __CFURL *') requires a bridged cast

This is what it looks like in the source editor:

The screenshot shows the Xcode source editor for the file 'SoundEffect.m'. The code is as follows:

```

17 #import "SoundEffect.h"
18
19 @implementation SoundEffect
20
21 - (id)initWithSoundNamed:(NSString *)filename
22 {
23     if ((self = [super init]))
24     {
25         NSURL *fileURL = [[NSBundle mainBundle] URLForResource:filename withExtension:nil];
26         if (fileURL != nil)
27         {
28             SystemSoundID theSoundID;
29             OSStatus error = AudioServicesCreateSystemSoundID((CFURLRef)fileURL, &theSoundID);
30             if (error == kAudioServicesNoError) // Cast of Objective-C pointer type 'NSURL *' to C pointer type 'CFURLRef' (aka 'const struct __CFURL *')
31                 _soundID = theSoundID;
32         }
33     }
34     return self;
35 }
36

```

A red rectangular highlight covers the line of code where the error occurred: `OSStatus error = AudioServicesCreateSystemSoundID((CFURLRef)fileURL, &theSoundID);`. A red circle with a white exclamation mark is placed to the left of the line, and a tooltip above it says 'Cast of Objective-C pointer type 'NSURL *' to C pointer type 'CFURLRef' (aka 'const struct __CFURL *') requires a bridged cast'.

We will go into more detail about this later, but the source code attempts to cast an `NSURL` object to a `CFURLRef` object. The `AudioServicesCreateSystemSoundID()` function takes a `CFURLRef` that describes where the sound file is located, but we are giving it an `NSURL` object instead. `CFURLRef` and `NSURL` are “toll-free bridged”, which makes it possible to use an `NSURL` object in place of a `CFURLRef` and vice versa.

Often the C-based APIs from iOS use Core Foundation objects (that’s what the CF stands for) while the Objective-C based APIs use “true” objects that extend the `NSObject` class. Sometimes you need to convert between the two and that is what the toll-free bridging technique allows for.

However, when you use ARC the compiler needs to know what it should do with those toll-free bridged objects. If you use an `NSURL` in place of a `CFURLRef`, then who is responsible for releasing that memory at the end of the day? To solve this conundrum, a set of new keywords was introduced for doing so-called “bridged

casts": `__bridge`, `__bridge_transfer` and `__bridge_retained`. You'll learn about the differences between them later in the chapter.

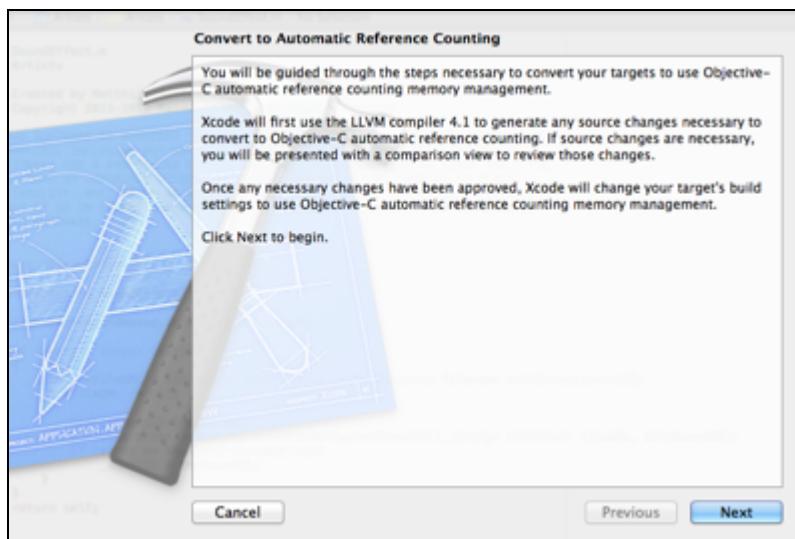
For now, change the source code to the following:

```
OSStatus error = AudioServicesCreateSystemSoundID(
    (__bridge CFURLRef) fileURL, &theSoundID);
```

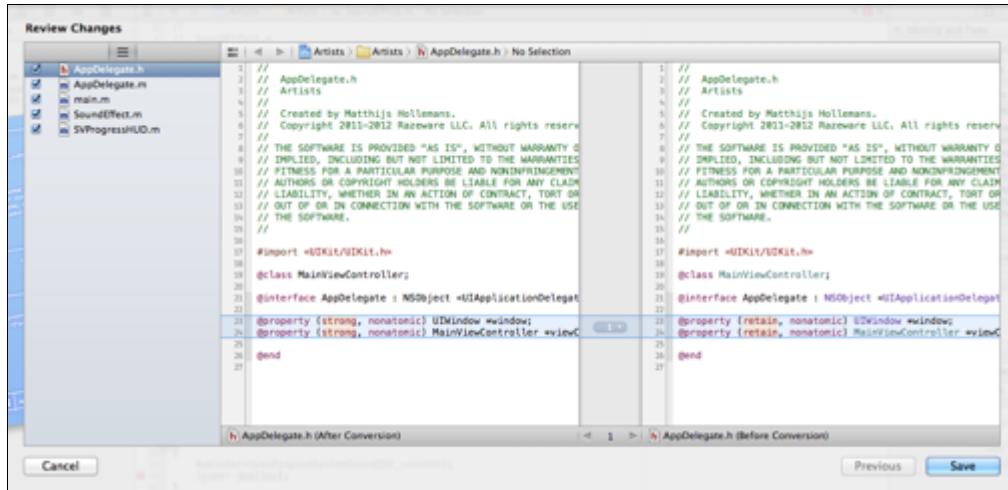
You added the `__bridge` keyword inside the cast to `CFURLRef`. (Again, that is two underscores.)

The pre-check may have given you errors other than just this one. You can safely ignore those; the above change in `SoundEffect.m` is the only one you need to make. The conversion tool just seems a little uncertain in what it considers an "ARC readiness issue" from time to time.

Let's run the conversion tool once more - **Edit\Refactor\Convert to Objective-C ARC**. This time the pre-check runs without problems and you're presented with the following screen:



Click **Next** to continue. After a few seconds, Xcode will show a preview of all the files that it will change and which changes it will make inside each file. The left-hand pane shows the changed files while the right-hand pane shows the originals.



It's always a good idea to step through these files to make sure Xcode doesn't mess up anything. Let's go through the changes that the conversion tool is proposing to make.

AppDelegate.h

```

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) MainViewController
    *viewController;

```

The app delegate has two properties, one for the window and one for the main view controller. This particular project does not use a MainWindow.xib file, so these two objects are created by the AppDelegate itself in `application:didFinishLaunchingWithOptions:` and stored into properties in order to simplify memory management.

These property declarations will change from this,

```
@property (retain, nonatomic)
```

to this:

```
@property (strong, nonatomic)
```

The `strong` keyword means what you think it does. It tells ARC that the synthesized instance variable that backs this property holds a strong reference to the object in question.

In other words, the `window` property contains a pointer to a `UIWindow` object and also acts as the owner of that `UIWindow` object. As long as the `window` property keeps its value, the `UIWindow` object stays alive. The same thing goes for the `viewController` property and the `MainViewController` object.

AppDelegate.m

In AppDelegate.m the lines that create the window and view controller objects have changed and the dealloc method is removed completely:

```

diff --git a/AppDelegate.m b/AppDelegate.m
--- a/AppDelegate.m
+++ b/AppDelegate.m
@@ -17,10 +17,10 @@
 // THIS SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF
 // ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 // MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
-// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
-// DAMAGES OR OTHER LIABILITY, WHETHER IN AN 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.
+// THIS SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF
+// ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
+// MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
+// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
+// DAMAGES OR OTHER LIABILITY, WHETHER IN AN 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.

 // Import "AppDelegate.h"
 #import "AppDelegate.h"
 #import "MainViewController.h"

 @implementation AppDelegate

@@ -22,10 +22,10 @@
 @synthesize window = _window;
 @synthesize viewController = _viewController;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
    self.viewController = [[MainViewController alloc] initWithNibName:@"MainViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
+}

- (void)dealloc
{
    [_window release];
    [_viewController release];
    [super dealloc];
}

@end

```

Spot the differences between this,

```
self.window = [[[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]] autorelease];
```

and this:

```
self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
```

That's correct, the call to autorelease is no longer needed. Likewise for the line that creates the view controller.

```
self.viewController = [[[MainViewController alloc]
    initWithNibName:@"MainViewController"
    bundle:nil] autorelease];
```

now becomes:

```
self.viewController = [[MainViewController alloc]
    initWithNibName:@"MainViewController"
    bundle:nil];
```

Before ARC, if you wrote the following you created a memory leak if the property was declared "retain":

```
self.someProperty = [[SomeClass alloc] init];
```

The init method returns an object that is retained already and placing it into the property would retain the object again. That's why you had to use `autorelease`, to balance the retain from the init method. But with ARC the above is just fine. The compiler is smart enough to figure out that it shouldn't do two retains here.

One of the things everyone loves about ARC is that in most cases it completely does away with the need to write `dealloc` methods. When an object is deallocated, its instance variables and synthesized properties are automatically released.

You no longer have to write:

```
- (void) dealloc
{
    [_window release];
    [_viewController release];
    [super dealloc];
}
```

because Objective-C automatically takes care of this now. In fact, it's not even possible to write the above anymore. Under ARC you are not allowed to call `release`, nor `[super dealloc]`. You can still implement `dealloc` – and you'll see an example of this later – but it's no longer necessary to release your instance variables by hand.

Note: Something that the conversion tool doesn't do is making `AppDelegate` a subclass of `UIResponder` instead of `NSObject`. When you create a new app using one of Xcode's templates, as of iOS 5 the `AppDelegate` class is now a subclass of `UIResponder`. It doesn't seem to do any harm to leave it as `NSObject`, but you can make it a `UIResponder` if you want to:

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

Main.m

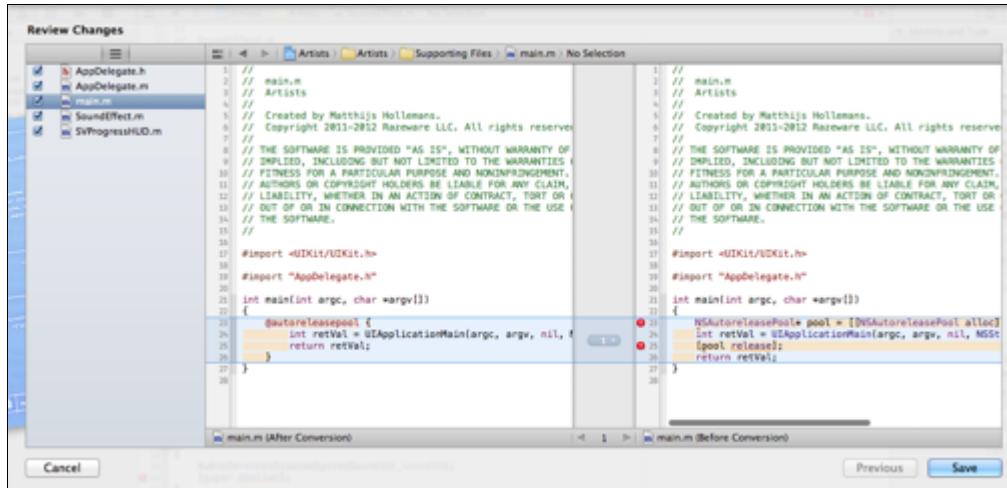
In manually memory managed apps, the `[autorelease]` method works closely together with an “autorelease pool”, which is represented by an `NSAutoreleasePool` object.

Every `main.m` has one and if you've ever worked with threads directly you've also had to make your own `NSAutoreleasePool` for each thread. Sometimes developers also put their own `NSAutoreleasePools` inside loops that do a lot of processing, just to make sure autoreleased objects created in that loop don't take up too much memory and get deleted from time to time.

Autorelease didn't go away with ARC, even though you never directly call the `[autorelease]` method on objects anymore. Any time you return an object from a method whose name doesn't start with `alloc`, `init`, `copy`, `mutableCopy` or `new`, the

ARC compiler will autorelease it for you. These objects still end up in an autorelease pool that gets drained periodically. Should you care? Not really. It just works™.

The big difference with before is that the `NSAutoreleasePool` has been retired in favor of a new language construct, `@autoreleasepool`.



The conversion tool turned the `main()` function from this,

```
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
int retVal = UIApplicationMain(argc, argv, nil,
                             NSStringFromClass([AppDelegate class]));
[pool release];
return retVal;
```

into this:

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

Not only is it simpler to read for us programmers but under the hood a lot has changed as well, making these new autorelease pools a lot faster than before. You hardly ever need to worry about autorelease with ARC, except that if you used `NSAutoreleasePool` in your code before, you will need to replace it with an `@autoreleasepool` section. The conversion tool should do that automatically for you as it did here.

SoundEffect.m

Not much changed in this file, only the call to `[super dealloc]` was removed. You are no longer allowed to call super in your `dealloc` methods.

```

Review Changes
/AppDelegate.h
/AppDelegate.m
/main.m
/SoundEffect.m
/SVProgressHUD.m

#import "SoundEffect.h"
@implementation SoundEffect
- (id)initWithSoundNamed:(NSString *)filename
{
    if ((self = [super init]))
    {
        NSURL *fileURL = [[NSBundle mainBundle] URLForResource:filename
                                                       withExtension:@"aiff"];
        if (fileURL != nil)
        {
            SystemSoundID theSoundID;
            OSSStatus error = AudioServicesCreateSystemSoundWithFile(fileURL,
                                                                      &theSoundID);
            if (error == AudioServicesNoError)
                _soundID = theSoundID;
        }
    }
    return self;
}
- (void)dealloc
{
    AudioServicesDisposeSystemSoundID(_soundID);
}
- (void)play
{
    AudioServicesPlaySystemSound(_soundID);
}
@end

SoundEffect.m (After Conversion)
SoundEffect.m (Before Conversion)

Cancel Previous Save

```

Notice that a `dealloc` method is still necessary here. In most of your classes you can simply forget about `dealloc` and let the compiler take care of things. Sometimes, however, you will need to release resources manually. That's the case with this class as well.

When the `SoundEffect` object is deallocated, a call to `AudioServicesDisposeSystemSoundID()` is still necessary to clean up the sound object and `dealloc` is the perfect place for that.

SVProgressHUD.m

This file has the most changes of them all but again they're quite trivial.

```

Review Changes
/AppDelegate.h
/AppDelegate.m
/main.m
/SoundEffect.m
/SVProgressHUD.m

// SVProgressHUD.m
// Created by Sam Vermette on 27.03.11.
// Copyright 2011 Sam Vermette. All rights reserved.

#import "SVProgressHUD.h"
#import <QuartzCore/QuartzCore.h>

@interface SVProgressHUD : NSObject
{
    @property (nonatomic, retain) SVProgressHUDMaskType *mask;
    @property (nonatomic, strong) NSTimer *fadeOutTimer;
    @property (nonatomic, retain) UILabel *stringLabel;
    @property (nonatomic, strong) UIImageView *imageView;
    @property (nonatomic, strong) UIActivityIndicatorView *activityIndicatorView;
}

- (void)showInView:(UIView *)view status:(NSString *)status;
- (void)setStatus:(NSString *)status;
- (void)dismiss;
- (void)dismissWithStatus:(NSString *)status error:(BOOL)error;
- (void)dismissWithStatus:(NSString *)status error:(BOOL)error;
- (void)memoryWarning:(NSNotification *)notification;
@end

@implementation SVProgressHUD
- (void)showInView:(UIView *)view status:(NSString *)status
{
    [self setStatus:status];
}
- (void)setStatus:(NSString *)status
{
    [self dismiss];
}
- (void)dismissWithStatus:(NSString *)status error:(BOOL)error
{
    [self dismiss];
}
- (void)dismissWithStatus:(NSString *)status error:(BOOL)error
{
    [self dismiss];
}
- (void)memoryWarning:(NSNotification *)notification
{
    [self dismiss];
}
@end

SVProgressHUD.m (After Conversion)
SVProgressHUD.m (Before Conversion)

Cancel Previous Save

```

At the top of **SVProgressHUD.m** you will find a so-called “class extension” that has several property declarations.

If you’re unfamiliar with class extensions, they are like categories except they have special powers. The declaration of a class extension looks like that of a category but it has no name between the () parentheses. Class extensions can have properties

and instance variables, something that categories can't, but you can only use them inside your .m file. (In other words, you can't create a class extension for someone else's class.)

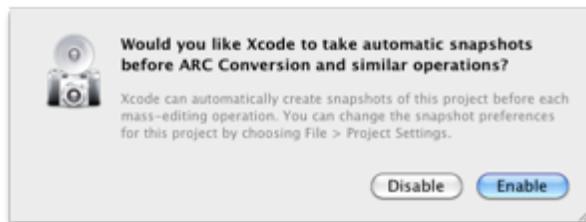
The cool thing about class extensions is that they allow you to add private properties and method names to your classes. If you don't want to expose certain properties or methods in your public `@interface`, then you can put them in a class extension. That's exactly what the author of `SVProgressHUD` did:

```
@interface SVProgressHUD ()  
  
...  
@property (nonatomic, readwrite) SVProgressHUDMaskType maskType;  
@property (nonatomic, strong) NSTimer *fadeOutTimer;  
@property (nonatomic, strong) UILabel *stringLabel;  
@property (nonatomic, strong) UIImageView *imageView;  
@property (nonatomic, strong) UIActivityIndicatorView  
    *spinnerView;  
  
...  
  
@end
```

As you've seen before, `retain` properties will become `strong` properties. If you scroll through the preview window you'll see that all the other changes are simply removal of `retain` and `release` statements.

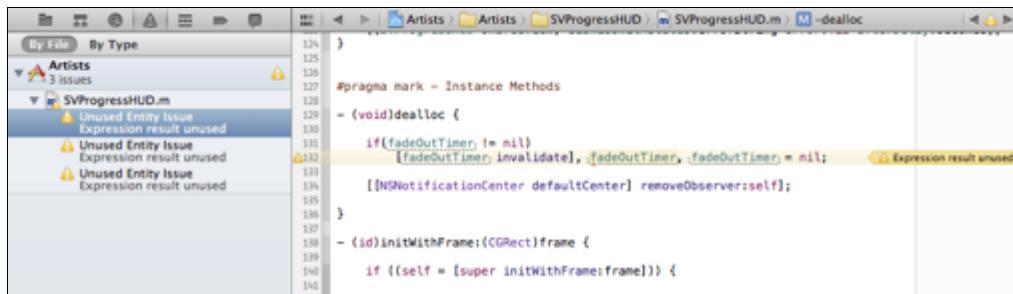
Doing It For Real

When you're satisfied with the changes that the conversion tool will make, press the Save button to make it happen. Xcode first asks whether you want it to take a snapshot of the project before it changes the files:



You should probably press Enable here. If you ever need to go back to the original code, you can find the snapshot in the Organizer window under Projects.

After the ARC conversion tool finishes, press Cmd+B to build the app. The build should complete successfully but there will be several new warnings in **SVProgressHUD.m**:



Notice again that the `dealloc` method is still used in this class, in this case to stop a timer and to unregister for notifications from `NSNotificationCenter`. Obviously, these are not things ARC will do for you.

The code at the line with the warning used to look like this:

```
if(fadeOutTimer != nil)  
    [fadeOutTimer invalidate], [fadeOutTimer release],  
    fadeOutTimer = nil;
```

Now it says:

```
if(fadeOutTimer != nil)  
    [fadeOutTimer invalidate], fadeOutTimer, fadeOutTimer = nil;
```

The tool did remove the call to `[release]`, but it left the variable in place. A variable all by itself doesn't do anything useful, hence the Xcode warning. This appears to be a situation the automated conversion tool did not foresee.

Note: If the commas confuse you, then know that it is valid in Objective-C to combine multiple expressions into a single statement using commas. The above trick is a common idiom for releasing objects and setting the corresponding instance variable to `nil`. Because everything happens in a single statement, the `if` doesn't need curly braces.

To silence the warning you can change this line, and the others like it, to:

```
if(fadeOutTimer != nil)  
    [fadeOutTimer invalidate], fadeOutTimer = nil;
```

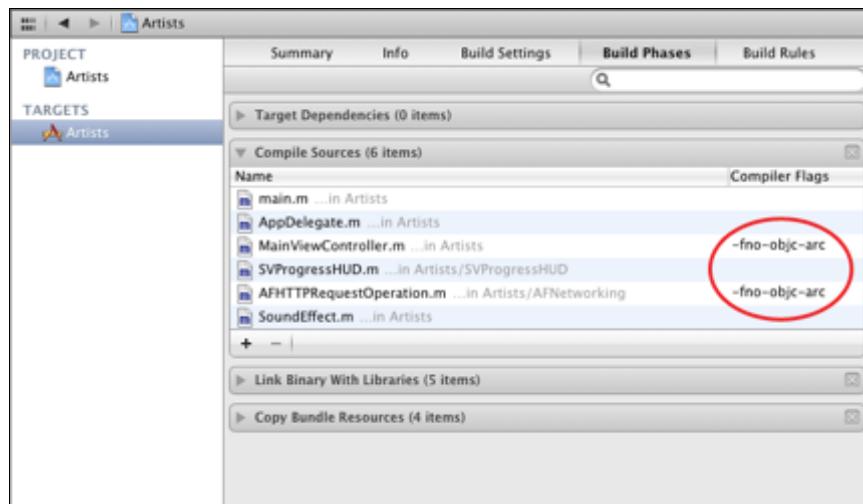
Technically speaking you don't need to set `fadeOutTimer` to `nil` in `dealloc`, because the object will automatically release any instance variables when it gets deleted.

In the other methods where the timer is invalidated, however, you definitely should set `fadeOutTimer` to `nil`. If you don't, the `SVProgressHUD` keeps hanging on to the invalidated `NSTimer` object longer than it is supposed to. (Remember, as long as there is at least one strong pointer to an object, that object stays alive.)

Build the app again and now there should be no warnings. Conversion complete!

But wait a minute... you skipped two entire classes, `MainViewController` and `AFHTTPRequestOperation`, when you did the conversion. How come they suddenly compile without problems? When you tried building the project with ARC enabled earlier there were certainly plenty of errors in those files.

The answer is simple: the conversion tool has disabled ARC for these two source files. You can see that in the **Build Phases** tab on the **Project Settings** screen:



You enabled ARC on a project-wide basis earlier when you changed the **Objective-C Automatic Reference Counting** setting under Build Settings to Yes. But you can make exceptions to this by telling the compiler to ignore ARC for specific files, using the `-fno-objc-arc` flag. Xcode will compile these files with ARC disabled.

Because it's unreasonable to expect developers to switch their entire projects to ARC at once, the folks at Apple made it possible to combine ARC code with non-ARC code in the same project.

Tip: An easy way to move only a small portion of your project to ARC is to convert only the files that you want to migrate with the conversion tool, and let it automatically add the `-fno-objc-arc` flag to the rest. You can also add this flag by hand but that's incredibly annoying when you have many files that you don't want to ARCify.

Migration Woes

As far as conversions to ARC go, this one went fairly smooth. You only had to make a single change to `SoundEffect.m` (inserting a `__bridge` statement) and then the tool did the rest.

However, the LLVM compiler is a little less forgiving in ARC mode than previous compilers so it is possible that you run into additional problems during the pre-check. You may have to edit your own code more extensively before the tool can take over.

Here's a handy reference of some issues you might run into, and some tips for how to resolve them:

“Cast ... requires a bridged cast”

This is the one you've seen before. When the compiler can't figure out by itself how to do a toll-free bridged cast, it expects you to insert a `_bridge` modifier. There are two other bridge types, `_bridge_transfer` and `_bridge_retained`, and which one you're supposed to use depends on exactly what you're trying to do. More about these other bridge types in the section on Toll-Free Bridging.

“Receiver type 'X' for instance message is a forward declaration”

If you have a class, let's say `MyView` that is a subclass of `UIView`, and you call a method on it or use one of its properties, then you have to `#import` the definition for that class. That is usually a requirement for getting your code to compile in the first place, but not always, due to the dynamic nature of Objective-C.

For example, you added a forward declaration in your .h file to announce that `MyView` is a class:

```
@class MyView;
```

Later in your .m file you do something like:

```
[myView setNeedsDisplay];
```

Previously this might have compiled and worked just fine, even without an `#import` statement. With ARC you always need to explicitly add an import:

```
#import "MyView.h"
```

“Switch case is in protected scope”

You get this error when your code does the following:

```
switch (x)
{
    case Y:
        NSString *s = ...;
        break;
}
```

This is no longer allowed. If you declare new pointer variables inside a case statement you must enclose the whole thing in curly braces:

```
switch (x)
{
    case Y:
    {
        NSString *s = ...;
        break;
    }
}
```

Now it is clear what the scope of the variable is, which ARC needs to know in order to release the object at the right moment.

“A name is referenced outside the `NSAutoreleasePool` scope that it was declared in”

You may have some code that creates its own autorelease pool:

```
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

// . . . do calculations . .

NSArray* sortedResults = [[filteredResults
                           sortedArrayUsingSelector:@selector(compare:)] retain];

[pool release];
return [sortedResults autorelease];
```

The conversion tool needs to turn that into something like this:

```
@autoreleasepool
{
    // . . . do calculations . .

    NSArray* sortedResults = [filteredResults
                              sortedArrayUsingSelector:@selector(compare:)];
}

return sortedResults;
```

But that is no longer valid code. The `sortedResults` variable is declared inside the `@autoreleasepool` scope and is therefore not accessible outside of that scope. To fix the issue you will need to move the declaration of the variable above the creation of the `NSAutoreleasePool` before starting the conversion:

```
NSArray* sortedResults;
```

```
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
. . .
```

Now the ARC conversion tool can properly rewrite your code.

“ARC forbids Objective-C objects in structs or unions”

One of the restrictions of ARC is that you can no longer put Objective-C objects inside C structs. The following code is not valid anymore:

```
typedef struct
{
    UIImage *selectedImage;
    UIImage *disabledImage;
}
ButtonImages;
```

You are recommended to replace such structs with true Objective-C classes instead. We'll talk more about this one later, and then I'll show you some other workarounds.

There may be other pre-check errors but these are the most common ones.

Note: The automated ARC conversion tool can be a little flakey if you use it more than once. If you don't convert all the files but leave some unchecked the way we did, the conversion tool may not actually do anything when you try to convert the remaining files later. My suggestion is that you run the tool just once and don't convert your files in batches.

Converting By Hand

You've converted almost the entire project to ARC already, except for `MainViewController` and `AFHTTPRequestOperation`. In this section you will find out how to convert `MainViewController` by hand. Sometimes it's fun to do things yourself so you get a better feel for what truly happens.

If you look at `MainViewController.h` you'll see that the class declares two instance variables:

```
@interface MainViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate,
UISearchBarDelegate, NSXMLParserDelegate>
{
    NSOperationQueue *_queue;
    NSMutableString *_currentStringValue;
```

```
}
```

When you think about it, the public interface of a class is a strange place to put instance variables. Usually, instance variables really are a part of the internals of your class and not something you want to expose in its public interface.

To a user of your class it isn't very important to know what the class's instance variables are. From the perspective of data hiding it is better if you move such implementation details into the `@implementation` section of the class. The new LLVM compiler version 3.0 and up make this now possible (whether you use ARC or not).

Remove the instance variable block from **MainViewController.h** and put it into **MainViewController.m**. The header file should now look like:

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

@interface MainViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate,
    UISearchBarDelegate, NSXMLParserDelegate>

@property (nonatomic, retain) IBOutlet UITableView *tableView;
@property (nonatomic, retain) IBOutlet UISearchBar *searchBar;

@end
```

And the top of **MainViewController.m** looks like:

```
@implementation MainViewController
{
    NSOperationQueue *_queue;
    NSMutableString *_currentStringValue;
}
```

Build the app and... it just works. This makes your .h files a lot cleaner and puts the instance variables where they really belong.

You can do the same for the `SoundEffect` class. Simply move the instance variable section into the .m file. Because you no longer reference the `systemSoundID` symbol anywhere in **SoundEffect.h**, you can also move the `#import` for `AudioServices.h` into **SoundEffect.m**. The `SoundEffect` header no longer exposes any details of its implementation. Nice and clean.

Note: You can also put instance variables in class extensions. This is useful for big projects where the implementation of a single class is spread over multiple files. You can then put the extension in a shared, private header file, so that

all these different implementation files have access to the same instance variables.

It's time to enable ARC on MainViewController.m. Go into the **Build Phases** settings and remove the `-fno-objc-arc` compiler flag from **MainViewController.m**. Then press Cmd+B to do a new build.

You may have some problems getting Xcode to recognize these changes. You should get a ton of compiler errors but if Xcode still says "Build Succeeded", then close the project and reopen it.

What's up with dealloc?

Let's go through these errors and fix them one by one, beginning with `dealloc`:

```

57     self.searchBar = nil;
58     self.soundEffect = nil;
59 }
60
61 - (void)dealloc
62 {
63     [_tableView release];
64     [_searchBar release];
65     [_queue release];
66     [_searchResults release];
67     [_soundEffect release];
68     [super dealloc];
69 }
70
71 - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
72 {
73     return (interfaceOrientation == UIInterfaceOrientationPortrait);
74 }
75

```

Every single line in `dealloc` gives an error. You're not supposed to call `[release]` anymore, nor `[super dealloc]`. Because this `dealloc` method isn't doing anything else, you can simply remove the entire method.

The only reason for keeping a `dealloc` method around is when you need to free certain resources that do not fall under ARC's umbrella. Examples of this are calling `CFRelease()` on Core Foundation objects, calling `free()` on memory that you allocated with `malloc()`, unregistering for notifications, invalidating a timer, and so on.

Note: Sometimes it is necessary to explicitly break a connection with an object if you are its delegate but usually this happens automatically. Most of the time delegates are weak references so when the object to be deallocated is someone else's delegate, the delegate pointer will be set to `nil` automatically when the object is destroyed. Weak pointers clean up after themselves.

By the way, in your `dealloc` method you can still use your instance variables because they haven't been released yet at that point. That doesn't happen until after `dealloc` returns.

The SoundEffect getter

The `soundEffect` method calls `release`, so that's an easy fix:

```

65 - (SoundEffect *)soundEffect
66 {
67     if (_soundEffect == nil) // lazy loading
68     {
69         SoundEffect *theSoundEffect = [[SoundEffect alloc] initWithSoundNamed:@"Sound.caf"];
70         self.soundEffect = theSoundEffect;
71         [theSoundEffect release]; // ARC forbids explicit message send of 'release'
72     }
73     return _soundEffect;
74 }
75 }
```

This method is actually the getter method for the `soundEffect` property. It employs a lazy loading technique to load the sound effect the first time it is used. The code here uses a common pattern for creating objects under manual memory management. First the new object is stored in a temporary local variable, then it is assigned to the actual property, and finally the value from the local variable is released.

This is how most Objective-C programmers used to write this sort of thing and you may have been doing it too:

```

SoundEffect *theSoundEffect = [[SoundEffect alloc]
                               initWithSoundNamed:@"Sound.caf"];
self.soundEffect = theSoundEffect;
[theSoundEffect release];
```

You could just remove the call to `release` and leave it at that, but now having a separate local variable isn't very useful anymore:

```

SoundEffect *theSoundEffect = [[SoundEffect alloc]
                               initWithSoundNamed:@"Sound.caf"];
self.soundEffect = theSoundEffect;
```

So instead, you can simplify it to just one line:

```

self.soundEffect = [[SoundEffect alloc]
                   initWithSoundNamed:@"Sound.caf"];
```

Under manual memory management this would cause a leak (there is one retain too many going on) but with ARC this sort of thing is just fine.

Please (auto)release me, let me go

Just like you can't call `release` anymore, you also cannot call `autorelease`:

```

56 - (UITableViewCell *)tableView:(UITableView *)theTableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
57 {
58     static NSString *CellIdentifier = @"Cell";
59
60     UITableViewCell *cell = [theTableView dequeueReusableCellWithIdentifier:CellIdentifier];
61     if (cell == nil)
62         cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier] autorelease];
63     if ([self.searchResults count] == 0)
64         cell.textLabel.text = @"(Nothing found)";
65     else
66         cell.textLabel.text = [self.searchResults objectAtIndex:indexPath.row];
67
68     return cell;
69 }

```

The code shows a warning at line 62: 'autorelease' is unavailable in automatic reference counting mode.

The fix is straightforward. Instead of doing,

```

cell = [[[UITableViewCell alloc]
          initWithStyle:UITableViewCellStyleDefault
          reuseIdentifier:CellIdentifier] autorelease];

```

this line becomes:

```

cell = [[UITableViewCell alloc]
          initWithStyle:UITableViewCellStyleDefault
          reuseIdentifier:CellIdentifier];

```

The next method that has errors is `escape:`, but we'll skip it just for a second. These issues are related to toll-free bridging, a topic that has a special section dedicated to it.

The remaining two errors are `releases`, in `searchBarSearchButtonClicked:` and in `parser:didEndElement::`. You can simply remove these two lines.

Do you really need all those properties?

If you look at the top of **MainViewController.m**, you'll see that it uses a class extension to declare two private properties, `searchResults` and `soundEffect`:

```

@interface MainViewController ()
@property (nonatomic, retain) NSMutableArray *searchResults;
@property (nonatomic, retain) SoundEffect *soundEffect;
@end

```

This is done primarily to make manual memory management easier and it's a common reason why developers use properties. When you do,

```

self.searchResults = [NSMutableArray arrayWithCapacity:10];

```

the setter will take care of releasing the old value that is stored in the property (if any) and properly retaining the new value. Developers have been using properties as a way of having to think less about when you need to retain and when you need to release. But now with ARC you don't have to think about this at all!

In the author's opinion, using properties just for the purposes of simplifying memory management is no longer necessary. You can still do so if you want to but it's simpler to just use instance variables now, and only use properties when you need to make data accessible to other classes from your public interface.

Therefore, remove the class extension and the `@synthesize` statements for `searchResults` and `soundEffect`. Add new instance variables to replace them:

```
@implementation MainViewController
{
    NSOperationQueue *_queue;
    NSMutableString *_currentStringValue;
    NSMutableArray *_searchResults;
    SoundEffect *_soundEffect;
}
```

Note that these new instance variables have an underscore in front of their names. That's a common Objective-C convention that intends to make it clear whether a symbol represents a local variable (no prefix), an instance variable (underscore prefix), or a property (using the "self." prefix).

Of course, replacing the properties with instance variables means you can no longer refer to them as `self.searchResults` and `self.soundEffect`. Change `viewDidLoad` to the following:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.tableView = nil;
    self.searchBar = nil;
    _soundEffect = nil;
}
```

The `soundEffect` method now becomes:

```
- (SoundEffect *)soundEffect
{
    if (_soundEffect == nil) // lazy loading
    {
        _soundEffect = [[SoundEffect alloc]
                        initWithSoundNamed:@"Sound.caf"];
    }
    return _soundEffect;
}
```

That's about as simple as you can make it. The `SoundEffect` object is allocated and assigned to the `_soundEffect` instance variable. This variable becomes its owner

and the object will stay alive until `_soundEffect` is set to `nil` (in `viewDidUnload`), or until the `MainViewController` is deallocated (because that will also release all its instance variables).

In the rest of the file, replace anywhere where it says `self.searchResults` with just `_searchResults`. When you build the app again, the only errors it should give are on the `escape:` method.

Note that `searchBarSearchButtonClicked` still does:

```
[self.soundEffect play];
```

This will work even though you no longer have a property named `soundEffect`. The dot syntax isn't restricted to just properties, although that's what it is most commonly used for.

If using dot syntax here offends you, you can change this line to a regular method call:

```
[[self soundEffect] play];
```

Don't change it to this, though:

```
[soundEffect play];
```

Because the code uses lazy loading to create the `soundEffect` object, the `_soundEffect` instance variable will always be `nil` until you call the `soundEffect` method at least once. Therefore, to be certain you actually have a loaded `SoundEffect` object, you should always access it through `self`.

Note: If you feel that this usage pattern does morally require you to declare `soundEffect` as a `@property`, then go right ahead. Different strokes for different folks. :-)

If it's a property, always use self

As a best practice, if you define something as a property, then you should *always* use it as a property and not through its backing variable.

The only places where you can access the property's backing instance variable directly is in the `init` and `dealloc` methods for your class and when you provide a custom getter and setter. Anywhere else you should access the property through `self.propertyName`.

That is why `@synthesize` statements often rename the instance variable by putting an underscore in front:

```
@synthesize propertyName = _propertyName;
```

This construct will prevent you from accidentally using the backing instance variable by typing "propertyName" when you really meant to use "self.propertyName".

Tip: With the latest versions of Xcode you're no longer required to manually synthesize your properties. You can completely leave out the `@synthesize` statement and the compiler will automatically create the backing instance variable for you. The name of this variable is the same as the property name, but again with an additional underscore in front.

This rule is especially important for `readonly` properties. ARC can get confused if you modify such properties by changing their instance variables, and strange bugs will result. The correct way is to redefine the property as `readwrite` in a class extension.

In the .h file, you declare the property as `readonly`:

```
@interface WeatherPredictor  
@property (nonatomic, strong, readonly) NSNumber *temperature;  
@end
```

And in the .m file you declare the property again but now as `readwrite`:

```
@interface WeatherPredictor()  
@property (nonatomic, strong, readwrite) NSNumber *temperature;  
@end
```

Make it weak or leak

Speaking of properties, `MainViewController` still has two outlet properties in its .h file:

```
@property (nonatomic, retain) IBOutlet UITableView *tableView;  
@property (nonatomic, retain) IBOutlet UISearchBar *searchBar;
```

The `retain` keyword for properties still works with ARC and is simply a synonym for `strong`. It is best to call your properties `strong` because that's the proper term from now on.

For these two particular properties, however, we have other plans. Instead of `strong`, you will declare them as `weak`:

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

Weak is the recommended relationship for all *outlet* properties, in other words properties that point to views in your nib or storyboard. These view objects are already part of the view controller's view hierarchy and don't need to be retained elsewhere. Your view controller just wants to refer to them but not become their co-owner.

What is the benefit of declaring your outlets `weak` instead of `strong`? Peace of mind. When you make an outlet a strong pointer, you give the view controller partial responsibility for managing the lifetime of the view, but that really is UIKit's business, not yours.

By setting the outlet properties to be weak, you relieve the view controller of any ownership duties. Primarily that means you no longer have to worry about setting your outlet properties to `nil` when the view gets unloaded in a low-memory situation.

Note: Exactly what happens when the iPhone receives a low-memory warning varies depending on the version of iOS you're using. On iOS 5 and below, UIKit will unload the view of any view controller that is not currently visible and calls the `viewDidUnload` method to let the view controller know that its view object is no longer alive.

On iOS 6, however, views are no longer automatically unloaded and as a result `viewDidUnload` is never called. If you're unaware of this difference, your code can have subtle bugs that only show up when the iPhone runs out of free memory space – and you don't want your apps to crash because of that!

The `viewDidUnload` method currently looks like this:

```
- (void)viewDidUnload
{
    [super viewDidUnload];
    self.tableView = nil;
    self.searchBar = nil;
    _soundEffect = nil;
}
```

When the iPhone receives a low-memory warning on iOS 5, the view controller's main view gets unloaded, which releases all of its subviews as well. At that point the `UITableView` and `UISearchBar` objects cease to exist and the zeroing weak pointer system automatically sets `self.tableView` and `self.searchBar` to `nil`.

Because they are now zeroing weak pointers, there is no more need to set the properties to `nil` by yourself in `viewDidUnload`. In fact, by the time `viewDidUnload` gets called these properties already are `nil`. Therefore, you can simplify this method to do just:

```
- (void)viewDidUnload
{
    [super viewDidUnload];
    _soundEffect = nil;
}
```

But remember, if your app is running on iOS 6 and a low-memory situation occurs, the view will not be unloaded and `viewDidUnload` method will *not* be called. So how do you handle this in a way that works on both iOS 5 and 6?

The recommendation is to no longer write `viewDidUnload` methods, even if you still want to support iOS 5, and to move any cleanup code into `didReceiveMemoryWarning` instead. Because the outlets are now weak, you no longer have to worry about them, but you're still responsible for cleaning up any other data that you do own. If you no longer want to hang on to objects, you need to set their pointers to `nil` explicitly.

Remove `viewDidUnload` and replace it with:

```
- (void) didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    _soundEffect = nil;
}
```

In the event of a low-memory warning, this method sets `_soundEffect` to `nil` to force the deallocation of the `SoundEffect` object. You should free as much memory as possible and the `SoundEffect` object is expendable at that point. It can be reloaded whenever you need it again.

Because instance variables create strong relationships by default, setting `_soundEffect` to `nil` will remove the owner from the `SoundEffect` object and it will be deallocated immediately.

Properties recap

To summarize, the new modifiers for properties are:

- **strong.** This is a synonym for the old “retain”. A strong property becomes an owner of the object it points to.
- **weak.** This is a property that represents a weak pointer. It will automatically be set to `nil` when the pointed-to object is destroyed. Remember, use this for outlets.
- **unsafe_unretained.** This is a synonym for the old “assign”. You use it only in exceptional situations and when you want to target iOS 4. More about this later.
- **copy.** This is still the same as before. It makes a copy of the object and creates a strong relationship.

- **assign.** You're no longer supposed to use this for objects, but you still use it for primitive values such as BOOL, int, and float.

Note: Remember that strong and weak are only for objects. For primitive values such as int, float, NSInteger, and BOOL you keep using assign, just like you did before.

```
Correct: @property (nonatomic, assign) BOOL yesOrNo;
WRONG:    @property (nonatomic, strong) BOOL yesOrNo;
```

Toll-Free Bridging

Let's fix that one last method so you can run the app again.

```
117 - (NSString *)escape:(NSString *)text
118 {
119     return [[NSString alloc] initWithUTF8String:[CFURLCreateStringByAddingPercentEscapes(
120         NULL,
121         (CFStringRef)text,
122         NULL,
123         (CFStringRef)@"!*'();:@&=+$,/?%#[ ]",
124         CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding))
125         autorelease];
126 }
127 }
128 }
```

This method uses the `CFURLCreateStringByAddingPercentEscapes()` function to URL-encode a string. We use it to make sure any spaces or other characters in the search text that the user types get converted to something that is valid for use in an HTTP GET request.

The compiler gives several errors:

- ARC Casting Rules: Cast of C pointer type 'CFStringRef' (aka 'const struct __CFString *') to Objective-C pointer type 'NSString *' requires a bridged cast
- Semantic Issue: 'autorelease' is unavailable: not available in automatic reference counting mode
- ARC Restrictions: ARC forbids explicit message send of 'autorelease'

The last two errors are really the same and simply mean that you cannot call `[autorelease]`. Let's get rid of that statement first. What remains is this:

```
- (NSString *)escape:(NSString *)text
{
    return (NSString *)CFURLCreateStringByAddingPercentEscapes(
        NULL,
        (CFStringRef)text,
        NULL,
        (CFStringRef)@"!*'();:@&=+$,/?%#[ ]",
        CFStringConvertNSStringEncodingToEncoding(
            NSUTF8StringEncoding));
}
```

```
}
```

The remaining error has to do with a cast that apparently should be “bridged”. There are three casts in this method:

- (`NSString *`)`CFURLCreateStringByAddingPercentEscapes`(...)
- (`CFStringRef`)`text`
- (`CFStringRef`)`@"!*'();:@&=+$,/?%#[]"`

The compiler only complains about the first one. Before you’ll fix it, let’s take a closer look at what “bridging” really means.

Bridged casts are necessary when you move an object between worlds. On the one hand there is the world of Objective-C, on the other there is Core Foundation.

For most apps these days there isn’t a big need to use Core Foundation, you can do almost anything you want from comfortable Objective-C classes. However, some lower-levels APIs such as Core Graphics and Core Text are based on Core Foundation and it’s unlikely there will ever be an Objective-C version of them. Fortunately, the designers of iOS made it really easy to move certain objects between these two different kingdoms. And you won’t be charged a thing!

`NSString` is an Objective-C object that represents a list of characters – you’ve probably used it once or twice before – and `CFStringRef` is a Core Foundation object that does the same thing. For all intents and purposes, `NSString` and `CFStringRef` can be treated as the being same.

You can take an `NSString` object and use it as if it were a `CFStringRef`, and take a `CFStringRef` object and use it as if it were an `NSString`. That’s the idea behind toll-free bridging.

Previously, in the days before ARC, that was as simple as doing:

```
CFStringRef s1 = [[NSString alloc] initWithFormat:  
    @"Hello, %@", name];
```

Of course, you also had to remember to release the object when you were done with it:

```
CFRelease(s1);
```

The other way around, from Core Foundation to Objective-C, was just as easy:

```
CFStringRef s2 = CFStringCreateWithCString(kCFAllocatorDefault,  
    bytes, kCFStringEncodingMacRoman);  
  
NSString *s3 = (NSString *)s2;  
  
// release the object when you're done
```

```
[s3 release];
```

Now that we have ARC, the compiler needs to know who is responsible for releasing such casted objects. If you treat an `NSObject` as a Core Foundation object, then it is no longer ARC's responsibility to release it. But you do need to tell ARC about your intentions; the compiler cannot infer this by itself.

Likewise, if you create a Core Foundation object but then cast it to an `NSObject`, you need to tell ARC to take ownership of it and delete that object when its time comes. That's what the bridging casts are for.

The `CFURLCreateStringByAddingPercentEscapes()` function takes a handful of parameters, two of which are `CFStringRef` objects. It also returns a new `CFStringRef` object. Because we're Objective-C programmers, we prefer to work with `NSStrings` instead. Previously you could just cast these `NSString` objects into a `CFStringRef`, and vice versa, but with ARC the compiler needs more information.

The cast,

```
(CFStringRef)@"!*'();:@&=+$,/?%#[ ]"
```

is a cast of a constant object and that doesn't require any special memory management. This is a string literal that will be baked into the application executable. Unlike "real" objects, it is never allocated or freed. No problems here.

Note: If you wanted to, you could also write this as:

```
CFSTR("!*'();:@&=+$,/?%#[ ]")
```

The `CFSTR()` macro creates a `CFStringRef` object from the specified string. The string literal here is a regular C string and therefore doesn't begin with the @ sign. Instead of making an `NSString` object and casting it to a `CFStringRef`, you now directly make a `CFStringRef` object. Which one you like better is largely a matter of taste, as they both deliver the exact same results.

The cast,

```
(CFStringRef)text
```

converts the contents of the `text` parameter, which again is an `NSString` object, into a `CFStringRef`. Like local variables, method parameters are strong pointers; their objects are retained upon entry to the method. The value from the `text` variable will continue to exist until the pointer is destroyed. Because it is a local variable that happens when the `escape:` method ends.

We want ARC to stay the owner of this variable but we also want to temporarily treat it as a `CFStringRef`. For this type of situation, the `__bridge` specifier is used. It

tells ARC that no change in ownership is taking place and that it should release the object using the normal rules.

You've already used `__bridge` before in **SoundEffect.m**:

```
OSStatus error = AudioServicesCreateSystemSoundID(
    (__bridge CFURLRef) fileURL, &theSoundID);
```

The exact same situation applies there. The `fileURL` variable contains an `NSURL` object and is managed by ARC. The `AudioServicesCreateSystemSoundID()` function, however, expects a `CFURLRef` object. Fortunately, `NSURL` and `CFURLRef` are toll-free bridged so you can cast the one into the other. Because you still want ARC to release the object when you're done with it, you use the `__bridge` keyword to indicate that ARC remains in charge.

Change the `escape:` method to the following:

```
- (NSString *)escape:(NSString *)text
{
    return (NSString *)CFURLCreateStringByAddingPercentEscapes(
        NULL,
        (__bridge CFStringRef)text,
        NULL,
        (CFStringRef)@"!*'();:@&=+$,/?%#[ ]",
        CFStringConvertNSStringEncodingToEncoding(
            NSUTF8StringEncoding));
}
```

Now the value from the `text` parameter is properly bridged. It lets the compiler know that the ownership of this object did not change and that ARC is still responsible for cleaning up afterwards.

Note: With the latest version of the LLVM compiler, the `__bridge` specifier isn't really required anymore for simple cases such as this. The compiler is smart enough to figure out that you intended a bridge cast here.

Most of the time when you cast an Objective-C object to a Core Foundation object or vice versa, you'll want to use `__bridge`. However, there are times when you do want to give ARC ownership of a Core Foundation object, or relieve ARC of its ownership. In that case there are two other bridging casts that you can use:

- **`__bridge_transfer`**: Give ARC ownership
- **`__bridge_released`**: Relieve ARC of its ownership

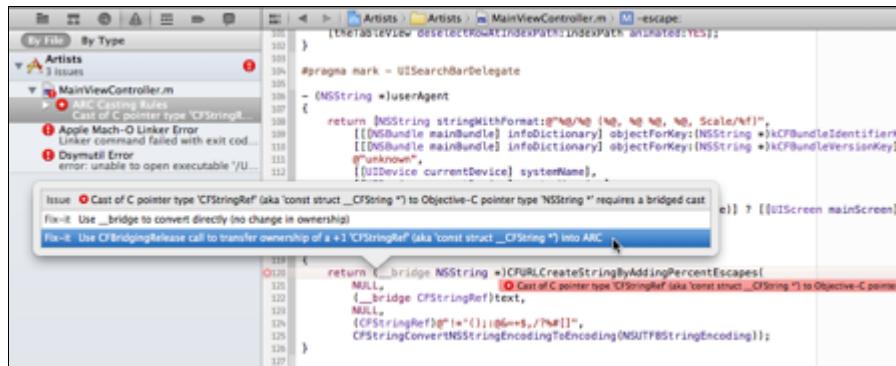
Instead of `__bridge_transfer`, the helper function `CFBridgingRelease()` is often used, as that makes it a bit clearer what the purpose of the cast is. You use it

where you would normally have used a `CFRelease()`. In a similar vein, the helper function for `_bridge_retained` is `CFBridgingRetain()` and it is used in the place of the regular `CFRetain()`.

There is one error remaining in the source file, on the line:

```
return (NSString *)CFURLCreateStringByAddingPercentEscapes(
```

Click on the error message and the following Fix-it will pop up:



It gives two possible solutions: `_bridge` and `CFBridgingRelease()`. The correct choice here is `CFBridgingRelease()`, which is the same as the `_bridge_transfer` cast.

The `CFURLCreateStringByAddingPercentEscapes()` function creates and returns a new `CFStringRef` object. Of course, you'd rather use `NSString` so you need to do a bridging cast. What you're really attempting to do in this method is this:

```
CFStringRef result = CFURLCreateStringByAddingPercentEscapes();
NSString *s = (NSString *)result;
return s;
```

Because the function has the word "Create" in its name, it returns a retained object (according to the Core Foundation memory management rules). Someone is responsible for releasing that retained object at some point. If the `escape:` method wasn't returning this object as an `NSString`, then the code may have looked something like this:

```
- (void)someMethod
{
    CFStringRef s = CFURLCreateStringByAddingPercentEscapes();

    // do something with the string
    // . . .

    CFRelease(s);
}
```

Remember that ARC only works for Objective-C objects, not for objects that are created by Core Foundation. You still need to call `CFRelease()` on such objects yourself!

What you want to do in `escape:` is convert that new `CFStringRef` object to an `NSString` object, and then ARC should automatically release that string whenever you're no longer using it. But ARC needs to be told about this.

Therefore, you use the `CFBridgingRelease()` function (or the `_bridge_transfer` cast) to say: "Hey ARC, this `CFStringRef` object is now an `NSString` object and I want you to dispose of it, so that I don't have to call `CFRelease()` on it myself."

The final version of the `escape:` method becomes:

```
- (NSString *)escape:(NSString *)text
{
    return (NSString *)CFBridgingRelease(
        CFURLCreateStringByAddingPercentEscapes(
            NULL,
            (__bridge CFStringRef)text,
            NULL,
            (CFStringRef)@"!*'();:@&=+$,/?%#[ ]",
            CFStringConvertNSStringEncodingToEncoding(
                NSUTF8StringEncoding)));
}
```

If you were to just use `_bridge` instead, then your app would have a memory leak. ARC doesn't know that it should release the object when you're done with it and no one calls `CFRelease()`. As a result, the object will stay in memory forever. It's important that you pick the proper bridge specifier!

Another common framework that requires these bridging casts is the AddressBook framework. For example:

```
- (NSString *)firstName
{
    return CFBridgingRelease(ABRecordCopyCompositeName(...));
}
```

Remember, anywhere you call a Core Foundation function named Create, Copy, or Retain you must do `CFBridgingRelease()` to safely transfer the value to ARC.

What about the other one, `CFBridgingRetain()` or `_bridge_retained`? You would use that going the other way around. Suppose you have an `NSString` and you need to give that to some Core Foundation API that wants to take ownership of your string object. You don't want ARC to also release that object, because then it would be released one time too many and apps have a tendency to crash when that happens.

In other words, you use `CFBridgingRetain()` to give the object to Core Foundation so that ARC is no longer responsible for releasing it. An example:

```
NSString *s1 = [[NSString alloc] initWithFormat:  
                  @"Hello, %@", name];  
  
CFStringRef s2 = CFBridgingRetain(s1);  
  
// do something with s2  
// . . .  
  
CFRelease(s2);
```

As soon as the `CFBridgingRetain()` cast happens, ARC considers itself no longer duty-bound to release the string object. The call to `CFRelease()` is properly balanced by `CFBridgingRetain()`. If you had used `__bridge` in this example, then the app would likely crash. ARC might deallocate the string object before the Core Foundation code is done with it.

I doubt you'll need to use this particular bridge type often. Off the top of my head, I can't think of a single API that is commonly used that requires this.

It is unlikely that you have a lot of Core Foundation code in your apps anyway. Most frameworks that you'll use are Objective-C, with the exception of Core Graphics (which doesn't have any toll-free bridged types), the Address Book, and the occasional low-level function. But if you do, rest assured that the compiler would point out to you when you need to use a bridging cast.

Note: Not all Objective-C and Core Foundation objects that sound alike are toll-free bridged. For example, `CGImage` and `UIImage` cannot be cast to each other, and neither can `cgcolor` and `UIColor`. The following page lists the types that can be used interchangeably: <http://bit.ly/j65Ceo>

The bridging casts are not limited to interactions with Core Foundation. Some APIs take `void *` pointers that let you store a reference to anything you want, whether that's an Objective-C object, a Core Foundation object, a `malloc()`'d memory buffer, and so on. The notation `void *` means: this is a pointer but the actual datatype of what it points to could be anything.

To convert from an Objective-C object to `void *`, or the other way around, you will need to do a `__bridge` cast. For example:

```
MyClass *myObject = [[MyClass alloc] init];  
[UIView beginAnimations:nil context:(__bridge void *)myObject];
```

In the animation delegate method, you do the conversion in reverse to get your object back:

```
- (void)animationDidStart:(NSString *)animationID
                      context:(void *)context
{
    MyClass *myObject = (__bridge MyClass *)context;
    . . .
}
```

You'll see another example of this in the next chapter, where we cover using ARC with Cocos2D.

To summarize:

- When changing ownership from Core Foundation to Objective-C you use `CFBridgingRelease()` or `__bridge_transfer`.
- When changing ownership from Objective-C to Core Foundation you use `CFBridgingRetain()` or `__bridge_retained`.
- When you want to use one type temporarily as if it were another without ownership change, you use `__bridge`.

That's it as far as `MainViewController` is concerned. All the errors should be gone now and you can build and run the app.

The conversion of the Artists app to ARC is complete!

(We won't convert `AFHTTPRequestOperation` to ARC in this tutorial. Compiling it with the flag `-fno-objc-arc` is sufficient to make it work.)

Note: For the near future you may find that many of your favorite third-party libraries do not come in an ARC flavor yet. It's no fun to maintain two versions of a library, one without ARC and one with, so expect many library maintainers to pick just one.

New libraries might be written for ARC only but older ones may prove too hard to convert. Therefore it's likely that a portion of your code will remain with ARC disabled (the `-fno-objc-arc` compiler flag).

Fortunately, ARC works on a per-file basis so it's no problem at all to combine these libraries with your own ARCified projects. Because it's sometimes a bit of a hassle to disable ARC for a large selection of files, we'll talk about smarter ways to put non-ARC third-party libraries into your projects in the next chapter.

Delegates and Weak Properties

The app you've seen so far is very simple and demonstrates only a few facets of ARC. To show you the rest, you'll first have to add a new screen to the app.

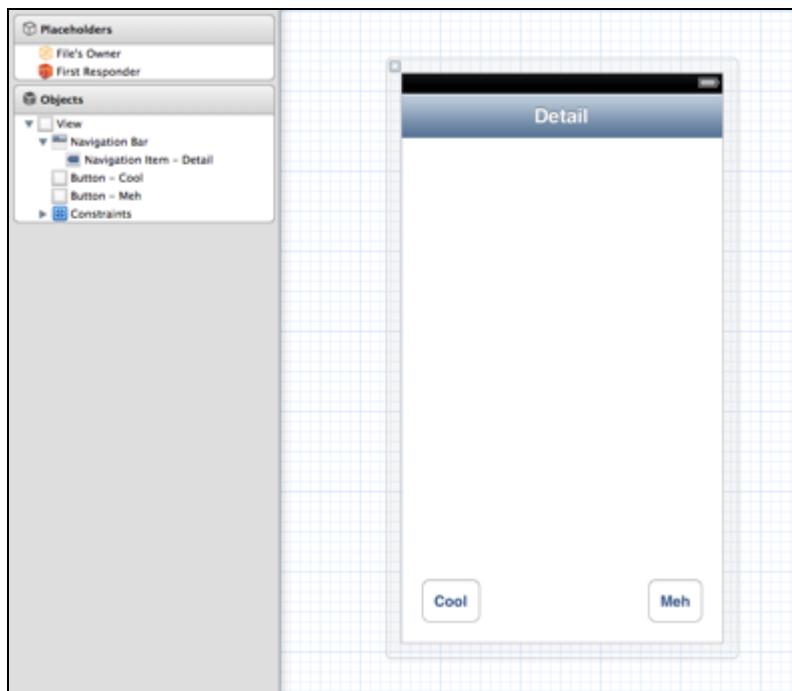
Using Xcode's **File\New File** menu, add a new **UIViewController subclass** to the project, with XIB for user interface, and name it **DetailViewController**.

Add two action methods to **DetailViewController.h**:

```
@interface DetailViewController : UIViewController  
  
- (IBAction)coolAction;  
- (IBAction)mehAction;  
  
@end
```

You will connect these actions to two buttons in the nib.

Open **DetailViewController.xib** and add a navigation bar and two buttons:



Note: By default, Xcode enables Auto Layout for all new nib files you create. This technology was added to make user interfaces easily resizable to fit different screen sizes – particularly the new dimensions of the iPhone 5 – and works on iOS 6 only. To make sure the Artists app still runs on iOS 5 as well, you should uncheck the **Use Autolayout** box in the nib's File Inspector.

Control-drag from each button to File's Owner and connect their Touch Up Inside events to their respective actions.

In **DetailViewController.m**, add the implementation of the two action methods to the bottom. For now leave these methods empty:

```
- (IBAction)coolAction
{
}

- (IBAction)mehAction
{
}
```

You will make some changes to the main view controller so that it invokes this Detail screen when you tap on a search result.

First, add an import to **MainViewController.h** (and not in the .m, for reasons that will become clear soon):

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

Then in **MainViewController.m**, change the `didSelectRowAtIndexPath` to:

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [theTableView deselectRowAtIndexPath:indexPath animated:YES];

    DetailViewController *controller =
        [[DetailViewController alloc]
            initWithNibName:@"DetailViewController" bundle:nil];

    [self presentViewController:controller animated:YES
                      completion:nil];
}
```

This instantiates the `DetailViewController` and presents it on top of the current one.

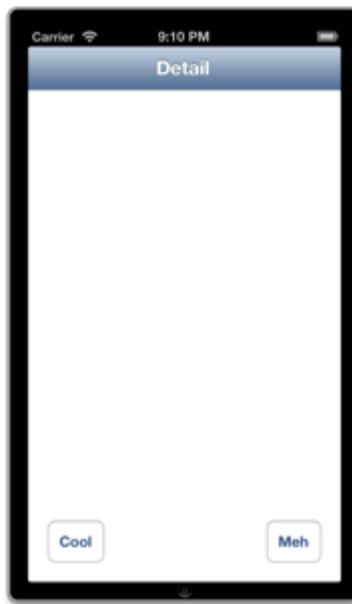
Add the following method:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if ([_searchResults count] == 0)
        return nil;
    else
```

```
    return indexPath;
}
```

If there are no search results the app puts a single row into the table that says "(Nothing found)". You don't want to open the Detail screen when the user taps that row.

If you run the app now, tapping on a row brings up the Detail screen, but you cannot close it yet. The actions that are wired to the "Cool" and "Meh" buttons are still empty and pressing the buttons has no effect.



To fix this, you will give the Detail screen a delegate. That's how you commonly make this type of arrangement work. If screen A invokes screen B, and screen B needs to tell A something – for example, that it needs to close – you make A the delegate of B. Certainly you've seen this pattern before as it's used throughout the iOS API.

Change **DetailViewController.h** to the following:

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

@class DetailViewController;

@protocol DetailViewControllerDelegate <NSObject>

- (void)detailViewController:(DetailViewController *)controller
    didPickButtonWithIndex:(NSInteger)buttonIndex;

@end
```

```
@interface DetailViewController : UIViewController

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

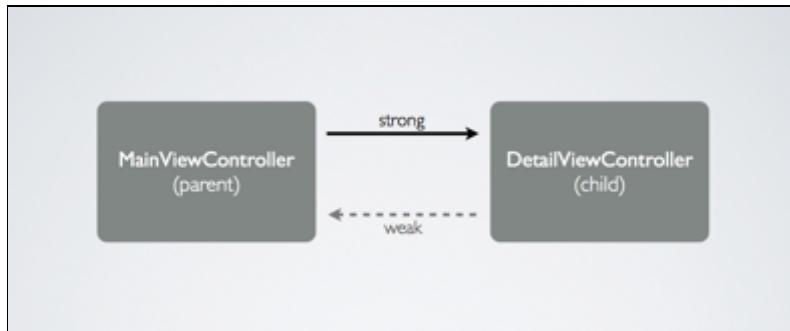
- (IBAction)coolAction;
- (IBAction)mehAction;

@end
```

You've added a delegate protocol with a single method, as well as a property for that delegate. Notice that the property is declared "weak". Making the delegate pointer weak is necessary to prevent ownership cycles.

You may be familiar with the concept of a retain cycle, where two objects retain each other so neither will ever be deallocated. That's a common form of memory leak. In systems that employ garbage collection (GC) to handle their memory management, the garbage collector can recognize such cycles and release them anyway. But ARC is not garbage collection and for dealing with ownership cycles you're still on your own. The weak pointer is an important tool for breaking such cycles.

The `MainViewController` creates the `DetailViewController` and presents it on the screen. That gives it a strong reference to this object. The `DetailViewController` in turn has a reference to a delegate. It doesn't really care which object is its delegate but most of the time that will be the view controller that presented it, in other words `MainViewController`. So here is a situation where two objects point at each other:



If both of these pointers were strong, then you would have an ownership cycle. It is best to prevent such cycles. The parent (`MainViewController`) owns the child (`DetailViewController`) through a strong pointer. If the child needs a reference back to the parent, through a delegate or otherwise, it should use a weak pointer.

Therefore, the rule is that delegates should be declared weak. Most of the time your properties and instance variables will be strong, but this is an exception.

In `DetailViewController.m`, change the action methods to:

```

- (IBAction)coolAction
{
    [self.delegate detailViewController:self
                      didPickButtonWithIndex:0];
}

- (IBAction)mehAction
{
    [self.delegate detailViewController:self
                      didPickButtonWithIndex:1];
}

```

In **MainViewController.h**, add `DetailViewControllerDelegate` to the `@interface` line:

```

@interface MainViewController : UIViewController
<UITableViewDataSource, UITableViewDelegate,
UISearchBarDelegate, NSXMLParserDelegate,
DetailViewControllerDelegate>

```

(This is why you added the `#import` statement to the `.h` file earlier, instead of to the `.m` file, so that it knows about the `DetailViewControllerDelegate` protocol.)

In **MainViewController.m**, change `didSelectRowAtIndexPath` to set the delegate property:

```

- (void)tableView:(UITableView *)theTableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [theTableView deselectRowAtIndexPath:indexPath animated:YES];

    DetailViewController *controller =
        [[DetailViewController alloc]
            initWithNibName:@"DetailViewController" bundle:nil];

    controller.delegate = self;

    [self presentViewController:controller animated:YES
                      completion:nil];
}

```

And finally, add the following to the bottom:

```

#pragma mark - DetailViewControllerDelegate

- (void)detailViewController:(DetailViewController *)controller
    didPickButtonWithIndex:(NSInteger)buttonIndex

```

```
{  
    NSLog(@"Picked button %d", buttonIndex);  
    [self dismissViewControllerAnimated:YES completion:nil];  
}
```

Here you simply dismiss the view controller. Run the app and try it out. Now you can press the Cool or Meh buttons to close the Detail screen.

Just to verify that `DetailViewController` gets released, give it a `dealloc` method that prints something to the Xcode debug pane:

```
- (void)dealloc  
{  
    NSLog(@"dealloc DetailViewController");  
}
```

In this case you could actually get away with making the delegate property strong (try it out if you don't believe me). As soon as the `MainViewController` calls `dismissViewControllerAnimated`, it loses the strong reference to `DetailViewController`. At that point there are no more pointers to that object and it will go away.

Still, it's a good idea to stick to the recommended pattern:

- **parent pointing to a child:** strong
- **child pointing to a parent:** weak

The child should not be helping to keep the parent alive. You'll see examples of ownership cycles that do cause problems when we talk about blocks in the Intermediate ARC chapter.

The devil is in the details

The Detail screen isn't very exciting yet but you can make it a little more interesting by putting the name of the selected artist in the navigation bar.

Add the following to `DetailViewController.h`:

```
@property (nonatomic, strong) NSString *artistName;  
@property (nonatomic, weak) IBOutlet UINavigationBar  
    *navigationBar;
```

The `artistName` property will contain the name of the selected artist. Previously you would have made this a `retain` property (or perhaps `copy`), so now it becomes `strong`.

The `navigationBar` property is an outlet. As before, outlets that are not top-level objects in the nib should be made weak so they are automatically released in low-memory situations on iOS 5 (but not on iOS 6).

There is no need to synthesize these properties as the LLVM compiler that comes with Xcode 4.5 automatically does that for you. Thank goodness for small favors. ☺

In **DetailViewController.m**, change `viewDidLoad` to:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationBar.topItem.title = self.artistName;
}
```

Don't forget to connect the navigation bar from the nib file to the outlet! (Hint: ctrl-drag from File's Owner to the navigation bar.)

In **MainViewController.m**, change `didSelectRowAtIndexPath` to:

```
- (void)tableView:(UITableView *)theTableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [theTableView deselectRowAtIndexPath:indexPath animated:YES];

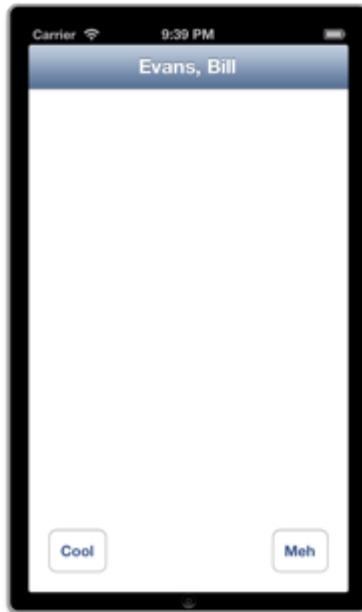
    NSString *artistName = _searchResults[indexPath.row];

    DetailViewController *controller =
        [[DetailViewController alloc]
            initWithNibName:@"DetailViewController" bundle:nil];

    controller.delegate = self;
    controller.artistName = artistName;

    [self presentViewController:controller animated:YES
                      completion:nil];
}
```

Run the app and you'll see the name of the artist in the navigation bar:



Note: To read the artist name from the search results array, the code does:

```
NSString *artistName = _searchResults[indexPath.row];
```

This is a sweet new feature in Objective-C. You are probably used to calling the `[array objectAtIndex:]` method to index an `NSArray`, but with the latest LLVM compiler you can use the simpler `[]` notation instead.

Often developers use “copy” properties for objects of classes such as `NSString` and `NSArray`. This is done to make sure no one can change that object after you have put it into the property. Even though an `NSString` object is immutable once created, the actual object given to the property could be an `NSMutableString` that can be modified afterward.

Using the `copy` modifier is still possible with ARC. If you’re slightly paranoid about your properties being truly immutable, then change the declaration of the `artistName` property to:

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

By adding the `copy` modifier, it makes it so that when we assign to the property like this,

```
controller.artistName = artistName;
```

the app first makes a copy of the string object from the local variable and then stores that copy into the property. Other than that, this property works exactly the same way as a strong reference.

Let's see what happens when you log the values of `artistName` and `navigationBar` in the `dealloc` method in **DetailViewController.m**:

```
- (void)dealloc
{
    NSLog(@"dealloc DetailViewController");
    NSLog(@"artistName '%@'", self.artistName);
    NSLog(@"navigationBar %@", self.navigationBar);
}
```

Run the app and close the Detail screen, and you will see that both properties still have their values:

```
Artists[6989:c07] dealloc DetailViewController
Artists[6989:c07] artistName 'Evans, Bill'
Artists[6989:c07] navigationBar <UINavigationBar: 0x8142700; frame = (0
0; 320 44); autoresizingMask = W+BM; gestureRecognizers = <NSArray: 0x8144370>;
layer = <CALayer: 0x8141800>
```

However, as soon as `dealloc` is over, these objects will be released and deallocated (since no one else is holding on to them). That is to say, the string object from `artistName` will be released and the `UINavigationBar` object is freed as part of the view hierarchy. The `navigationBar` property itself is weak and is therefore excluded from memory management.

Now that we have this second screen in the app, you can test what happens with the `MainViewController`'s view in low-memory situations. To do this, add some `NSLog()` statements to the `didReceiveMemoryWarning` method in **MainViewController.m**:

```
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    _soundEffect = nil;

    NSLog(@"tableView %@", self.tableView);
    NSLog(@"searchBar %@", self.searchBar);
}
```

Run the app and open the Detail screen. Then from the Simulator's Hardware menu, choose **Simulate Memory Warning**. What you will see next in the Xcode debug pane depends on which version of iOS you're running the app.

On iOS 6, you will get the following output:

```
Artists[7049:c07] Received memory warning.
Artists[7049:c07] tableView <UITableView: 0x8a01000; ...>
```

```
Artists[7049:c07] searchBar <UISearchBar: 0x81abfd0; ...>
```

Because iOS 6 no longer unloads the views of any view controllers that are not visible, the `UITableView` and `UISearchBar` objects stay in memory. Simulate another memory warning and you should see the same pointer values in the debug output.

On iOS 5, however, the output looks like this:

```
Artists[7123:c07] Received memory warning.  
Artists[7123:c07] tableView (null)  
Artists[7123:c07] searchBar (null)
```

Because `tableView` and `searchBar` are weak properties, the `UITableView` and `UISearchBar` objects are only owned by the view hierarchy. As soon as the main view gets unloaded, it releases all its subviews. The view controller doesn't hold on to these views with strong pointers, so they get deleted before `didReceiveMemoryWarning` is invoked.

Note: This means there is a subtle difference between how iOS 5 and 6 handle these sorts of situations. On iOS 5 it is possible that your `viewDidLoad` method gets invoked for a second time when a previously unloaded view controller becomes visible again. If you want your app to behave the same way on iOS 6, then change the `didReceiveMemoryWarning` method to:

```
- (void)didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
    _soundEffect = nil;  
  
    if ([self isViewLoaded] && self.view.window == nil)  
    {  
        NSLog(@"forcing my view to unload");  
        self.view = nil;  
    }  
}
```

Unsafe_unretained

We're almost done covering the basics of ARC – I just wanted to mention one more thing you should know.

Besides `strong` and `weak` there is another new modifier, `unsafe_unretained`. You typically don't want to use that. The compiler will add no automated retains or releases for variables or properties that are declared as `unsafe_unretained`.

The reason this new modifier has the word “unsafe” in its name is that it can point to an object that no longer exists. If you try to use such a pointer it’s very likely your app will crash. This is the sort of thing you used the NSZombieEnabled debugging tool to find.

Technically speaking, if you don’t use any `unsafe_unretained` properties or variables, you can never send messages to deallocated objects anymore.

Most of the time you want to use `strong`, sometimes `weak`, and almost never `unsafe_unretained`. The reason `unsafe_unretained` still exists is for compatibility with iOS 4, where the weak pointer system is not available, and for a few other tricks.

Let’s see how this works. Change the properties in **MainViewController.h**:

```
@property (nonatomic, unsafe_unretained) IBOutlet UITableView *tableView;
@property (nonatomic, unsafe_unretained) IBOutlet UISearchBar *searchBar;
```

Make sure the `didReceiveMemoryWarning` method still has an `NSLog()` at the bottom:

```
- (void) didReceiveMemoryWarning
{
    ...
    NSLog(@"%@", self.tableView);
    NSLog(@"%@", self.searchBar);
}
```

Run the app and simulate the low-memory warning. You should try this on the iOS 5.x simulator for maximum effect:

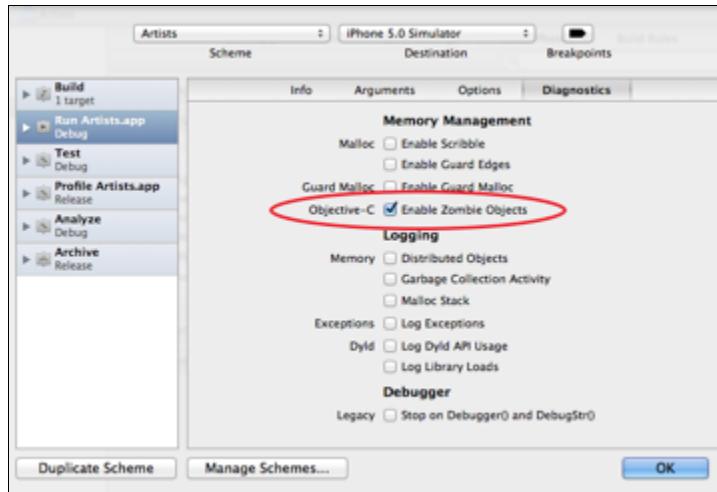
```
Artists[982:207] Received memory warning.
Artists[982:207] *** -[UITableView retain]: message sent to deallocated
instance 0x7033200
```

Whoops, the app crashes. An `unsafe_unretained` pointer does not have ownership over the object it points to. That means the `UITableView` was not kept alive by this pointer and it got deallocated before `didReceiveMemoryWarning` was called (its only owner was the main view). If this were a true weak pointer, then its value would have been set to `nil`. Remember, that was the cool feature of “zeroing” weak pointers. You saw that earlier when the `NSLog()` said “`(null)`”.

However, unlike a true weak pointer, an `unsafe_unretained` pointer is not reset to `nil` when the associated object dies. It keeps its old value. When you try to send a message to the object – which is what happens when you `NSLog()` it – you’re sending the message to an object that no longer exists.

Sometimes this may accidentally work, if the memory for that object hasn't been overwritten yet by another object, but often it will crash your app... which is exactly what you saw happening here. That should illustrate why these things are called "unsafe".

Note: This bug got caught because the active scheme for this project has enabled the Zombie Objects debug tool in the Diagnostics tab. To see this settings panel, choose **Product>Edit Scheme...** from the menu bar.



Without this setting, the app may not have crashed at all, or it may have crashed at some later point. Good luck trying to figure that one out! Those are tricky bugs to fix.

By the way, this is probably a good point to return the properties to weak:

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

For ARC-enabled apps the Enable Zombie Objects setting (also known as NSZombieEnabled) isn't terribly useful anymore, so you can disable it... except when you're using `unsafe_unretained` pointers!

If it is so harmful then why use `unsafe_unretained` in the first place? A big reason is compatibility with iOS 4.

Using ARC on iOS 4

Because ARC is largely a new feature of the LLVM compiler and not specifically of iOS 5, you can also use it on iOS 4.0 and up. The only part of ARC that does require iOS 5 is the weak pointer system. That means if you wish to deploy your ARC app on iOS 4, you cannot use `weak` properties or `__weak` variables.

You don't need to do anything special to make your ARC project work on iOS 4. If you choose a version of iOS 4 as your Deployment Target, then the compiler will automatically insert a compatibility library ("ARCLite") into your project that makes the ARC functionality available on iOS 4. That's it, just pick iOS 4.x as the Deployment Target and you're done.

If you use weak references anywhere in your code, the compiler will give the following error:

```
"Error: the current deployment target does not support automated  
_weak references"
```

You cannot use `weak` or `_weak` on iOS 4, so replace weak properties with `unsafe_unretained` and `_weak` variables with `_unsafe_unretained`. Remember that these variables aren't set to `nil` when the referenced object is deallocated, so if you're not careful your variables may be pointing at objects that no longer exist. Be sure to test your app with `NSZombieEnabled`!

Tip: The open source `PLWeakCompatibility` library is a cool hack that you can use to have your cake and eat it too! This library adds `_weak` support to iOS 4. Check it out at <https://github.com/plausiblelabs/PLWeakCompatibility>

Where To Go From Here?

Congratulations, you've covered the basics of ARC and are ready to start using it in your own new apps – and you know how to port your old ones!

If you want to learn more about ARC, stay tuned for the next chapter, where we'll cover:

- **Using blocks with ARC.** The rules for using blocks have changed a little. You need to take special care to avoid ownership cycles, the only memory problem that even ARC cannot take care of automatically.
- **How to make singletons with ARC.** You can no longer override `retain` and `release` to ensure your singleton classes can have only one instance, so how do you make singletons work with ARC?
- **More about autorelease.** All about `autorelease` and the autorelease pool.
- **Making games with ARC and Cocos2D.** Also explains how ARC fits in with Objective-C++, which you need to know if your game uses the Box2D physics engine.
- **Static libraries.** How to make your own static library to keep the ARC and non-ARC parts of your project separate.



Chapter 3: Intermediate ARC

By Matthijs Hollemans

At this point, you are pretty familiar with ARC and how to use it in new and existing projects.

However, ARC is such an important new aspect of iOS 5 development that there are several more aspects of ARC development that I thought you should know about.

So in this chapter, you're going to continue your investigations of ARC. We'll start by discussing how ARC and Blocks work together, continue with a discussion of autorelease and singletons, cover how to use ARC with the ever-popular Cocos2D and Box2D game frameworks, and end with a section on making your own static libraries.

This chapter continues where we left it off in the Beginning ARC chapter – you'll still be working with the Artists project. So open it in Xcode if you haven't already, and let's get started!

Blocks

Blocks and ARC go well together. In fact, ARC makes using blocks even easier than before. As you may know, blocks are initially created on the stack. If you wanted to keep a block alive beyond the current scope you had to copy it to the heap with `[copy]` or `Block_copy()` functions. ARC now takes care of that automatically.

However, a few things are different with blocks and ARC too and we will go over those differences in this section.

You're going to add a new view to the Detail screen called `AnimatedView`. This is a `UIView` subclass that redraws itself several times per second. The drawing instructions are provided by a block and can be whatever you want. This makes it easy to create new animations without having to subclass the `AnimatedView` object.

Add a new **Objective-C class** file to the project named `AnimatedView`, subclass of `UIView`. Then replace the contents of `AnimatedView.h` with the following:

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

typedef void (^AnimatedViewBlock)(CGContextRef context, CGRect rect, CFTimeInterval totalTime, CFTimeInterval deltaTime);

@interface AnimatedView : UIView

@property (nonatomic, copy) AnimatedViewBlock block;

@end
```

The class has a single property, `block`. This is an Objective-C block that takes four parameters: `context`, `rect`, `totalTime` and `deltaTime`. The `context` and `rect` are for drawing, while the two time parameters can be used to determine by how big a step the animation should proceed.

Replace the contents of **AnimatedView.m** with the following:

```
#import "AnimatedView.h"

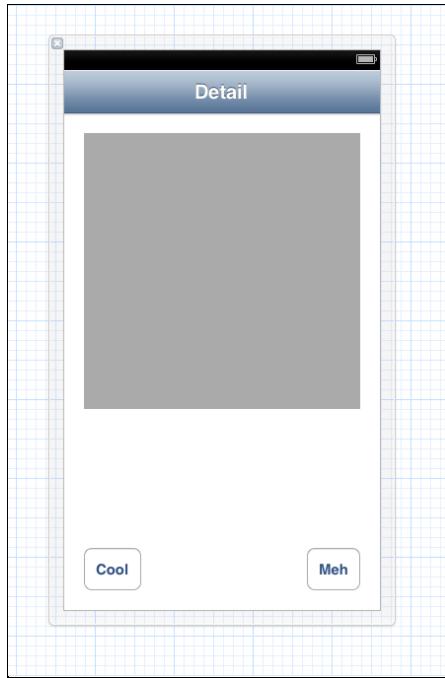
@implementation AnimatedView

- (void)dealloc
{
    NSLog(@"dealloc AnimatedView");
}

@end
```

The implementation doesn't do much yet. First you will hook up this class to the nib.

Open **DetailViewController.xib** and drag a new View object (a plain white view) into the view controller. Change the background color of the new view to Light Gray Color so you can actually see what you're doing, and set its dimensions to 280 by 280 points. The new layout should look something like this:



Select the new view and in the Identity inspector set its Class to **AnimatedView**. Now the `DetailViewController` will instantiate your view subclass for this view.

Add an outlet property for the view in **DetailViewController.h**. This should be a `weak` property because it is an outlet for a subview. Also add a forward declaration so the compiler knows that `AnimatedView` is an object:

```
@class AnimatedView;  
  
...  
  
@property (nonatomic, weak) IBOutlet AnimatedView *animatedView;
```

In **DetailViewController.m**, import the header:

```
#import "AnimatedView.h"
```

The final step is to go into Interface Builder and connect the view from the nib to this new outlet property. After you've done that, build and run the app to make sure everything still works.

When you close the Detail screen, the Xcode Debug pane should also say:

```
Artists[1389:207] deallocate AnimatedView
```

The `NSLog()` in `dealloc` is there to verify that the `AnimatedView` is truly deallocated when you're done with it. Even though ARC makes it nearly impossible for your apps to crash because you forgot a `retain` or over-released an object, you still

need to be careful about memory leaks. Objects stay alive as long as they are being pointed to, and as you will soon see, sometimes those strong pointers aren't immediately obvious.

Let's make the view actually do something. `AnimatedView` will get a timer. Every time the timer fires it asks the view to redraw itself and inside the `drawRect:` method the view invokes the block to do the actual drawing.

These are the changes to **AnimatedView.m**:

```
#import <QuartzCore/QuartzCore.h>
#import "AnimatedView.h"

@implementation AnimatedView
{
    NSTimer *_timer;
    CFTimeInterval _startTime;
    CFTimeInterval _lastTime;
}

- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        _timer = [NSTimer scheduledTimerWithTimeInterval:0.1
                                                 target:self
                                               selector:@selector(handleTimer:)
                                             userInfo:nil
                                              repeats:YES];

        _startTime = _lastTime = CACurrentMediaTime();
    }
    return self;
}

- (void)dealloc
{
    NSLog(@"dealloc AnimatedView");
    [_timer invalidate];
}

- (void)handleTimer:(NSTimer*)timer
{
    [self setNeedsDisplay];
}

- (void)drawRect:(CGRect)rect
```

```
{
    CFTimeInterval now = CACurrentMediaTime();
    CFTimeInterval totalTime = now - _startTime;
    CFTimeInterval deltaTime = now - _lastTime;
    _lastTime = now;

    if (self.block != nil)
        self.block(UIGraphicsGetCurrentContext(), rect,
                   totalTime, deltaTime);
}

@end
```

The various “time” instance variables simply keep track of how much time has passed since the view was created and since the previous call to `handleTimer`. It’s handy to know both how long the animation has been running and how much time has elapsed since the previous frame.

This implementation seems to make sense – you create the timer in `initWithCoder:` and stop the timer in `dealloc` – but already you’re dealing with an ownership cycle. Run the app and close the Detail screen. Notice that the `dealloc` method from `AnimatedView` isn’t called anymore! There is no `NSLog()` message in the output pane.

`NSTimer` apparently holds a strong reference to its target, which happens to be the `AnimatedView` object itself. So `AnimatedView` has a strong reference to `NSTimer` and `NSTimer` has a strong reference back to `AnimatedView`. Unless you explicitly release one of these objects, they will keep each other alive indefinitely.

You can break this particular retain cycle by adding a `stopAnimation` method. Change `dealloc` to the following and add `stopAnimation`:

```
- (void)stopAnimation
{
    [_timer invalidate], _timer = nil;
}

- (void)dealloc
{
    NSLog(@"dealloc AnimatedView");
}
```

Also add the method signature for the new method to the `AnimatedView.h` header:

```
- (void)stopAnimation;
```

Before the `AnimatedView` object gets released, the user of this class should call `stopAnimation`. In this app that becomes the responsibility of `DetailViewController`. Change the `dealloc` method from **DetailViewController.m** to the following:

```
- (void)dealloc
{
    NSLog(@"dealloc DetailViewController");
    [self.animatedView stopAnimation];
}
```

If you now run the app again, you'll see the `AnimatedView` does properly get deallocated when the Detail screen closes.

No doubt there are other ways to solve this ownership cycle. However, simply making the `_timer` instance variable weak doesn't work:

```
@implementation AnimatedView
{
    __weak NSTimer *_timer;
    CFTimeInterval _startTime;
    CFTimeInterval _lastTime;
}
```

This causes `AnimatedView` no longer to be the owner of the `NSTimer` object. But that won't do you any good. The `NSTimer` is still owned by another object, the run loop, and because the timer has a strong reference back to `AnimatedView`, it will keep `AnimatedView` existing forever. The timer does not release its target until you invalidate it.

Let's make the view do some drawing. You're not going to make it animate just yet, it will simply draw the same thing – the name of the selected artist – over and over. In **DetailViewController.m**, change `viewDidLoad` to create the block with the drawing code and assign it to the `AnimatedView`:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationBar.topItem.title = self.artistName;

    UIFont *font = [UIFont boldSystemFontOfSize:24.0f];
    CGSize textSize = [self.artistName sizeWithFont:font];

    self.animatedView.block = ^(CGContextRef context, CGRect rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
    {
        NSLog(@"totalTime %f, deltaTime %f", totalTime,
              deltaTime);
    };
}
```

```
    CGPoint textPoint = CGPointMake(
        (rect.size.width - textSize.width)/2,
        (rect.size.height - textSize.height)/2);

    [self.artistName drawAtPoint:textPoint withFont:font];
}
}
```

Outside the block you create a `UIFont` object and calculate how big the text will be when drawn. Inside the block you use the value from that `textSize` variable to center the text in the rectangle and then draw it. The `NSLog()` is there just to show you that this block is called every couple of milliseconds.

This looks innocuous enough but when you run the app you'll notice something is missing from the Debug output: not only will `AnimatedView` no longer be deallocated, neither will `DetailViewController`!

If you've used blocks before then you know the block captures the value of every variable that you use inside the block. If those variables are pointers, the block retains the objects they point to. That means the block retains `self`, i.e. the `DetailViewController`, because `self.artistName` is accessed inside the block.

As a result, `DetailViewController` will never be deallocated even after it gets closed because the block keeps holding on to it. The timer also keeps running because the app never got around to calling `stopAnimation`, although you don't see that in the Debug pane (because the view doesn't redraw anymore).

There are a few possible solutions to this problem. One is to not use `self` inside the block. That means you cannot access any properties, instance variables, or methods from the block. Local variables are fine. The reason you cannot use instance variables is that this does `self->variable` behind the scenes and therefore still refers to `self`.

For example, you could capture the artist name into a local variable and use that inside the block instead:

```
NSString *text = self.artistName;

self.animatedView.block = ^(CGContextRef context, CGRect
    rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    CGPoint textPoint = CGPointMake(
        (rect.size.width - textSize.width)/2,
        (rect.size.height - textSize.height)/2);

    [text drawAtPoint:textPoint withFont:font];
}
};
```

This will work. All the values captured by the block are now local variables. Nothing refers to `self` and therefore the block will not capture a pointer to the `DetailViewController`. Run the app and notice that everything gets dealloc'd just fine.

Sometimes you can't avoid referring to `self` in the block. Before ARC, you could use the following trick:

```
__block DetailViewController *blockSelf = self;

self(animatedView.block = ^(CGContextRef context, CGRect
    rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    . . .

    [blockSelf.artistName drawAtPoint:textPoint
        withFont:font];
}
```

The block did not retain any variables prefixed with the `__block` keyword. Therefore, `blockSelf.artistName` could be used to access the `artistName` property without the block capturing the true `self` object.

Alas, this no longer works with ARC. Variables are strong by default, even if they are marked as `__block` variables. The only function of `__block` is to allow you to change captured variables (without `__block`, they are read-only).

The solution is to use a `__weak` variable instead:

```
__weak DetailViewController *weakSelf = self;

self(animatedView.block = ^(CGContextRef context, CGRect
    rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    DetailViewController *strongSelf = weakSelf;
    if (strongSelf != nil)
    {
        NSLog(@"totalTime %f, deltaTime %f", totalTime,
            deltaTime);
        CGPoint textPoint = CGPointMake(
            (rect.size.width - textSize.width)/2,
            (rect.size.height - textSize.height)/2);

        [strongSelf.artistName drawAtPoint:textPoint
            withFont:font];
    }
};
```

The `weakSelf` variable refers to `self` but does not retain it. You let the block capture `weakSelf` instead of `self`, so there is no ownership cycle. However, you shouldn't actually use `weakSelf` inside the block. Because this is a weak pointer, it will become `nil` when `DetailViewController` is deallocated.

While you are allowed to send messages to `nil` in Objective-C, it's still a good idea to check inside the block whether the object is still alive. Even better, by assigning the weak pointer to a new variable, `strongSelf`, you temporarily turn it into a strong reference so that the object cannot be destroyed out from under you while you're using it.

For the Artists app this is probably a bit of overkill. You could have simply used `weakSelf` and everything would have worked fine. After all, it is impossible for the `DetailViewController` to be deallocated before the `AnimatedView`, because `AnimatedView` is part of the controller's view hierarchy.

However, this may not be true in other situations. For example, if the block is used asynchronously then creating a strong reference to keep the object in question alive is a good idea. Additionally, if you were using the `_artistName` instance variable directly, you might have written something like this:

```
__weak DetailViewController *weakSelf = self;

self(animatedView.block = ^(CGContextRef context, CGRect
    rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    . .
    [weakSelf->_artistName drawAtPoint:textPoint
        withFont:font];
}
```

This works fine until the `DetailViewController` is deallocated and `weakSelf` becomes `nil`. Sending messages to `nil` works fine, including accessing `nil` properties, but doing `nil->artistName` will definitely crash your app!

Therefore, if you're using blocks with ARC and you want to avoid capturing `self`, the following pattern is recommended:

```
__weak id weakSelf = self;
block = ^{
{
    id strongSelf = weakSelf;
    if (strongSelf != nil)
    {
        // do stuff with strongSelf
    }
};
```

If you're targeting iOS 4, you cannot use `__weak`. Instead do:

```
__block __unsafe_unretained id unsafeSelf = self;
```

Note that in this case you will never know if `unsafeSelf` still points to a valid object because unlike `__weak` pointers, `__unsafe_unretained` pointers do not automatically get set to nil (that's why they are "unsafe"). You will need to take additional steps to make sure the pointer is still valid before you dereference it or face the zombies!

Note: Even though ARC takes care to copy blocks to the heap when you assign them to instance variables or return them, there are still a few situations where you need to copy blocks manually. In the `@property` declaration for the block you specified the ownership qualifier `copy` instead of `strong`. If you try it with `strong`, the app might crash when you open the Detail screen. You may run into other situations in your apps where just passing a block crashes the whole thing. In that case, see if `[block copy]` makes any difference.

On the whole, writing ARC code is exactly the same as what you did before except now you don't call `retain` and `release`. But this can also introduce subtle bugs that were not there before, because removing the call to `retain`, `release` or `autorelease` can mean the block no longer captures an object and therefore the object may not stay alive for as long as you think it does.

Take an imaginary class, `DelayedOperation`. It waits for "delay" number of seconds and then executes the block. Inside the block you could previously call `autorelease` to free the `DelayedOperation` instance. Because the block captures the `operation` instance and thereby keeps it alive, this pattern worked without problems before ARC.

```
DelayedOperation *operation = [[DelayedOperation alloc]
    initWithDelay:5 block:^{
        NSLog(@"Performing operation");

        // do stuff

        [operation autorelease];
    }];
}
```

However, with ARC you can no longer call `autorelease` and the code becomes:

```
DelayedOperation *operation = [[DelayedOperation alloc]
    initWithDelay:5 block:^{
    };
```

```
    NSLog(@"Performing operation");

    // do stuff
}];
```

Guess what, the block will never execute. The `DelayedOperation` instance is destroyed as soon as it is created because there is nothing holding on to it. Converting this to ARC has actually introduced a bug! Very sneaky...

One way to fix this is to capture the `operation` instance and set it to nil when you're done:

```
__block DelayedOperation *operation = [[DelayedOperation alloc]
    initWithDelay:5 block:^{
{
    NSLog(@"Performing operation");

    // do stuff

    operation = nil;
}};
```

Now the block keeps the object alive again. Notice that the `operation` variable must be declared as `__block` because you're changing its value inside the block.

Singletons

If your apps use singletons, their implementation may have these methods:

```
+ (id)allocWithZone:(NSZone *)zone
{
    return [[self sharedInstance] retain];
}

- (id)copyWithZone:(NSZone *)zone
{
    return self;
}

- (id)retain
{
    return self;
}

- (NSUInteger)retainCount
```

```
{  
    return NSUIntegerMax;  
}  
  
- (oneway void)release  
{  
    // empty  
}  
  
- (id)autorelease  
{  
    return self;  
}
```

In this common singleton recipe, methods such as `retain` and `release` are overridden so that it is impossible to make more than one instance of this object. After all, that's what a singleton is, an object that can have no more than a single instance.

With ARC this will no longer work. Not only can you not call `retain` and `release`, but also you're not allowed to override these methods.

In my opinion, the above is not a very useful pattern anyway. How often does it happen that you truly want only a single instance of an object? It's easier to use a variation of the singleton pattern that I've heard someone call the "Interesting Instance Pattern". That is what Apple uses in their APIs as well. You typically access this preferred instance through a `sharedInstance` or `defaultInstance` class method, but if you wanted to you could make your own instances as well.

Whether that is a good thing or not for your own singletons can be documented or made a matter of convention. For certain singleton classes from the iOS API, having the ability to make your own instances is actually a feature, such as with `NSNotificationCenter`.

To demonstrate the preferred way of making singletons, you're going to add one to our app. Add a new **NSObject subclass** to the project and name it **GradientFactory**.

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

```
@interface GradientFactory : NSObject  
  
+ (id)sharedInstance;  
  
- (CGGradientRef)newGradientWithColor1:(UIColor *)color1  
                           color2:(UIColor *)color2  
                           color3:(UIColor *)color3  
                           midpoint:(CGFloat)midpoint;
```

```
@end
```

This class has a `sharedInstance` class method that is to be used to access it, and a `newGradient` method that returns a `CGGradientRef` object. You could still `[[alloc] init]` your own instance of `GradientFactory`, but convention says you shouldn't.

In **GradientFactory.m**, add the implementation of the `sharedInstance` method:

```
#import "GradientFactory.h"

@implementation GradientFactory

+ (id)sharedInstance
{
    static GradientFactory *sharedInstance;
    if (sharedInstance == nil)
    {
        sharedInstance = [[GradientFactory alloc] init];
    }
    return sharedInstance;
}
```

This is really all you need to do to make a singleton. The `sharedInstance` method uses a static local variable to track whether an instance already exists. If not, it makes one.

Note that you don't have to explicitly set the variable to `nil`:

```
static GradientFactory *sharedInstance = nil;
```

With ARC, all pointer variables are `nil` by default. Before ARC this was only true for instance variables and statics, not local variables. If you did something like this,

```
- (void)myMethod
{
    int someNumber;
    NSLog(@"Number: %d", someNumber);

    NSString *someString;
    NSLog(@"String: %p", someString);
}
```

then Xcode complained – “Variable is uninitialized when used here” – and the output would be random numbers:

```
Number: 67
String: 0x4bab5
```

With ARC, however, the output is:

```
Number: 10120117
String: 0x0
```

The `int` still contains some garbage value (and using it in this fashion issues a compiler warning) but the initial value of `someString` is `nil`. That's great because now it has become nearly impossible to use a pointer that doesn't point at a valid object.

Let's finish the implementation of `GradientFactory`. Add the following method:

```
- (CGGradientRef)newGradientWithColor1:(UIColor *)color1
                                color2:(UIColor *)color2
                                color3:(UIColor *)color3
                           midpoint:(CGFloat)midpoint
{
    NSArray *colors = @[(id)color1.CGColor, (id)color2.CGColor,
                        (id)color3.CGColor];

    const CGFloat locations[3] = { 0.0f, midpoint, 1.0f };

    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    CGGradientRef gradient = CGGradientCreateWithColors(
        colorSpace, (__bridge CFArrayRef)colors, locations);
    CGColorSpaceRelease(colorSpace);

    return gradient;
}
```

This creates a gradient with three colors, two on the outer edges of the gradient and one in the middle. The position of the midpoint is flexible, and you'll use that to create a simple animation later.

The `CGGradientCreateWithColors()` function from Core Graphics is used to construct the gradient. This takes a pointer to a `cCGColorSpaceRef` object, an array of `cCGColorRef` objects, and an array of `CGFloats`. The `colors` array is a `CFArrayRef` object, but the `locations` array is a straight C-array. Creating the `colorSpace` object and the array of locations is pretty straightforward.

However, for the `colors` array it's much nicer to use an `NSArray` rather than a `CFArrayRef`. Thanks to toll-free bridging, you can write:

```
NSArray *colors = @[ (id)color1.CGColor, (id)color2.CGColor,
                     (id)color3.CGColor ];
```

The colors array should contain `cgcgcolorRef` objects. The parameters to the `newGradientWithColor...` method are `UIColor` objects, so first you have to convert them. `UIColor` and `cgcgcolorRef` are NOT toll-free bridged, so you cannot simply cast them. Instead, `UIColor` has a `.CGColor` property that returns a `cgcgcolorRef` object.

Because `NSArray` can only hold Objective-C objects, not Core Foundation objects, you have to cast that `cgcgcolorRef` back to an `id`. That works because all Core Foundation object types are toll-free bridged with `NSObject`, but only in regards to memory handling. So you can treat a `CGColor` as an `NSObject`, but not as a `UIColor`.

This might be terribly confusing, but that's what you get when you mix two different types of framework architectures.

Of course, the `CGGradientCreateWithColors()` function doesn't accept an `NSArray` so you need to cast colors to a `CFArraryRef` to do toll-free bridging the other way around. This time, however, a bridged cast is necessary. The compiler can't figure out by itself whether ownership of the object should change or not. In this case you still want to keep ARC responsible for releasing the `NSArray` object, so a simple `__bridge` cast serves to indicate that no transfer of ownership is taking place.

```
CGGradientRef gradient = CGGradientCreateWithColors(
    colorSpace, (__bridge CFArraryRef)colors, locations);
```

No `__bridge` statement was necessary when you casted the `cgcgcolorRef` objects to `id` earlier. The compiler was able to figure out by itself what the proper rules were. (It will automatically retain the color objects for as long as the array exists, just like any object that you put into an `NSArray`.)

Finally, the app returns the new `CGGradientRef` object. Note that the caller of this method is responsible for releasing this gradient object. It is a Core Foundation object and therefore is not handled by ARC. Whoever calls the `newGradient` method is responsible for calling `CGGradientRelease()` on the gradient, or the app will leak memory. (And a lot of it too as you will be calling this method from the animation block that runs several times per second.)

The `newGradient` method has the word "new" in its name. That is not for nothing. The Cocoa naming rules say that methods whose name starts with `alloc`, `init`, `new`, `copy` or `mutableCopy` transfer ownership of the returned object to the caller.

To be honest, if your entire code base uses ARC then these naming conventions aren't important at all. The compiler will do the right thing anyway. But it's still a good idea to let the users of your classes and methods know that you expect them to release the objects manually if you're returning Core Foundation objects or `malloc()`'d buffers.

Note: I still urge you to respect the Cocoa naming conventions, even though they are inconsequential when your entire project compiles as ARC. The names of methods are still important when *not* every file in your project is ARC, such as in this example project. For non-ARC code and ARC code to properly interoperate, sticking to the Cocoa naming conventions is essential.

Suppose you have a method in a non-ARC file that is named `newWidget` and it returns an autoreleased string rather than a retained one. If you use that method from ARC code then ARC will try to release the returned object and your app will crash on an over-release.

It's better to rename that method to `createWidget` or `makeWidget` so that ARC knows there is no need to release anything. (Alternatively, if you can't change the name, use the `NS_RETURNS_NOT_RETAINED` or `NS_RETURNS_RETAINED` annotations to tell the compiler about the non-standard behavior of these methods.)

You have to be really careful with mixing Core Foundation and Objective-C objects. Spot the bug in the following code snippet:

```
CGColorRef cgColor1 = [[UIColor alloc] initWithRed:1 green:0
                                         blue:0 alpha:1].CGColor;
CGColorRef cgColor2 = [[UIColor alloc] initWithRed:0 green:1
                                         blue:0 alpha:1].CGColor;
CGColorRef cgColor3 = [[UIColor alloc] initWithRed:0 green:0
                                         blue:1 alpha:1].CGColor;

NSArray *colors = @[ (__bridge id)cgColor1,
                     (__bridge id)cgColor2,
                     (__bridge id)cgColor3 ];
```

If you do this in your app it will crash. The reason is simple, you create a `UIColor` object that is not autoreleased but retained (because you call `alloc + init`). As soon as there are no strong pointers to this object, it gets deallocated. Because a `CGColorRef` is not an Objective-C object, the `cgColor1` variable does not qualify as a strong pointer. The new `UIColor` object is immediately released after it is created and `cgColor1` points at garbage memory. Yikes!

You can't solve it like this either:

```
CGColorRef cgColor1 = [UIColor colorWithRed:1 green:0 blue:0
                                         alpha:1].CGColor;
```

Because you now allocate the `UIColor` object using a method that appears to return an autoreleased object, you would think it stays alive until the autorelease pool is

flushed. But there is no guarantee that the compiler actually gives you an autoreleased object, so you shouldn't depend on this!

Note: The “getting a `CGColor` from a `UIColor` crashes my app” problem is probably the number one most common issue that people run into when switching an existing codebase to ARC. What’s worse, the above code snippet appears to work fine on the simulator but crashes when run on a device!

The safest solution is to keep the `UIColor` object alive using a strong pointer:

```
UIColor *color1 = [UIColor colorWithRed:1 green:0 blue:0 alpha:1];  
  
. . . and use it later as color1.CGColor
```

If you don’t want to worry about these weird situations, then mix and match as little with Core Foundation as possible. :-)

Enough theory. Let’s put this `GradientFactory` to work in a silly animation example. Go to **DetailViewController.m** and add an import:

```
#import "GradientFactory.h"
```

Change `viewDidLoad` to:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
    self.navigationBar.topItem.title = self.artistName;  
  
    UIFont *font = [UIFont boldSystemFontOfSize:24.0f];  
    CGSize textSize = [self.artistName sizeWithFont:font];  
  
    float components[9];  
   NSUInteger length = [self.artistName length];  
    NSString* lowercase = [self.artistName lowercaseString];  
  
    for (int t = 0; t < 9; ++t)  
    {  
        unichar c = [lowercase characterAtIndex:t % length];  
        components[t] = ((c * (10 - t)) & 0xFF) / 255.0f;  
    }  
  
    UIColor *color1 = [UIColor colorWithRed:components[0]  
                                    green:components[3] blue:components[6] alpha:1.0f];
```

```
UIColor *color2 = [UIColor colorWithRed:components[1]
                                    green:components[4] blue:components[7] alpha:1.0f];
UIColor *color3 = [UIColor colorWithRed:components[2]
                                    green:components[5] blue:components[8] alpha:1.0f];

__weak DetailViewController *weakSelf = self;

self.animatedView.block = ^(CGContextRef context, CGRect
    rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    DetailViewController *strongSelf = weakSelf;
    if (strongSelf != nil)
    {
        CGPoint startPoint = CGPointMake(0.0, 0.0);
        CGPoint endPoint = CGPointMake(0.0,
                                       rect.size.height);
        CGFloat midpoint = 0.5f + (sinf(totalTime))/2.0f;

        CGGradientRef gradient =
            [[GradientFactory sharedInstance]
                newGradientWithColor1:color1 color2:color2
                color3:color3 midpoint:midpoint];

        CGContextDrawLinearGradient(context, gradient,
            startPoint, endPoint,
            kCGGradientDrawsBeforeStartLocation |
            kCGGradientDrawsAfterEndLocation);

        CGGradientRelease(gradient);

        CGPoint textPoint = CGPointMake(
            (rect.size.width - textSize.width)/2,
            (rect.size.height - textSize.height)/2);

        [strongSelf.artistName drawAtPoint:textPoint
            withFont:font];
    }
};

}
```

What happens here may look complicated but basically it takes the name of the artist and uses that to derive three colors. Each color contains three components (red, green, blue) so that makes nine color components in total. First it loops through the name of the artist, converted to lowercase, and transforms each character into a value between 0.0f and 1.0f using a basic formula:

```

float components[9];
NSUInteger length = [self.artistName length];
NSString* lowercase = [self.artistName lowercaseString];

for (int t = 0; t < 9; ++t)
{
    unichar c = [lowercase characterAtIndex:t % length];
    components[t] = ((c * (10 - t)) & 0xFF) / 255.0f;
}

```

It then turns these color components into `UIColor` objects:

```

UIColor *color1 = [UIColor colorWithRed:components[0]
                                 green:components[3] blue:components[6] alpha:1.0f];
UIColor *color2 = [UIColor colorWithRed:components[1]
                                 green:components[4] blue:components[7] alpha:1.0f];
UIColor *color3 = [UIColor colorWithRed:components[2]
                                 green:components[5] blue:components[8] alpha:1.0f];

```

This can all be done outside of the block because it only needs to happen once. Inside the block it does the trick with `weakSelf` and `strongSelf` again and then calculates the start and end points for the gradient:

```

CGPoint startPoint = CGPointMake(0.0, 0.0);
CGPoint endPoint = CGPointMake(0.0,
                               rect.size.height);
CGFloat midpoint = 0.5f + (sinf(totalTime))/2.0f;

```

The midpoint moves up and down between the start and end points. The sine function is used to ease the animation in and out.

Now that the colors and the midpoint position are calculated, the new gradient object can be created:

```

CGGradientRef gradient =
[[GradientFactory sharedInstance]
    newGradientWithColor1:color1 color2:color2
    color3:color3 midpoint:midpoint];

```

The code accesses the `GradientFactory` object through the `sharedInstance` class method and then calls `newGradient`. Thanks to the singleton pattern, the very first time it does this a new instance of `GradientFactory` is created, but for every time after that the app simply re-uses that same instance.

Then a Core Graphics function draws that gradient between the start and end points:

```
CGContextDrawLinearGradient(context, gradient,
    startPoint, endPoint,
    kCGGradientDrawsBeforeStartLocation |  
    kCGGradientDrawsAfterEndLocation);
```

And finally, the gradient object is released:

```
CGGradientRelease(gradient);
```

Remember that this release is necessary because ARC does not concern itself with Core Foundation objects, only Objective-C objects. You still need to do manual memory management when you're dealing with Core Foundation!

Here is an example of what the animation looks like. Every artist has its own colors:



If your singleton can be used from multiple threads then the simple `sharedInstance` accessor method does not suffice. A more sturdy implementation is this:

```
+ (id)sharedInstance  
{  
    static GradientFactory *sharedInstance;  
    static dispatch_once_t done;  
  
    dispatch_once(&done, ^  
    {  
        sharedInstance = [[GradientFactory alloc] init];  
    });  
  
    return sharedInstance;  
}
```

```
}
```

Replace `sharedInstance` from **GradientFactory.m** with this method. This uses the `dispatch_once()` function from the Grand Central Dispatch library to ensure that `alloc` and `init` are truly executed only once, even if multiple threads at a time try to perform this block.

And that's it for singletons!

Autorelease

You have already seen that autorelease and the autorelease pool are still used with ARC, although it's now a language construct rather than a class (`@autoreleasepool`).

Methods basically always return an autoreleased object, except when the name of the method begins with `alloc`, `init`, `new`, `copy` or `mutableCopy`, in which case they return a retained object. That's still the same as it was with manual memory management. (It needs to be because ARC code needs to be able to play nice with non-ARC code.)

A retained object is deallocated as soon as there are no more variables pointing to it, but an autoreleased object is only deallocated when the autorelease pool is drained. Previously you had to call `[drain]` (or `release`) on the `NSAutoreleasePool` object, but now the pool is automatically drained at the end of the `@autoreleasepool` block:

```
@autoreleasepool
{
    NSString *s = [NSString stringWithFormat:...];
}

// the string object is deallocated when the code gets here
```

Contrast the above with this:

```
NSString *s;

@autoreleasepool
{
    s = [NSString stringWithFormat:...];
}

// the string object is still alive here
```

Even though the `NSString` object was created inside the `@autoreleasepool` block and was returned autoreleased from the `stringWithFormat:` method (it doesn't have

`alloc`, `init` or `new` in its name), you store the string object in local variable `s`, and that is a strong pointer. As long as `s` is in scope, the string object stays alive.

There is a way to make the string object in the previous example be deallocated by the autorelease pool:

```
__autoreleasing NSString *s;

@autoreleasepool
{
    s = [NSString stringWithFormat:...];
}

// the string object is deallocated at this point

 NSLog(@"%@", s); // crash!
```

The special `__autoreleasing` keyword tells the compiler that this variable's contents may be autoreleased. Now it no longer is a strong pointer and the string object will be deallocated at the end of the `@autoreleasepool` block. Note, however, that `s` will keep its value and now points at a dead object. If you were to send a message to `s` afterwards, the app will crash.

You will hardly ever need to use `__autoreleasing`, but you might come across it in the case of out-parameters or pass-by-reference, especially with methods that take an `(NSError **)` parameter.

Let's add one last feature to the app to demonstrate this. When the user presses the Cool button you're going to capture the contents of the `AnimatedView` and save this image to a PNG file.

In `DetailViewController.m`, add the following method:

```
- (NSString *)documentsDirectory
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);
    return [paths lastObject];
}
```

Also add an import for QuartzCore at the top:

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

And replace `coolAction:` with the following:

```
- (IBAction)coolAction
{
```

```

UIGraphicsBeginImageContext(self.animatedView.bounds.size);
[self.animatedView.layer renderInContext:
    UIGraphicsGetCurrentContext()];
UIImage *image =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();

NSData *data = UIImagePNGRepresentation(image);
if (data != nil)
{
    NSString *filename = [[self documentsDirectory]
        stringByAppendingPathComponent:@"Cool.png"];

    NSError *error;
    if (![data writeToFile:filename
        options:NSDataWritingAtomic error:&error])
    {
        NSLog(@"Error: %@", error);
    }
}

[self.delegate detailViewController:self
    didPickButtonWithIndex:0];
}

```

The revised `coolAction:` method is quite straightforward. First it captures the contents of the `AnimatedView` into a new `UIImage` object:

```

UIGraphicsBeginImageContext(self.animatedView.bounds.size);
[self.animatedView.layer renderInContext:
    UIGraphicsGetCurrentContext()];
UIImage *image =
    UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();

```

Then it turns that image into a PNG file, which gives you an `NSData` object. It also creates the output filename, which is "Cool.png" inside the app's Documents folder.

```

NSData *data = UIImagePNGRepresentation(image);
if (data != nil)
{
    NSString *filename = [[self documentsDirectory]
        stringByAppendingPathComponent:@"Cool.png"];
}

```

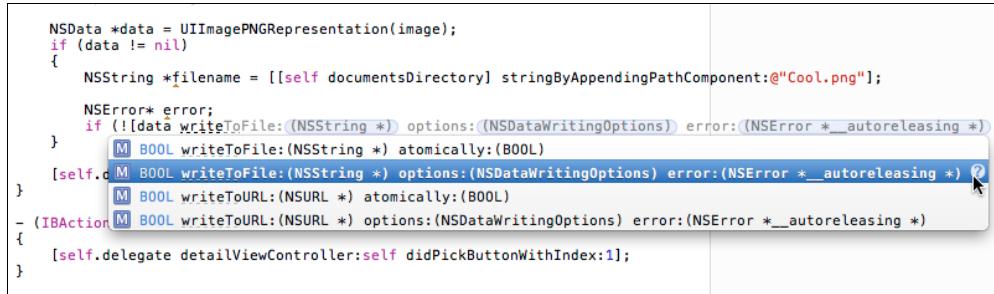
The `writeToFile` method from `NSData` accepts a pointer to a variable for an `NSError` object, in other words an `(NSError **)`. Yes, that is two stars! If there is some kind

of error, the method will create a new `NSError` object and store that into your `error` variable. This is also known as an out-parameter and it's often used to return more than one value from a method or function.

In this case `writeToFile` already returns YES or NO to indicate success or failure. Upon failure, it also returns an `NSError` object with more information in the out-parameter.

```
NSError *error;
if (![data writeToFile:filename
                     options:NSDataWritingAtomic error:&error])
{
    NSLog(@"Error: %@", error);
}
```

If you type in the code by hand you'll get an auto-complete popup. There you'll see that this `NSError **` parameter is actually specified as `NSError * __autoreleasing *`:



This is a common pattern for implementing out-parameters with ARC. It tells the compiler that the `NSError` object that is returned from `writeToFile` must be treated as an autoreleased object. Typically you don't have to worry about this. As you can see in the code above you've never actually used the `__autoreleasing` keyword anywhere. The compiler will figure this out automatically.

You only need to use `__autoreleasing` when you're writing your own method that needs to return an out-parameter, or when you have a performance problem.

When you write this:

```
NSError *error;
```

then you implicitly say that the `error` variable is `__strong`. However, if you then pass the address of that variable into the `writeToFile` method, you want to treat it as an `__autoreleasing` variable. These two statements cannot both be true; a variable is either strong or autoreleasing, not both at the same time.

To resolve this situation the compiler makes a new temporary variable. Under the hood the above code actually looks like this:

```
NSError *error;
```

```
__autoreleasing NSError *temp = error;
BOOL result = ![data writeToFile:filename
                           options:NSDataWritingAtomic error:&temp];
error = temp;

if (!result)
{
    NSLog(@"Error: %@", error);
}
```

Generally speaking this extra temporary variable is no big deal. But if you want to avoid it you can write the code as follows:

```
__autoreleasing NSError *error;
if (![data writeToFile:filename options:NSDataWritingAtomic
                           error:&error])
{
    NSLog(@"Error: %@", error);
}
```

Now the type of your local `error` variable is the same as the type of `writeToFile`'s `error` parameter and no conversion is necessary. Personally, I wouldn't use the `__autoreleasing` keyword in situations such as these. Your code will work fine without it and in most cases the above is an unnecessary optimization.

To write your own method that needs to return an out-parameter, you would do something like this:

```
- (NSString *)fetchKeyAndValue:
             (__autoreleasing NSNumber **)value
{
    NSString *theKey;
    NSString *theValue;

    // do whatever you need to do here

    *value = theValue;
    return theKey;
}
```

This returns an `NSString` object the regular way and places an `NSNumber` object in the out-parameter. You would call this method as follows:

```
NSNumber *value;
NSString *key = [self fetchKeyAndValue:&value];
```

The default ownership qualifier for out-parameters is `__autoreleasing`, by the way, so you could have written the method simply as follows:

```
- (NSString *)fetchKeyAndValue:(NSNumber **)value
{
    . . .
}
```

Note that you are not required to specify `__autoreleasing` for the out-parameter. If instead you did not want to put the object into the autorelease pool, you could declare the out-parameter `__strong`:

```
- (NSString *)fetchKeyAndValue:(__strong NSNumber **)value
{
    . . .
}
```

For ARC it doesn't really matter whether out-parameters are autoreleased or strong, it will do the right thing anyway. However, if you want to use this method from non-ARC code, the compiler expects you to do a manual `[release]` on the returned object. Forgetting to do so will result in a memory leak, so be sure to document it properly when your methods return retained objects through out-parameters!

One thing to be aware of is that some API methods can use their own autorelease pool. For example, `NSDictionary`'s `enumerateKeysAndObjectsUsingBlock:` method first sets up an autorelease pool before it calls your block. That means any autoreleased objects you create in that block will be released by `NSDictionary`'s pool. That usually is exactly what you want to have happen, except in the following situation:

```
- (void)loopThroughDictionary:(NSDictionary *)d
                      error:(NSError **)error
{
    [d enumerateKeysAndObjectsUsingBlock:^(
        id key, id obj, BOOL *stop)
    {
        // do stuff . . .

        if (there is some error && error != nil)
        {
            *error = [NSError errorWithDomain:@"MyError" code:1
                                         userInfo:nil];
        }
    }];
}
```

The `error` variable is intended to be autoreleased because it is an out-parameter. Because `enumerateKeysAndObjectsUsingBlock:` has its own autorelease pool, any new error object that you create will be deallocated long before the method returns.

To solve this problem you use a temporary strong variable to hold the `NSError` object:

```
- (void)loopThroughDictionary:(NSDictionary *)d
                      error:(NSError **)error
{
    __block NSError *temp;

    [d enumerateKeysAndObjectsUsingBlock:^(
        id key, id obj, BOOL *stop)
    {
        // do stuff . . .

        if (there is some error)
        {
            temp = [NSError errorWithDomain:@"MyError" code:1
                                         userInfo:nil];
        }
    }];

    if (error != nil)
        *error = temp;
}
```

Autoreleased objects may stick around for longer than you want. The autorelease pool is emptied after each UI event (tap on a button, etc) but if your event handler does a lot of processing – for example in a loop that creates a lot of objects – you can set up your own autorelease pool to prevent the app from running out of memory:

```
for (int i = 0; i < 1000000; i++)
{
    @autoreleasepool
    {
        NSString *s = [NSString stringWithFormat:. . .];

        // do stuff . . .
    }
}
```

In older code you sometimes see special trickery to only empty the autorelease pool every X iterations of the loop. That is because people believed that `NSAutoreleasePool` was slow (it wasn't) and that draining it on every iteration would

not be very efficient. Well, that's no longer possible. Nor is it necessary because `@autoreleasepool` is about six times faster than `NSAutoreleasePool` was, so plugging an `@autoreleasepool` block into a tight loop should not slow down your app at all.

Note: If you're creating a new thread you also still need to wrap your code in an autorelease pool using the `@autorelease` syntax. The principle hasn't changed, just the syntax.

ARC has a bunch of further optimizations for autoreleased objects as well. Most of your code consists of methods that return autoreleased objects, but often there is no need for these objects to end up in the autorelease pool. Suppose you do something like this:

```
NSString *str = [NSString stringWithFormat:...];
```

The `stringWithFormat` method returns an autoreleased object. But `str` here is a strong local variable. When `str` goes out of scope, the string object can be destroyed. There is no need to also put the string object in the autorelease pool. ARC recognizes this pattern and through some magic will now not autorelease the object at all. So not only is `@autoreleasepool` faster, a lot less objects end up in it!

Note: There is no such thing as autoreleasing a Core Foundation object. The principle of autorelease is purely an Objective-C thing. Some people have found ways to autorelease CF objects anyway by first casting them to an `id`, then calling `autorelease`, and then casting them back again:

```
return (CGImageRef)[(id)myImage autorelease];
```

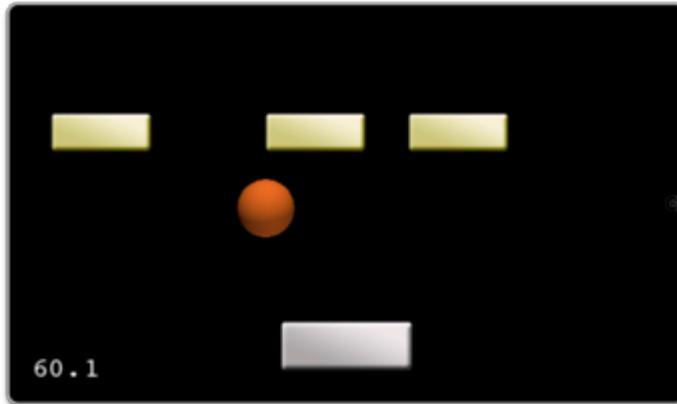
That obviously won't work anymore because you cannot call `autorelease` with ARC. If you really feel like you need to be able to autorelease Core Foundation objects, then you can make your own `CFAutorelease()` function and put it in a file that gets compiled with `-fno-objc-arc`.

Cocos2D and Box2D

So far we've just talked about ARC and UIKit-based apps. The same rules apply to Cocos2D games, of course, but because the current version of Cocos2D is not 100% compatible with ARC I will explain how to put it into your ARC-based games.

For this part of the chapter I have prepared some starter code, based on the tutorial How to Create a Simple Breakout Game with Box2D and Cocos2D from raywenderlich.com. You can find the original tutorial here:

- Part 1: <http://www.raywenderlich.com/475/how-to-create-a-simple-breakout-game-with-box2d-and-cocos2d-tutorial-part-12>
- Part 2: <http://www.raywenderlich.com/505/how-to-create-a-simple-breakout-game-with-box2d-and-cocos2d-tutorial-part-22>



The code from this project is adapted to the latest available version of Cocos2D 1.x at the time of writing (v1.1 beta 2).

Note: The previous edition of this book used the stable version 1.0.1 of Cocos2D, but that is out-of-date now and no longer compiles cleanly with the latest LLVM compiler and SDK. It also required extensive changes to the Cocos2D source itself to make it play nice with ARC.

Fortunately, with v1.1 these changes are no longer necessary and integrating Cocos2D in your ARC-enabled games is much easier. There is also a new 2.x branch that is a complete rewrite of Cocos2D using OpenGL ES 2.0. For new projects, version 2.x is probably the version you would use, and the principles of putting Cocos2D v2.x into your ARC projects are the same as for v1.1.

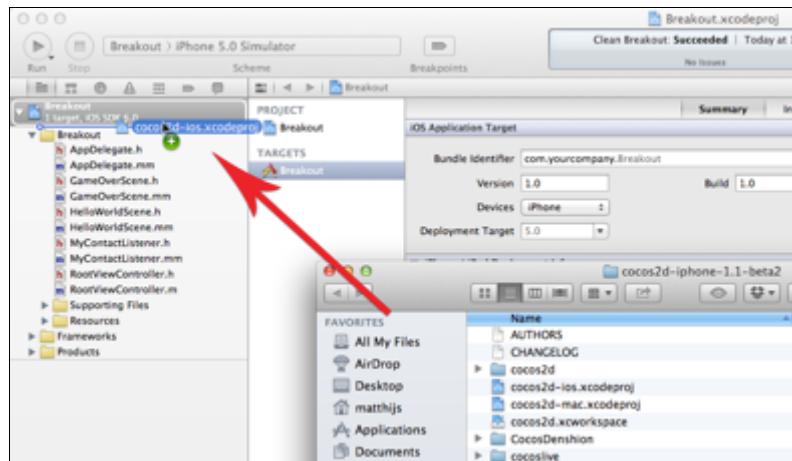
You can find the starter code with this chapter's resources. This is a simple project that uses Cocos2D and Box2D to make a ball & paddle game. Because Box2D is written in C++, most of the game's source code files are Objective-C++ (they have a **.mm** file extension).

The starter code does not include Cocos2D, so first you need to install that. Download the complete package from:

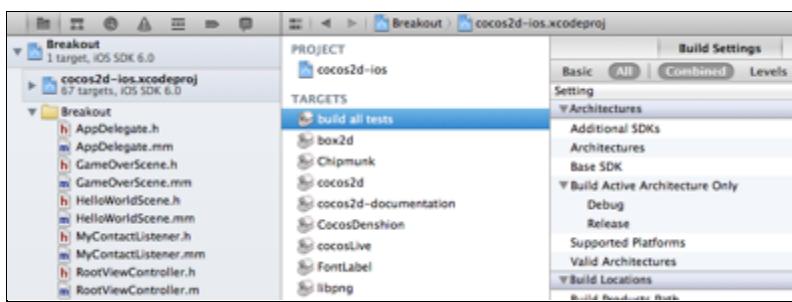
- <http://cocos2d-iphone.googlecode.com/files/cocos2d-iphone-1.1-beta2b.tar.gz>

You may also download a newer version from the v1.x branch but there could be slight differences with the descriptions that follow.

After the download is complete, unzip the package. Inside the new Cocos2D folder you will find a file named **cocos2d-ios.xcodeproj**. Drag this file into Xcode so that the **cocos2d-ios** project becomes part of the Breakout project:



The result is a project inside a project:



The cocos2d-ios project contains targets for a number of static libraries and tests. You will now add some of these libraries to the Breakout project.

Breakout already links with the following system frameworks:

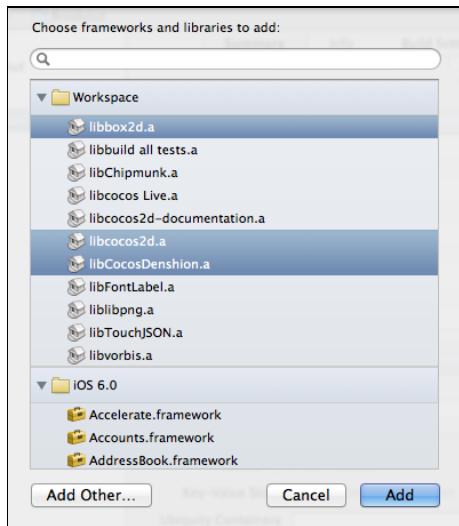
- AVFoundation
- AudioToolbox
- CoreGraphics
- Foundation
- OpenAL
- OpenGL ES
- QuartzCore
- UIKit
- libz

I mention this because you will also need to add these frameworks to your own game projects in order to use Cocos2D and the CocosDenshion audio library.

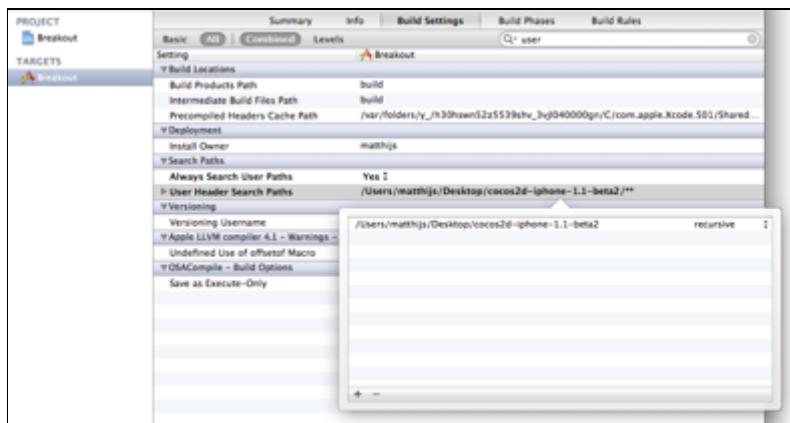
In the **Project Settings** for Breakout, under the **Summary** tab, there is a section **Linked Frameworks and Libraries** (you may have to scroll down to see it). Press the **+** button to add new libraries from the cocos2d project.

You only need these three:

- libbox2d.a
- libcocos2d.a
- libCocosDenshion.a



You are almost done. In order to make Xcode find the .h files from Cocos2D, you also have to tell it where these files are located. Go to the **Build Settings** tab, click on **All**, and search for “**user**”. Under the **Search Paths** section, set **Always Search User Paths** to **YES** and add the path to where you unpacked the Cocos2D folder to the **User Header Search Paths** setting:

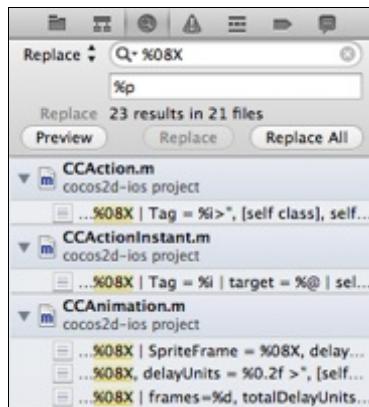


Make sure you select the Recursive option!

Tip: If the path to the Cocos2D folder contains spaces, then put double quotes around it, otherwise Xcode gets confused.

Try and build the game. If the build succeeds, great! Unfortunately, v1.1-beta2 has some issues that prevented the build from completing for me. If the build also fails for you, then you will have to edit the Cocos2D source files. Don't worry, they are small changes. It is possible that these issues have been fixed in an update by the time you read this, in which case you can skip the next bit.

Go into the Search Navigator and click on Find to change the search mode to **Replace** and make it replace the text `%08x` with `%p` in all the source files. This will get rid of most of the "Format String Issue" errors that Xcode complains about.



There are two more Format String Issue errors, in **CCTMXXMLParser.m** and in **CCAnimation.m**. Click on the error messages in the Issue Navigator and use Xcode's Fix-It suggestion to resolve both these problems.

The remaining compilation errors are in **FontLabel.m** and **FontLabelString-Drawing.m**. Both are the result of an incomplete `switch`-statement. Use the Issue Navigator to jump to these errors and add a `default` clause to the end of each `switch`:

```
default:  
    break;
```

Now the Breakout project should compile with issues. There may still be some warning messages from Box2D, but that's no problem.

Run the game and play some breakout!

Converting to ARC

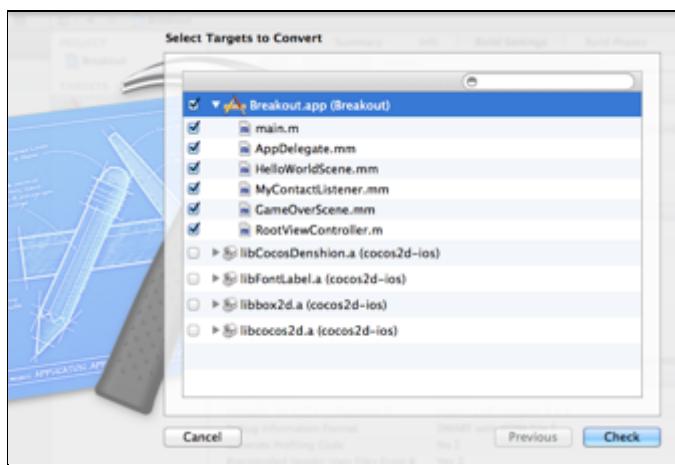
At this point, Xcode may suggest that you **Validate Project Settings** to update to the recommended settings. It's a good idea to do so, either by clicking the warning message in the Issue Navigator or the Validate Settings button in the Project Settings screen. Doing this will ensure the project is built with the latest LLVM compiler.

While you're in the Project Settings screen, it's also smart to set the **Other Warning Flags** to `-Wall` and **Run Static Analyzer** to `YES`.

Do **Product\Clean** to throw away all the files from the previous build, and compile & run the app again. You should get no new errors or warning messages. That's always a good sign. :-)

Cocos2D is a pretty big library. It would be insane to try and convert Cocos2D itself to ARC. That would probably take you a few weeks. It's smarter to convert the rest of the game but to leave Cocos2D as-is. It's very handy that Cocos2D already comes as a static library. That means you can tell Xcode to compile the Breakout project as ARC but keep the Cocos2D library as non-ARC.

Let's run the ARC conversion tool. Select the **Breakout** project and then choose **Edit\ Refactor\Convert to Objective-C ARC**. Select all the files under Breakout.app, but none of the Cocos2D or Box2D ones:



Press the Check button to perform the ARC readiness tests that determine whether the source code can be converted to ARC. Unsurprisingly, it finds a bunch of errors...

The first error in **HelloWorldScene.mm** is:

Assigning to 'void *' from incompatible type 'CCSprite * __strong'

```

59
60
61
62
63
64 // Assigning to 'void *' from incompatible type 'CCSprite * __strong'
65
66

```

It happens in the following bit of code:

```

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(100/PTM_RATIO, 100/PTM_RATIO);
ballBodyDef.userData = ball; // <--- here is the error
b2Body * ballBody = _world->CreateBody(&ballBodyDef);

```

To explain what goes wrong here, we must take a slight detour into the world of C and C++. One of the new rules of ARC is that you can no longer put Objective-C objects into C structs. That makes it too hard for the compiler to figure out when and where it must insert retain and release statements. In order to avoid any nasty problems, the compiler gurus have simply decreed that pointers to objects can no longer be placed in structs.

There are a few workarounds. You can use a `void *` instead. If you haven't done much C programming before, then think of `void *` as being similar to Objective-C's `id` type. "Void star" basically represents a pointer to anything. If C programmers need a pointer but they don't know in advance what datatype it will point to, they use a `void *`.

If you have this struct in your app,

```
typedef struct
{
    int someNumber;
    NSString *someString;
}
my_struct_t;
```

then you can change it to the following to make it compile under ARC:

```
typedef struct
{
    int someNumber;
    void *someString;
}
my_struct_t;
```

You can still store `NSString` objects into this struct, although you will have to cast them with a `__bridge` cast:

```
my_struct_t m;
m.someString = (__bridge void *)[NSString stringWithFormat:
                                @"ARC is %@!", @"awesome"];
```

Not very pretty but at least it will still work. You can also do this, by the way:

```
typedef struct
{
    int someNumber;
    __unsafe_unretained NSString *someString;
}
my_struct_t;
```

The thing to remember is that you also need to keep strong pointers to those `NSString` objects somewhere else, or they will be deallocated. Putting them inside the struct is not enough to keep these objects alive!

Back to Breakout. In `HelloWorldScene.mm` the situation is slightly different because here you're dealing with C++, not regular C. `b2BodyDef` is a class and in Objective-C++ code, classes **are** allowed to store pointers to Objective-C objects.

The problem here is that the datatype of `ballBodyDef.userData` is `void *`, not `ccsprite`. The line with the error is:

```
ballBodyDef.userData = ball;
```

ARC doesn't like this because `ball` is a `ccsprite` object, not a `void *` pointer.

You can't simply do this to fix it:

```
ballBodyDef.userData = (void *)ball;
```

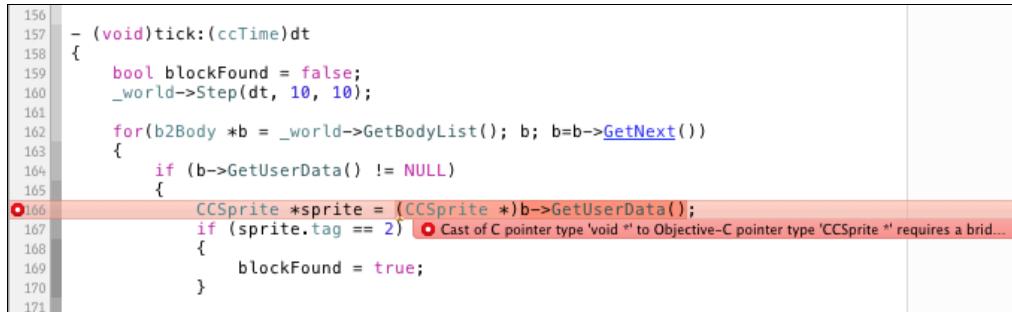
That's almost good enough, but now the compiler says: "Cast of Objective-C pointer type 'CCSprite *' to C pointer type 'void *' requires a bridged cast". This is similar to the situation you encountered before with toll-free bridging. The compiler needs to know who will be responsible for releasing the object.

In this case, the `b2BodyDef` should just have a pointer to the sprite so that the code can update the sprite's position when the Box2D body moves. The `ccsprite` object never changes owners, so the proper fix is:

```
ballBodyDef.userData = (__bridge void *)ball;
```

The next two errors in this source file have exactly the same fix.

The remaining errors are all in the `tick:` method:



A screenshot of Xcode showing the `tick:` method. The code is as follows:

```

156
157 - (void)tick:(ccTime)dt
158 {
159     bool blockFound = false;
160     _world->Step(dt, 10, 10);
161
162     for(b2Body *b = _world->GetBodyList(); b; b=b->GetNext())
163     {
164         if (b->GetUserData() != NULL)
165         {
166             CCSprite *sprite = (CCSprite *)b->GetUserData();
167             if (sprite.tag == 2) // Cast of C pointer type 'void *' to Objective-C pointer type 'CCSprite *' requires a brid...
168             {
169                 blockFound = true;
170             }
171         }
172     }
173 }
```

The line `CCSprite *sprite = (CCSprite *)b->GetUserData();` is highlighted with a red rectangle, indicating a bridging cast warning. A tooltip above the line reads: "Cast of C pointer type 'void *' to Objective-C pointer type 'CCSprite *' requires a bridged cast".

Here you do the reverse, you read the `userData` field from the `b2Body` object and cast it to a `ccsprite`. The code already performs the cast from the `void *` to `ccsprite *`, but the compiler again would like to know whether the ownership should transfer or not.

Again, adding a simple `__bridge` will suffice, also for the other errors in this method:

```
CCSprite *sprite = (__bridge CCSprite *)b->GetUserData();
```

Run the ARC conversion tool once more and now everything should check out.

Note: With Cocos2D version 1.1-beta2, the ARC conversion tool gives a couple of warnings on `ccArray.h`. The Cocos2D library itself is not converted to ARC, but because your own source files do include the Cocos2D header files, these `.h` files have to be ARC-compatible. This was not the case with earlier versions of Cocos2D (1.0.1 and before), which made it harder to embed those versions into ARC-enabled apps. With Cocos2D v1.1 and the 2.x branch, the headers are fully compatible with ARC. The warning in `ccArray.h` is harmless. If it bothers you, you can apply the Fix-Its that Xcode suggests to make it go away.

If you look at the proposed changes in the ARC conversion preview window you will see they are very similar to what happened with the Artists app. Properties go from `retain` to `strong`; `retain`, `release` and `autorelease` calls are dropped; `dealloc` is removed when no longer necessary. The changes in the code are minor.

Now you should be able to compile the app, and voila, you have a Cocos2D game running on ARC. Awesome!

More on C++

You've seen that you're not allowed to store object pointers in C-structs under ARC but that it is no problem with C++ classes. You can do the following and it will compile just fine:

```
class MyClass
{
public:
    CCSprite *sprite;
};

// elsewhere in your code:
CCSprite *ball = [CCSprite spriteWithFile:...];
MyClass *myObject = new MyClass;
myObject->sprite = ball;
```

ARC will automatically add the proper retain and release calls to the constructor and destructor of this C++ class. The `sprite` member variable is `__strong` because all variables are strong by default.

You can also use `__weak` if you must:

```
class MyClass
{
public:
    __weak CCSprite *sprite;
};
```

The same thing goes for structs in Objective-C++ code:

```
// in a .mm or .h file:
struct my_type_t
{
    CCSprite *sprite;
};

// elsewhere in your code:
static my_type_t sprites[10];
CCSprite *ball = [CCSprite spriteWithFile:...];
sprites[0].sprite = ball;
```

This is no problem whatsoever. It works because a C++ struct is much more powerful than a plain old C-struct. Just like a class it has a constructor and a destructor and ARC will use these to take care of the proper memory management.

You can also use Objective-C objects in vectors and other standard container templates:

```
std::vector<CCSprite *> sprites;
sprites.push_back(ball);
sprites.push_back(paddle);

for (std::vector<CCSprite *>::iterator i = sprites.begin();
     i != sprites.end(); ++i)
{
    NSLog(@"sprite %@", *i);
}
```

This creates a `std::vector` that keeps strong references to `ccsprite` objects.

Of course, you can also use weak references. In that case you must also take care to use the `__weak` specifier on your iterators, or the compiler will shout at you:

```
std::vector<__weak CCSprite *> sprites;
```

```
for (std::vector<__weak CCSprite *>::iterator i =
      sprites.begin(); i != sprites.end(); ++i)
{
    NSLog(@"sprite %@", *i);
}
```

It's not smart to put weak references into `NSArray`, because those pointers may become `nil` at any given time and `NSArray` doesn't allow `nil` values. But you can get away with zeroing weak references in C++ containers because unlike `NSArray`, `std::vector` can hold `NULL` objects just fine.

Making Your Own Static Library

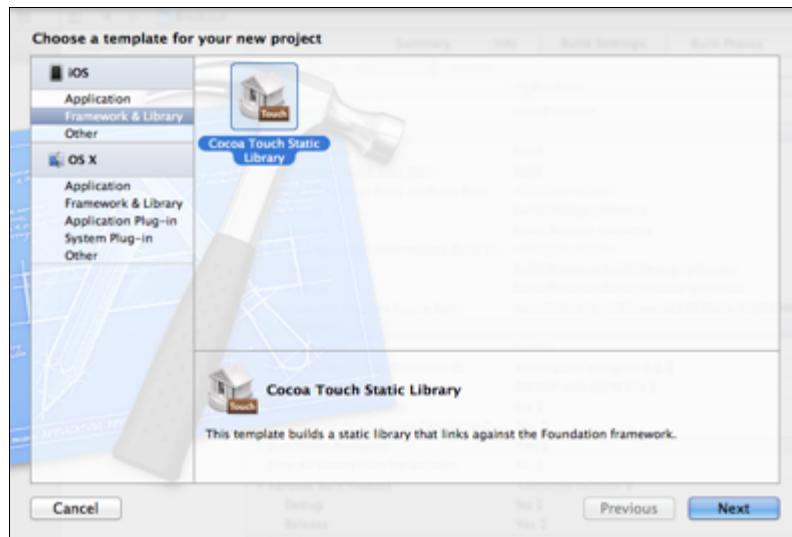
Not all third-party libraries come bundled with a handy `.xcodeproj` file for building a static library. You could add the source files to your project and then disable ARC by hand on all of them with `-fno-objc-arc`, but that can be a bit of a chore.

To keep it clear which parts of your project are ARC and which aren't, it is handy to bundle these third-party components into one or more separate static libraries that are compiled with ARC disabled, just like you've done above with Cocos2D.

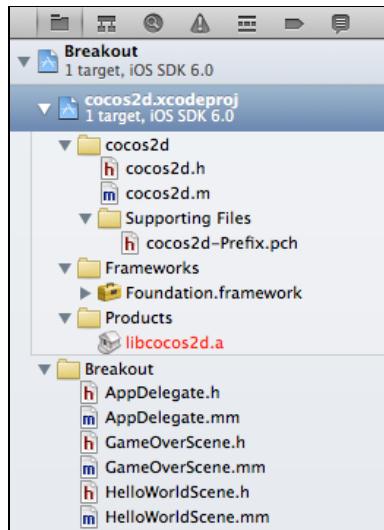
Creating your own static library is pretty easy. In this section you will create one that includes all of the Cocos2D sources that are needed for the Breakout project.

First remove the **cocos2d-ios** project from the Breakout project. Don't move it to the trash, just choose Remove References.

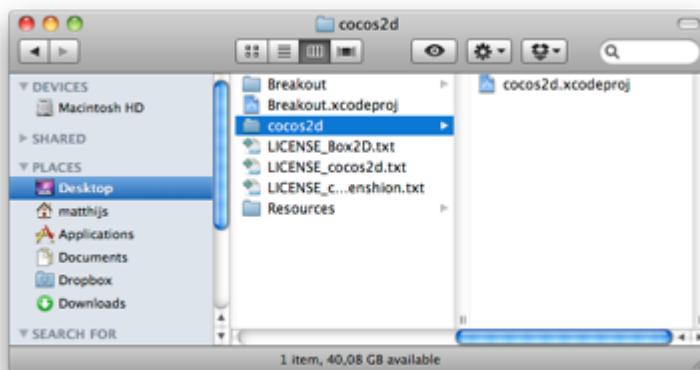
Right-click the Breakout project in the Project Navigator and choose **New Project...** Pick the **iOS\Framework & Library\Cocoa Touch Static Library** project template:



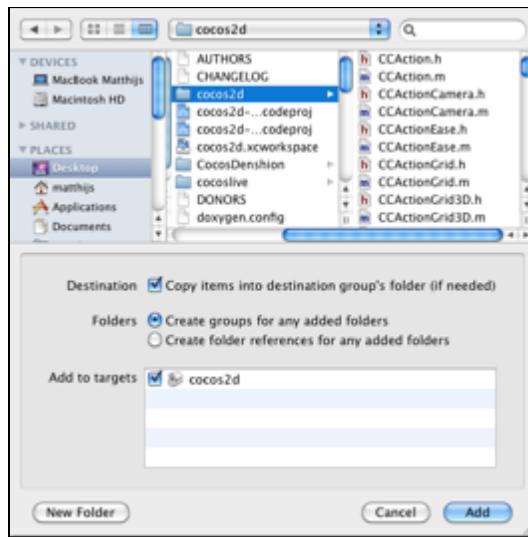
For Product Name choose “cocos2d”. Disable Automatic Reference Counting. This will add a new static library project inside the Breakout project, with a few default files:



You can delete the **cocos2d** source group. Also delete it from Finder (it will delete the files but won't remove the cocos2d subfolder by default) so that you are just left with a **cocos2d** folder with a **cocos2d.xcodeproj** file inside.



In Xcode, control-click the new **cocos2d** project in the Project Navigator and choose **Add Files**. Navigate to the folder where you unpacked Cocos2D and select the **cocos2d** folder. Make sure **Copy items into destination group's folder** is checked and that you're adding the sources to the new **cocos2d target**:



Repeat this for the **CocosDenshion** and **external/FontLabel** folders. Also add the files from **external/Box2d/Box2D** (that is two times Box2D, you don't want to add any of the Testbed files).

You need to change some build settings to make this all work. Select the **cocos2d** project and go the **Build Settings** tab.

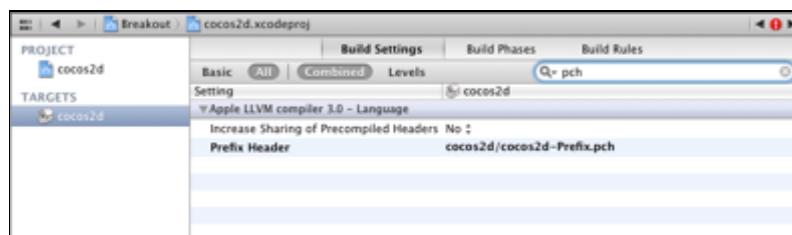
- **Search Paths\Always Search User Paths:** Set to **Yes**
- **Search Paths\User Header Search Paths:** Set to **./****

This is necessary for Box2D to find its header files.

Just to make sure, the **Skip Install** setting should be **Yes**. If it isn't, you won't be able to make archives for distribution to the App Store because the static library will be installed inside the package too and that should not happen.

By default the **iOS Deployment Target** setting for the new library is set to 6.0 (or whatever version of the SDK you're using). Change this to **5.0** if you want the app to run on iOS 5 as well.

One more thing before you can build your new static library. The default Xcode static library template is configured to include a pre-compiled header file (Prefix file). But Cocos2D doesn't use one and you deleted the one from the template earlier, so you need to tell the compiler that it shouldn't use the Prefix file. Go into the **Build Settings** and search for "**pch**":



Simply delete the **Prefix Header** option.

Using the Scheme box at the top of the Xcode window, switch the active project to **cocos2d** and press **Cmd+B** to build it. If all went well you should get no build errors (but possibly some warnings). Congrats, you just built your own static library!

Switch the active project back to Breakout. In the **Linked Frameworks and Libraries** section, add **libcocos2d.a**. This is the static library that you've just created. You should be able to build and run the Breakout app now, using your custom version of Cocos2D!

Note: If you are doing this in a real project, it may be a good idea to look at all the build settings from the official Cocos2D project and take these over to your own static library. That way you won't miss out on any important preprocessor flags and optimization settings.

Where To Go From Here?

I think ARC rocks! It's an important advancement for the Objective-C programming language. All the other new stuff in iOS 5 is cool too, but ARC completely changes – and improves! – the way we write apps. Using ARC frees you up from having to think about unimportant bookkeeping details, so you can spend that extra brainpower on making your apps even more awesome.

One of the things I've shown you in this tutorial is how to convert existing apps to ARC. These instructions assumed that you were going to migrate most of the app but you still wanted to keep certain files – usually third-party libraries – out of it. If you want to migrate at a slower pace, you can also do it the other way around. Keep the project-wide ARC setting off and convert your files to ARC one-by-one. To do this, you can simply set the compiler flag `-fobjc-arc` on the files you have converted. From then on, only these files will be compiled with ARC.

If you maintain a reusable library and you don't want to switch it over to ARC (yet?), then you can use preprocessor directives to make it at least compatible with ARC where necessary. You can test for ARC with:

```
#if __has_feature(objc_arc)
// do your ARC thing here
#endif
```

Or even safer, if you also still want to support the old GCC compiler:

```
#if defined(__has_feature) && __has_feature(objc_arc)
// do your ARC thing here
```

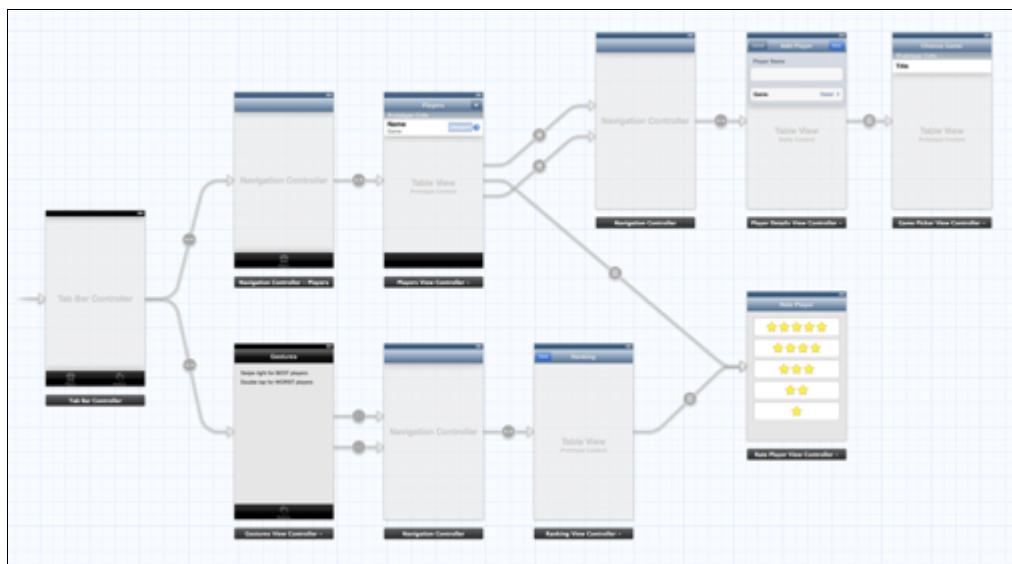
```
#endif
```

The official documentation for ARC is “Transitioning to ARC Release Notes” that you can find in your Xcode documentation. Other good sources for information are WWDC 2011 video 323, Introducing Automatic Reference Counting, and video 322, Objective-C Advancements in Depth.

Chapter 4: Beginning Storyboards

By Matthijs Hollemans

Storyboarding is an exciting new feature in iOS 5 that will save you a lot of time building user interfaces for your apps. To show you what a storyboard is, I'll let a picture do the talking. This is the storyboard for the app that you will be making in this chapter and the next:



You may not know exactly yet what the app does but you can clearly see which screens it has and how they are related. That is the power of using storyboards.

If you have an app with many different screens then storyboards can help reduce the amount of glue code you have to write to go from one screen to the next. Instead of using a separate nib file for each view controller, your app uses a single storyboard that contains the designs of all of these view controllers and the relationships between them.

Storyboards have a number of advantages over regular nibs:

- With a storyboard you have a better conceptual overview of all the screens in your app and the connections between them. It's easier to keep track of

everything because the entire design is in a single file, rather than spread out over many separate nibs.

- The storyboard describes the transitions between the various screens. These transitions are called “segues” and you create them by simply ctrl-dragging from one view controller to the next. Thanks to segues you need less code to take care of your UI.
- Storyboards make working with table views a lot easier with the new prototype cells and static cells features. You can design your table views almost completely in the storyboard editor, something else that cuts down on the amount of code you have to write.

Not everything is perfect, of course, and storyboards do have some limitations. The storyboard version of Interface Builder isn’t as powerful as the old nib editor and there are a few handy things nibs can do that storyboards unfortunately can’t. You also need a big monitor, especially when you write iPad apps!

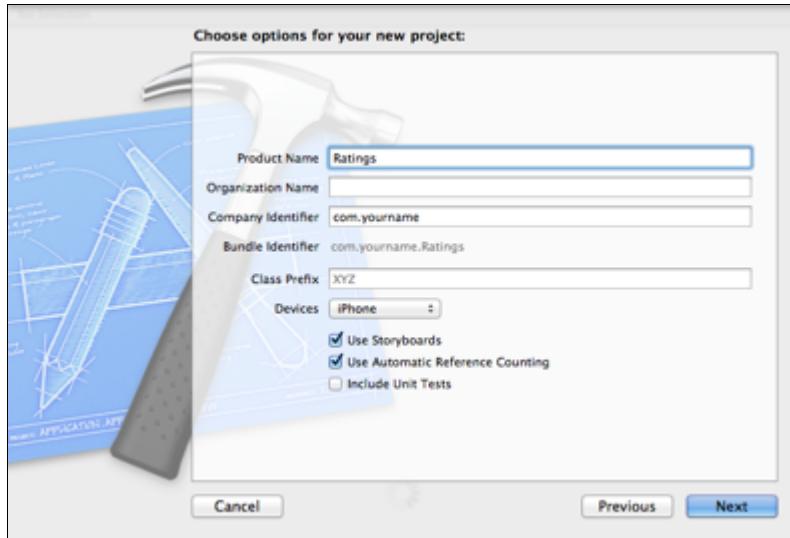
If you’re the type who hates Interface Builder and who really wants to create his entire UI programmatically, then storyboards are probably not for you. Personally, I prefer to write as little code as possible – especially UI code! – so this tool is a welcome addition to my arsenal.

You can still use nibs with iOS 5. Nib files aren’t suddenly frowned upon now that we have storyboards. If you want to keep using nibs then go right ahead, but know that you can also combine storyboards with nibs. It’s not an either-or situation.

In this tutorial you’ll take a look at what you can do with storyboards. You’re going to build a simple app that lets you create a list of players and games, and rate their skill levels. In the process, you’ll learn the most common tasks that you’ll be using storyboards for on a regular basis!

Getting Started

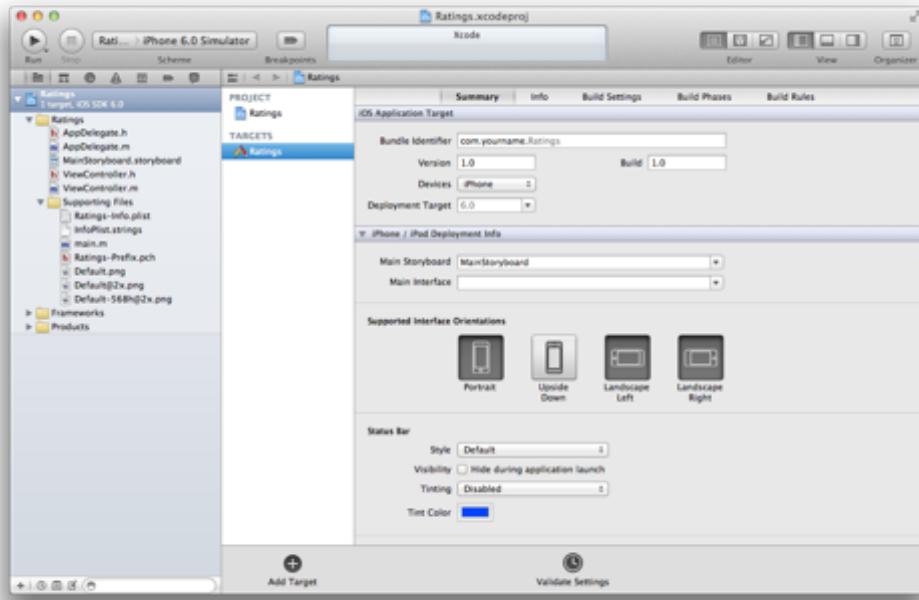
Fire up Xcode 4.5 or better and create a new project. You’ll use the Single View Application template as the starting point and then build up the app from there.



Fill in the template options as follows:

- **Product Name:** Ratings
- **Organization Name:** fill this in however you like
- **Company Identifier:** the identifier that you use for your apps, in reverse domain notation
- **Class Prefix:** leave this empty
- **Devices:** iPhone
- **Use Storyboards:** check this
- **Use Automatic Reference Counting:** check this
- **Include Unit Tests:** this should be unchecked

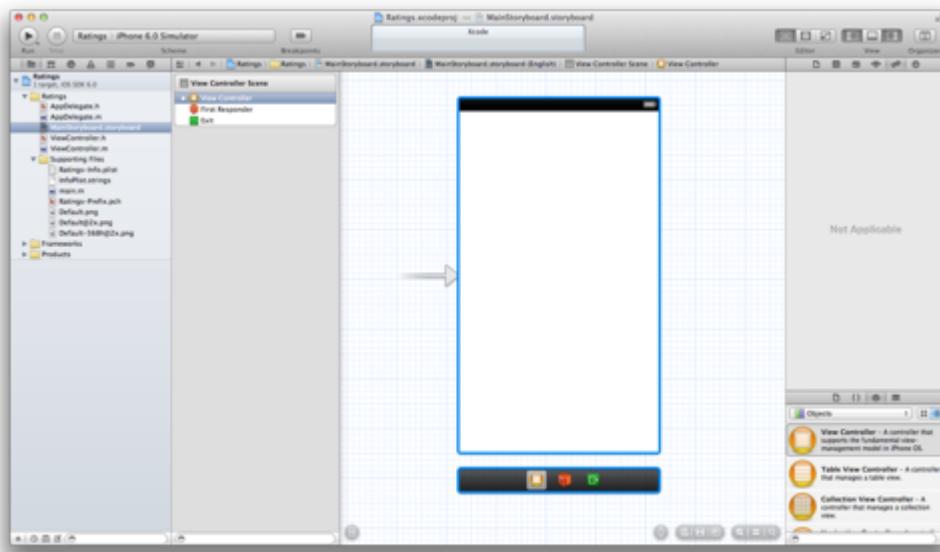
After Xcode has created the project, the main Xcode window looks like this:



The new project consists of two classes, `AppDelegate` and `ViewController`, and the star of this tutorial: the **MainStoryboard.storyboard** file. Notice that there are no `.xib` files in the project, not even `MainWindow.xib`.

This is a portrait-only app, so before you continue, uncheck the **Landscape Left** and **Landscape Right** options under **Supported Interface Orientations**. Also set the Deployment Target to 5.0 to make this app run on iOS 5.0 and up.

Now let's take a look at that storyboard. Click the **MainStoryboard.storyboard** file in the Project Navigator to open it up in Interface Builder:



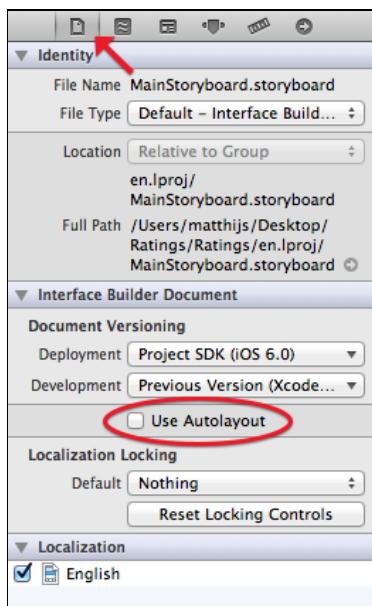
Editing storyboards in Interface Builder works pretty much the same way as editing nibs. You can drag new controls from the Object Library (see bottom-right corner) into your view controller to design its layout. The difference is that the storyboard doesn't contain just one view controller from your app, but all of them.

The official storyboard terminology for a view controller is "scene", but you can use the terms interchangeably. The scene simply represents the view controller in the storyboard. Previously you would use a separate nib for each scene / view controller, but now they are all combined into a single storyboard.

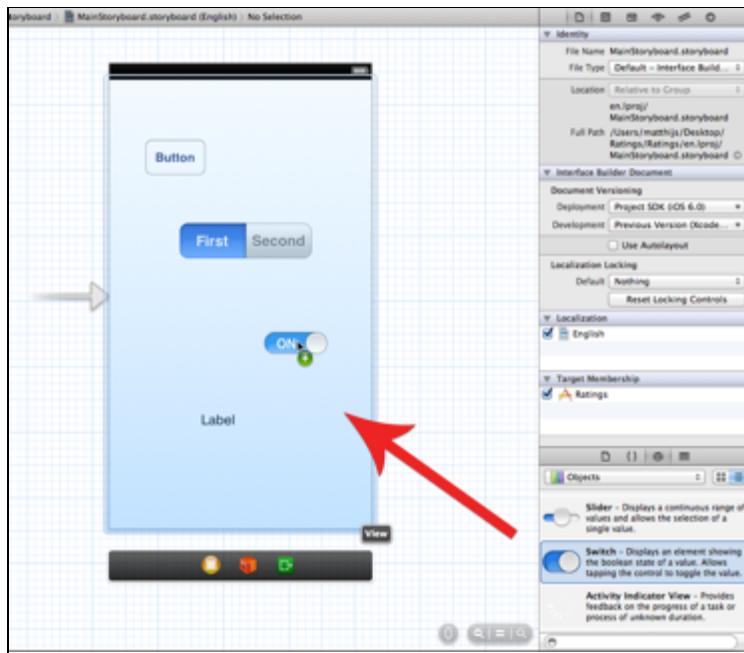
On the iPhone usually only one of these scenes is visible at a time, but on the iPad you can show several at once, for example the master and detail panes in a split-view, or the content of a popover.

Note: Xcode 4.5 enables Auto Layout by default for storyboard and nib files. Auto Layout is a cool new technology for making flexible user interfaces that can easily resize, which is useful on the iPad and for supporting the new iPhone 5, but the downside is that it only works on iOS 6 and up. To learn more about Auto Layout, see our new book *iOS 6 by Tutorials*.

Disable Auto Layout from the File inspector for the storyboard:



To get some feel for how the storyboard editor works, drag some controls into the blank view controller:

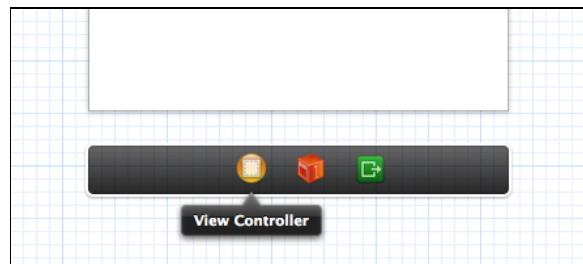


The sidebar on the left is the Document Outline:



When editing a nib this area lists just the components from that one nib but for a storyboard it shows the contents of all your view controllers. Currently there is only one view controller (or scene) in your storyboard but in the course of this tutorial you'll be adding several others.

There is a miniature version of this Document Outline below the scene, named the Dock:

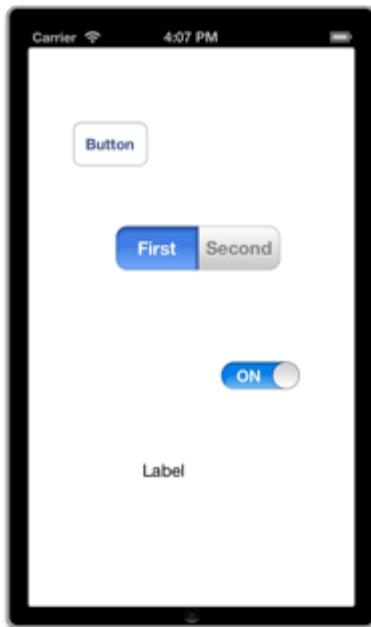


The Dock shows the top-level objects in the scene. Each scene has at least a View Controller object and a First Responder object, but it can potentially have other top-

level objects as well. The Dock is convenient for making connections to outlets and actions. If you need to connect something to the view controller, you can simply drag to its icon in the Dock.

Note: You probably won't be using the First Responder very much. This is a proxy object that refers to whatever object has first responder status at any given time. It was also present in your nibs and you probably never had a need to use it then either. As an example, you can hook up the Touch Up Inside event from a button to First Responder's `cut:` selector. If at some point a text field has input focus then you can press that button to make the text field, which is now the first responder, cut its text to the pasteboard.

Run the app and it should look exactly like what you designed in the editor (shown here on the iPhone 5 simulator):



No more MainWindow.xib

If you've ever made a nib-based app before then you always had a `MainWindow.xib` file. This nib contained the top-level `UIWindow` object, a reference to the App Delegate, and one or more view controllers. When you put your app's UI in a storyboard, however, `MainWindow.xib` is no longer used.



So how does the storyboard get loaded by the app if there is no MainWindow.xib file?

Let's take a peek at the application delegate. Open up **AppDelegate.h** and you'll see it looks like this:

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

@interface AppDelegate : UIResponder <UIApplicationDelegate>

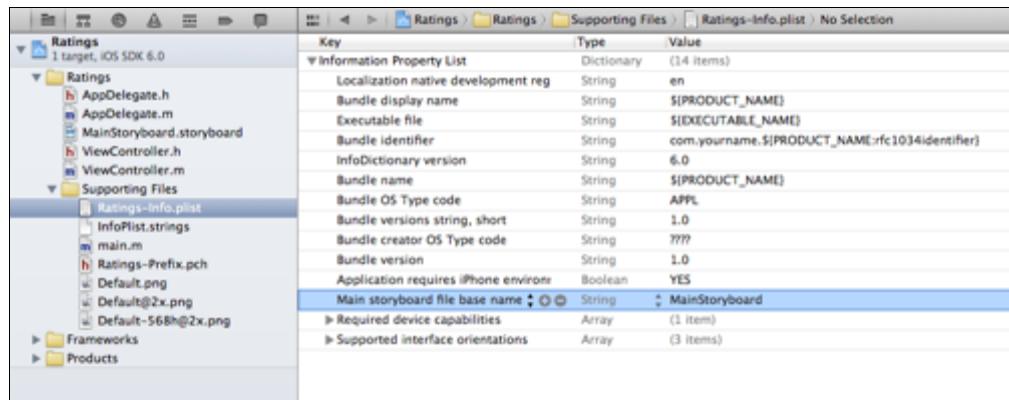
@property (strong, nonatomic) UIWindow *window;

@end
```

It is a requirement for using storyboards that your application delegate inherits from `UIResponder` (previously it used to inherit directly from `NSObject`) and that it has a `UIWindow` property (unlike before, this is not an `IBOutlet`).

If you look into **AppDelegate.m**, you'll see that it does absolutely nothing, all the methods are practically empty. Even `application:didFinishLaunchingWithOptions:` simply returns `YES`. Previously, this method would either add the main view controller's view to the window or set the window's `rootViewController` property, but none of that happens here.

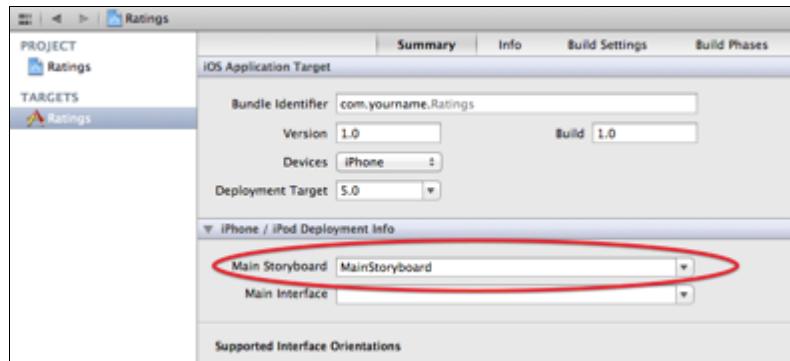
The secret is in the Info.plist file. Click on **Ratings-Info.plist** (you can find it in the **Supporting Files** group) and you'll see this:



In nib-based projects there was a key in Info.plist named `NSMainNibFile`, or "Main nib file base name", that instructed `UIApplication` to load `MainWindow.xib` and hook it into the app. Your Info.plist no longer has that setting.

Instead, storyboard apps use the key `UIStoryboardName`, or "Main storyboard file base name", to specify the name of the storyboard that must be loaded when the app starts. When this setting is present, `UIApplication` will load the "MainStoryboard.storyboard" file and automatically instantiates the first view controller from that storyboard and puts its view into a new `UIWindow` object. No programming necessary.

You can also see this in the Target Summary screen:



There is a new iPhone/iPod Deployment Info section that lets you choose between starting from a storyboard or from a nib file.

For the sake of completeness, also open `main.m` to see what's in there:

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

#import "AppDelegate.h"

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

```
        NSStringFromClass([AppDelegate class]));
    }
}
```

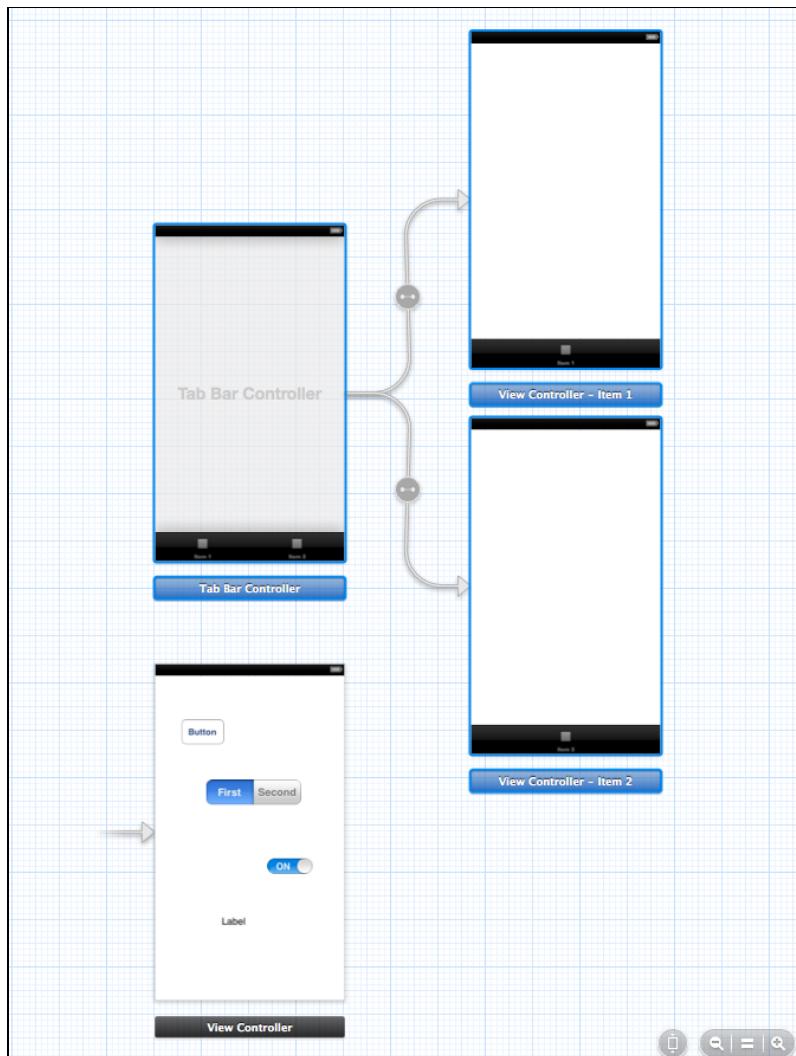
Previously, the last parameter for `UIApplicationMain()` was simply `nil` but now it is the mouthful `NSStringFromClass([AppDelegate class])`.

A big difference with having a `MainWindow.xib` is that the app delegate is not part of the storyboard. Because the app delegate is no longer being loaded from a nib (nor from the storyboard), you have to tell the `UIApplicationMain()` function specifically what the name of your app delegate class is, otherwise it won't be able to find it.

Just Add It To My Tab

The Ratings app has a tabbed interface with two screens. With a storyboard it is really easy to create tabs.

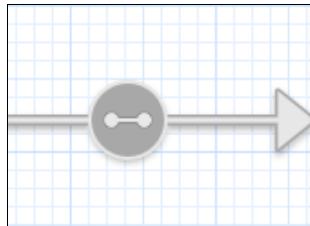
Switch back to **MainStoryboard.storyboard** and drag a Tab Bar Controller from the Object Library into the canvas. You may want to maximize your Xcode window first, because the Tab Bar Controller comes with two view controllers attached and you'll need some room to maneuver. You can zoom out using the little floating panel in the bottom-right corner of the canvas.



The new Tab Bar Controller comes pre-configured with two other view controllers; one for each tab. `UITabBarController` is a so-called *container* view controller because it contains one or more other view controllers. Two other common containers are the Navigation Controller and the Split View Controller (you'll see both of them later).

Note: Another cool addition to iOS 5 is a new API for writing your own container controllers – and later on in this book, we have a chapter on that!

The container relationship is represented in the Storyboard Editor by the arrows between the Tab Bar controller and the view controllers that it contains.

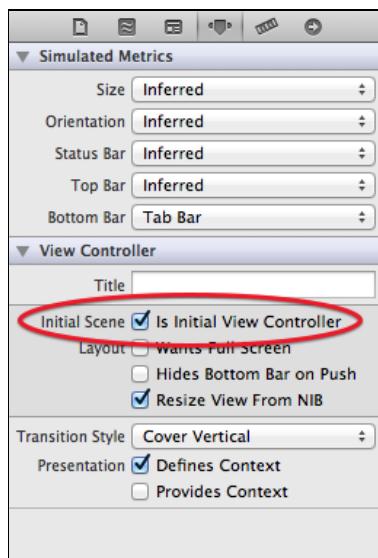


Tip: If you want to move the Tab Bar controller and its attached view controllers as a group, you can Cmd-click (or shift-click) to select multiple scenes and then move them around together. (Selected scenes have a thick blue outline.)

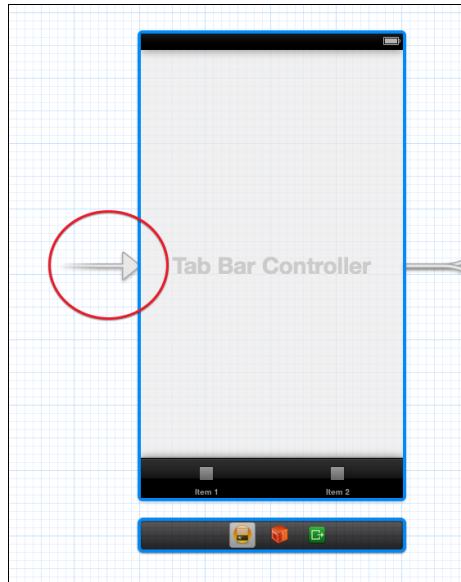
Drag a label into the first view controller and give it the text "First Tab". Also drag a label into the second view controller and name it "Second Tab". This allows you to actually see something happen when you switch between the tabs.

Note: You can't drag stuff into the scenes when the editor is zoomed out. You'll need to return to the normal zoom level first. You can do that quickly by double-clicking in the canvas.

Select the Tab Bar Controller and go to the Attributes inspector. Check the box that says **Is Initial View Controller**.



In the canvas the arrow that at first pointed to the regular view controller now points at the Tab Bar Controller:



This means that when you run the app, `UIApplication` will make the Tab Bar Controller the main screen. The storyboard always has a single view controller that is designated the *initial view controller*, that serves as the entry point into the storyboard.

Tip: To change the initial view controller, you can also drag the arrow between view controllers.

Run the app and try it out. The app now has a tab bar and you can switch between the two view controllers with the tabs:



Xcode actually comes with a template for building a tabbed app (unsurprisingly called the Tabbed Application template) that you could have used, but it's good to know how this works so you can also create one by hand if you have to.

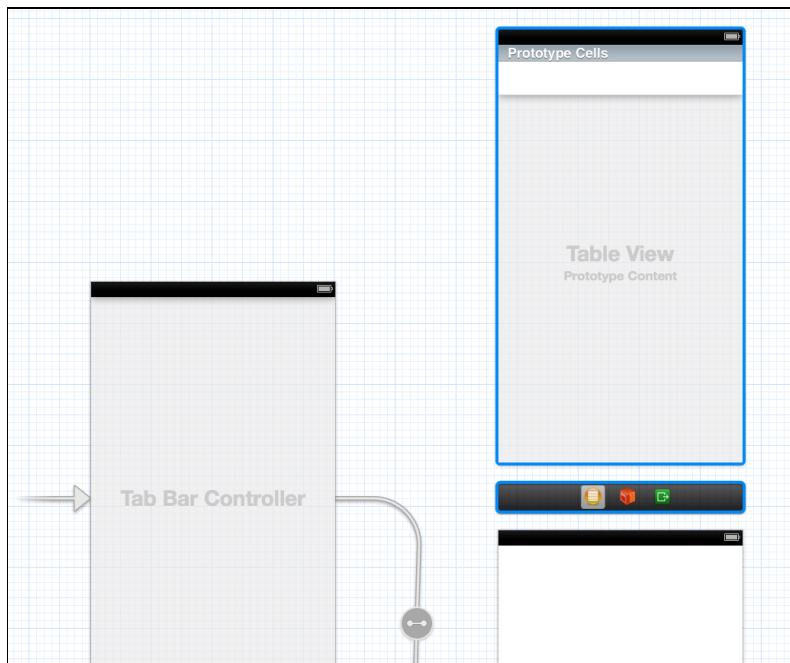
Remove the view controller that was originally added by the template, as you'll no longer be using it. The storyboard now contains just the tab bar and the two scenes for its tabs.

By the way, if you connect more than five scenes to the Tab Bar Controller, it automatically gets a More... tab when you run the app. Pretty neat!

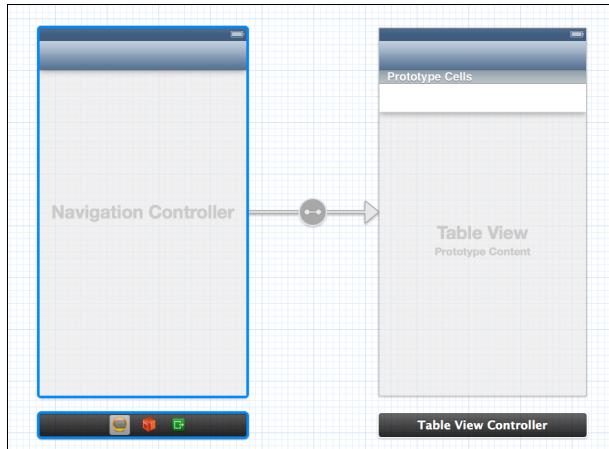
Adding a Table View Controller

The two scenes that are currently attached to the Tab Bar Controller are both regular `UIViewController`s. You are going to replace the scene from the first tab with a `UITableView` instead.

Click on that first view controller to select it, and then delete it. From the Object Library drag a new Table View Controller into the canvas in the place where that previous scene used to be:



With the Table View Controller selected, choose **Editor\Embed In\Navigation Controller** from Xcode's menubar. This adds yet another view controller to the canvas:



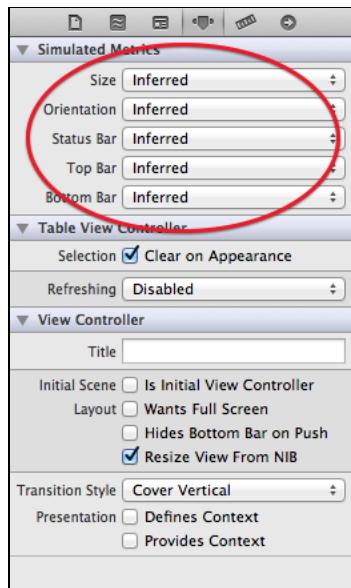
You could also have dragged in a Navigation Controller from the Object Library, but this Embed In command is just as easy.

Because the Navigation Controller is also a container view controller (just like the Tab Bar Controller), it has a relationship arrow pointing at the Table View Controller. You can also see these relationships in the Document Outline:



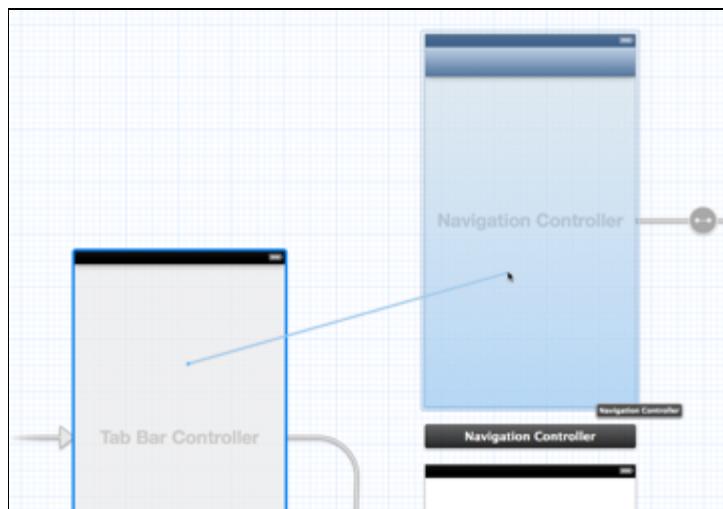
Notice that embedding the Table View Controller gave it a navigation bar. Interface Builder automatically put it there because this scene will now be displayed inside the Navigation Controller's frame. It's not a real `UINavigationBar` object but a simulated one.

If you look at the Attributes inspector for the Table View Controller, you'll see the Simulated metrics section at the top:

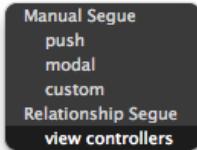


"Inferred" is the default setting for storyboards and it means the scene will show a navigation bar when it's inside of a Navigation Controller, a tab bar when it's inside of a Tab Bar Controller, and so on. You could override these settings if you wanted to, but keep in mind they are here only to help you design your screens. The Simulated Metrics aren't used during runtime; they're just a visual design aid that shows what your screen will end up looking like.

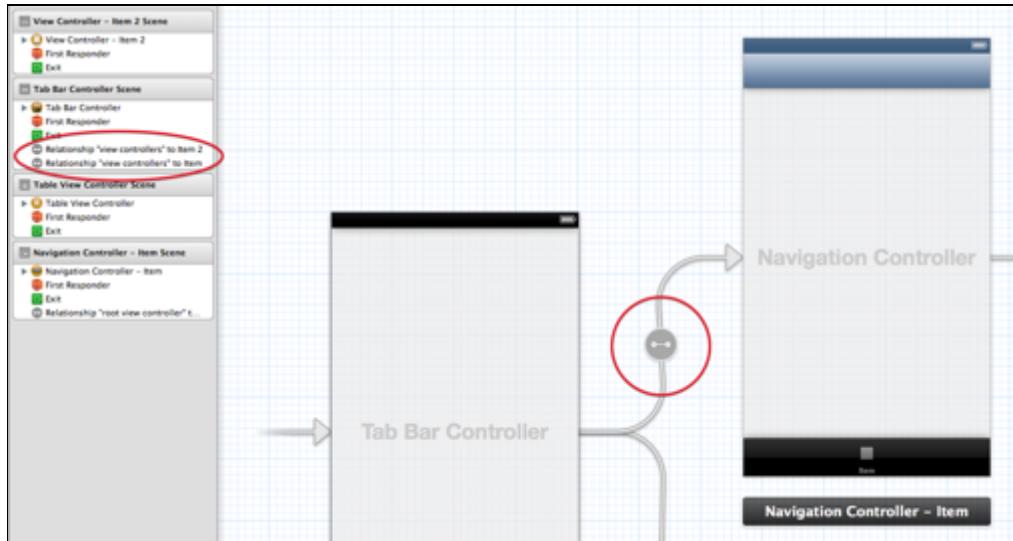
Let's connect these two new scenes to the Tab Bar Controller. Ctrl-drag from the Tab Bar Controller to the Navigation Controller:



When you let go, a small popup menu appears:

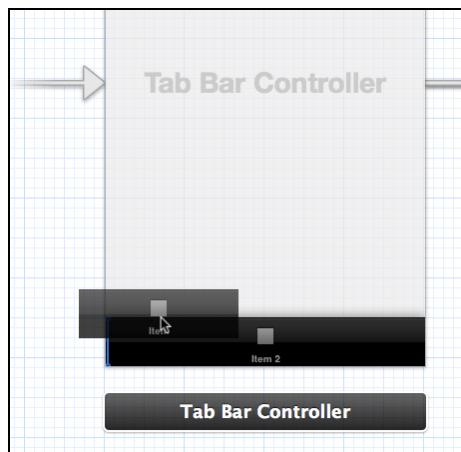


Choose the **Relationship Segue – view controllers** option. This creates a new relationship arrow between the two scenes:



The Tab Bar Controller has two such relationships, one for each tab. The Navigation Controller itself has a relationship connection to the Table View Controller. There is also another type of arrow, the segue, that we'll talk about later.

When you made this new connection, a new tab was added to the Tab Bar Controller, simply named "Item". For this app, you want this new scene to be the first tab, so drag the tabs around to change their order:



Note: If Interface Builder doesn't let you drag to rearrange the tabs inside the Tab Bar Controller, then temporarily go to another file in the project and switch back to the storyboard again. That seems to fix it. Or follow the iOS developer's motto: When in doubt, restart Xcode.

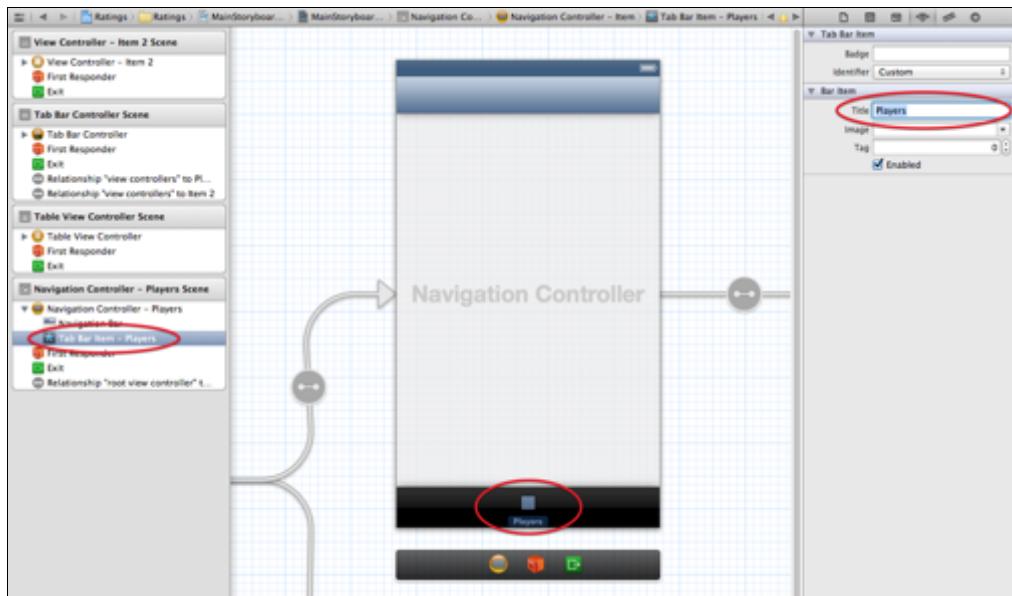
Run the app and try it out. The first tab now contains a table view inside a navigation controller.



Before you put some actual functionality into this app, let's clean up the storyboard a little. You will name the first tab "Players" and the second "Gestures". Unlike what you may expect, you do not change this on the Tab Bar Controller itself, but in the view controllers that are connected to these tabs.

As soon as you connect a view controller to the Tab Bar Controller, it is given a Tab Bar Item object. You use the Tab Bar Item to configure the tab's title and image.

Select the Tab Bar Item inside the Navigation Controller, and in the Attributes inspector set its Title to "Players":



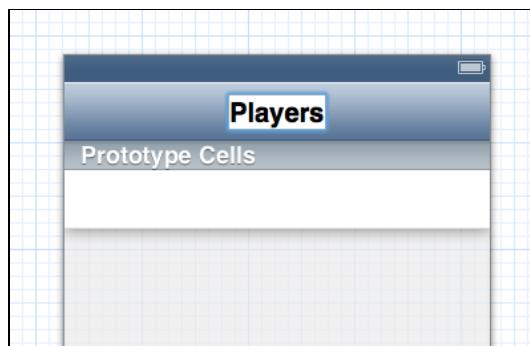
Rename the Tab Bar Item for the view controller from the second tab to "Gestures".

A well-designed app should also put some pictures on these tabs. The resources for this chapter contain a subfolder named Images. Add that folder to the project.

Then, in the Attributes inspector for the Players Tab Bar Item, choose the **Players.png** image. You probably guessed it, give the Gestures item the image **Gestures.png**.

A view controller that is embedded inside a Navigation Controller has a Navigation Item that is used to configure the navigation bar. Select the Navigation Item for the Table View Controller and change its title in the Attributes inspector to "Players".

Alternatively, you can double-click the navigation bar and change the title there. (Note: You should double-click the simulated navigation bar in the Table View Controller, not the actual Navigation Bar object in the Navigation Controller.)

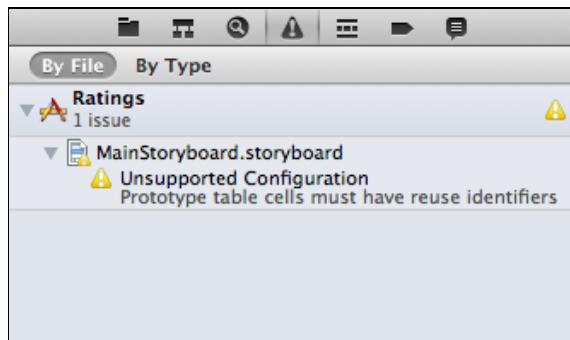


Run the app and marvel at your pretty tab bar, all without writing a single line of code!



Prototype Cells

You may have noticed that ever since you added the Table View Controller, Xcode has been complaining:



The warning message is, "Unsupported Configuration: Prototype table cells must have reuse identifiers". When you add a Table View Controller to a storyboard, it wants to use *prototype cells* by default but you haven't properly configured this yet, hence the warning.

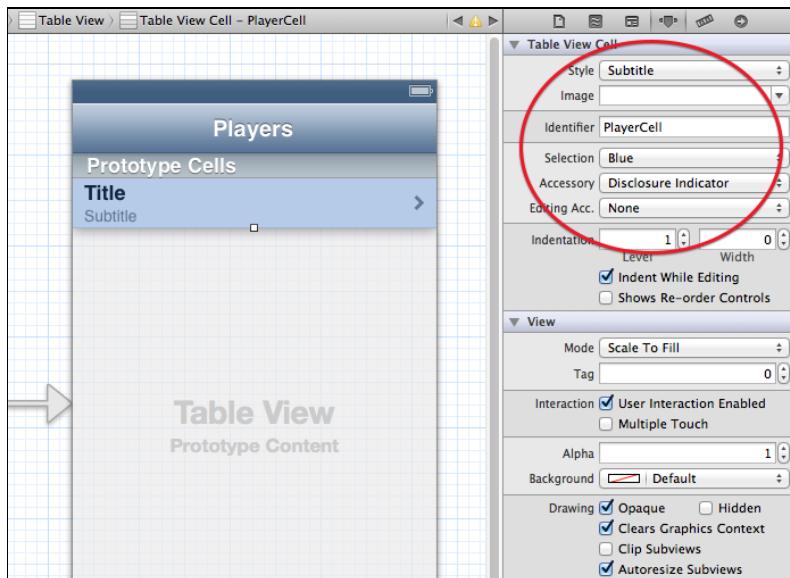
Prototype cells allow you to easily design a custom layout for your cells directly from within the storyboard editor.

Note: Prior to iOS 5, if you wanted to use a table view cell with a custom design you either had to add your subviews to the cell programmatically, or create a new nib specifically for that cell and then load it from the nib with some magic. iOS 5 makes this much easier – you can use the prototype cell

functionality built into the Storyboard editor, or the new NIB cell registration functionality you'll read about later in this book.

The Table View Controller comes with a blank prototype cell. Click on that cell to select it and in the Attributes inspector set the Style option to Subtitle. This immediately changes the appearance of the cell to include two labels.

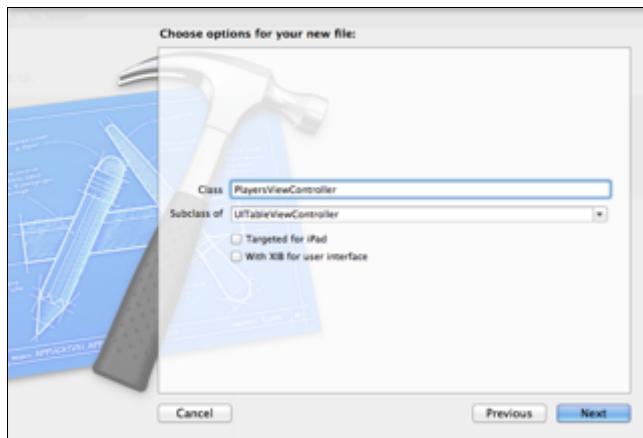
If you've used table views before and created your own cells by hand, you may recognize this as the `UITableViewCellStyleSubtitle` style. With prototype cells you can either pick one of the built-in cell styles as you just did, or create your own custom design (which you'll do shortly).



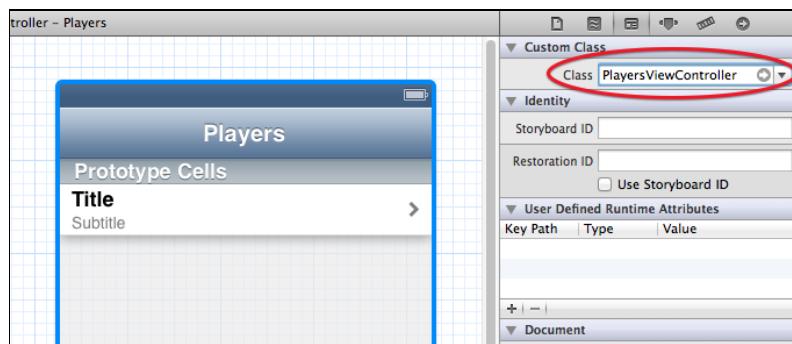
Set the Accessory attribute to Disclosure Indicator and give the cell the Reuse Identifier **PlayerCell**. That will make Xcode shut up about the warning. All prototype cells are still regular `UITableViewCell` objects and therefore should have a reuse identifier. Xcode is just making sure you don't forget (at least for those of us who pay attention to its warnings).

Run the app, and... nothing has changed. That's not so strange, you still have to make a data source for the table so it will know what rows to display.

Add a new file to the project. Choose the **Objective-C class** template. Name the class **PlayersViewController** and make it a subclass of **UITableViewController**. The **With XIB for user interface** option should be unchecked because you already have the design of this view controller in the storyboard. No nibs today!



Go back to the storyboard and select the Table View Controller. In the Identity inspector, set its Class to **PlayersViewController**. That is the essential step for hooking up a scene from the storyboard with your own view controller subclass. Don't forget this or your class won't be used!



From now on when you run the app that table view controller from the storyboard is actually an instance of the `PlayersViewController` class.

Add a mutable array property to **PlayersViewController.h**:

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

@interface PlayersViewController : UITableViewController

@property (nonatomic, strong) NSMutableArray *players;

@end
```

This array will contain the main data model for the app, an array that contains `Player` objects. Add a new file to the project using the **Objective-C class** template. Name it **Player**, subclass of **NSObject**.

Change **Player.h** to the following:

```
@interface Player : NSObject
```

```
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *game;
@property (nonatomic, assign) int rating;

@end
```

There's nothing special going on here. `Player` is simply a container object for these three properties: the name of the player, the game he's playing, and a rating of 1 to 5 stars.

Note: Thanks to recent improvements to the LLVM compiler that ships with Xcode, it is no longer necessary to `@synthesize` your properties. Sweet!

You'll make the array of players and some test `Player` objects in the App Delegate and then assign it to the `PlayersViewController`'s `players` property. The reason you make this array inside `AppDelegate` and not inside `PlayersViewController` is that in the next chapter you will also pass the array to the view controller for the second tab.

In `AppDelegate.m`, add an `#import` for the `Player` and `PlayersViewController` classes at the top of the file, and add a new instance variable named `_players`:

```
#import "AppDelegate.h"
#import "Player.h"
#import "PlayersViewController.h"

@implementation AppDelegate
{
    NSMutableArray *_players;
}

. . .
```

Then change the `didFinishLaunchingWithOptions` method to:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    _players = [NSMutableArray arrayWithCapacity:20];

    Player *player = [[Player alloc] init];
    player.name = @"Bill Evans";
    player.game = @"Tic-Tac-Toe";
    player.rating = 4;
```

```
[_players addObject:player];

player = [[Player alloc] init];
player.name = @"Oscar Peterson";
player.game = @"Spin the Bottle";
player.rating = 5;
[_players addObject:player];

player = [[Player alloc] init];
player.name = @"Dave Brubeck";
player.game = @"Texas Hold'em Poker";
player.rating = 2;
[_players addObject:player];

UITabBarController *tabBarController =
    (UITabBarController *)self.window.rootViewController;

UINavigationController *navigationController =
    [tabBarController viewControllers][0];

PlayersViewController *playersViewController =
    [navigationController viewControllers][0];

playersViewController.players = _players;

return YES;
}
```

This simply creates some `Player` objects and adds them to the `_players` array. But then it does the following:

```
UITabBarController *tabBarController =
    (UITabBarController *)self.window.rootViewController;

UINavigationController *navigationController =
    [tabBarController viewControllers][0];

PlayersViewController *playersViewController =
    [navigationController viewControllers][0];

playersViewController.players = _players;
```

Yikes, what is that?! You want to assign the `_players` array to the `players` property of `PlayersViewController` so it can use this array for its data source. But the app delegate doesn't know anything about `PlayersViewController` yet, so it will have to dig through the storyboard to find it.

This is one of the limitations of storyboards that is slightly annoying. With nibs you always had a reference to the App Delegate in your MainWindow.xib and you could make connections from your top-level view controllers to outlets on the App Delegate.

That is currently not possible with storyboards. You cannot make references to the app delegate from your top-level view controllers. That's unfortunate, but you can always get those references programmatically, which is what you do here.

Let's take it step-by-step:

```
UITabBarController *tabBarController =
    (UITabBarController *)self.window.rootViewController;
```

You know that the storyboard's initial view controller is a Tab Bar Controller, so you can look up the window's rootViewController and cast it.

The PlayersViewController sits inside a navigation controller in the first tab, so you first look up that UINavigationController object,

```
UINavigationController *navigationController =
    [tabBarController viewControllers][0];
```

and then ask it for its root view controller, which is the PlayersViewController that you are looking for:

```
PlayersViewController *playersViewController =
    [navigationController viewControllers][0];
```

It takes a bit of effort to dig through the storyboard to get the view controller you want, but that's the way to do it. Of course, if you change the order of the tabs, or change the app so that it no longer has a Tab Bar Controller at the root, then you will have to revise this logic as well.

Note: The notation [0] is new with the latest versions of Xcode. Previously, in order to index an array you had to call [array objectAtIndex:index] but from now on you can simply type [index] to do the same thing.

Now that you have an array full of Player objects, you can continue building the data source for PlayersViewController. Open up **PlayersViewController.m**, and add an import at the top:

```
#import "Player.h"
```

Change the table view data source methods to the following:

```
#pragma mark - Table view data source
```

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

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

The real work happens in `cellForRowAtIndexPath`. Previously, this method typically looked something like this:

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

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil)
    {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier];
    }

    // Configure the cell...

    return cell;
}
```

You would ask the table view to dequeue a cell and if that returned `nil` because there were no free cells to reuse, you would create a new instance of the cell class. That is no doubt how you've been writing your own table view code all this time. Well, no longer!

Replace that method with:

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

```
UITableViewCell *cell = [tableView  
    dequeueReusableCellWithIdentifier:@"PlayerCell"];  
  
Player *player = (self.players)[indexPath.row];  
cell.textLabel.text = player.name;  
cell.detailTextLabel.text = player.game;  
return cell;  
}
```

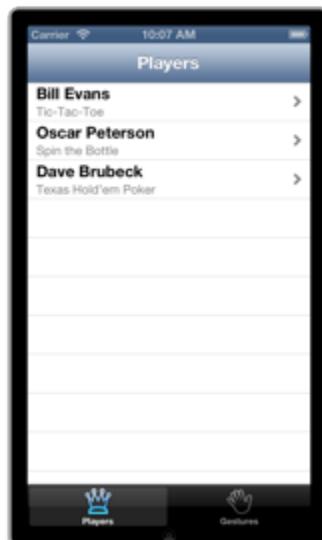
That looks a lot simpler! The only thing you need to do to get a new cell is:

```
UITableViewCell *cell = [tableView  
    dequeueReusableCellWithIdentifier:@"PlayerCell"];
```

If there is no existing cell that can be recycled, this will automatically make a new copy of the prototype cell and return it to you. All you need to do is supply the reuse identifier that you set on the prototype cell in the storyboard editor, in this case "PlayerCell". Don't forget to set that identifier, or this little scheme won't work!

Note: When you create a UITableViewController class using the Xcode template, it already puts some default code into `cellForRowAtIndexPath`. That code calls the method `dequeueReusableCellWithIdentifier:forIndexPath:` instead of just `dequeueReusableCellWithIdentifier:`. Be aware that this new method only works on iOS 6. For backwards compatibility with iOS 5, use the version without the `forIndexPath:` parameter.

Now you can run the app, and lo and behold, the table view has players in it:



It just takes one line of code to use these newfangled prototype cells. I think that's just great!

Note: In this app you're using only one prototype cell but if your table needs to display different kinds of cells then you can simply add additional prototype cells to the storyboard. You can either duplicate the existing cell to make a new one, or increment the value of the Table View's Prototype Cells attribute. Be sure to give each cell its own re-use identifier, though.

Designing Your Own Prototype Cells

Using a standard cell style is OK for many apps, but for this app you want to add an image on the right-hand side of the cell that shows the player's rating (one to five stars). Having an image view in that spot is not supported by the standard cell styles, so you'll have to make a custom design.

Switch back to **MainStoryboard.storyboard**, select the prototype cell in the table view, and set its Style attribute to Custom. The default labels now disappear.

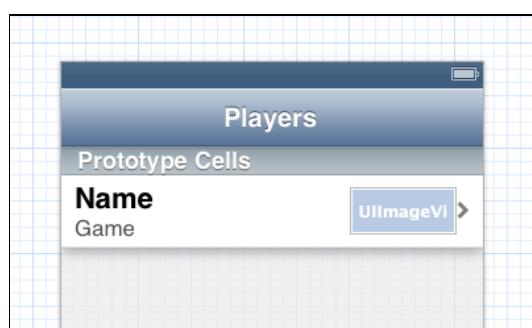
First make the cell a little taller. Either drag its handle at the bottom or change the Row Height value in the Size inspector. Make the cell 55 points high.

Drag two Label objects from the Objects Library into the cell and place them roughly where the standard labels were previously. Just play with the font and colors and pick something you like.

Set the Highlighted color of both labels to white. That will look better when the user taps the cell and the cell background turns blue.

Drag an Image View into the cell and place it on the right, next to the disclosure indicator. Make it 81 points wide, the height isn't very important. Set its Mode to Center (under View in the Attributes inspector) so that whatever image you put into this view is not stretched.

Resize the labels so they don't overlap with the image view (about 200 points wide). The final design for the prototype cell looks something like this:



Because this is a custom designed cell, you can no longer use `UITableViewCell`'s `textLabel` and `detailTextLabel` properties to put text into the labels. These properties refer to labels that aren't on this cell anymore; they are only valid for the standard cell types. Instead, you will use tags to find the labels.

Give the Name label tag 100, the Game label tag 101, and the Image View tag 102. You can do this in the Attributes inspector.

Then open **PlayersViewController.m** and change `cellForRowAtIndexPath` to:

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

    Player *player = (self.players)[indexPath.row];

    UILabel *nameLabel = (UILabel *)[cell viewWithTag:100];
    nameLabel.text = player.name;

    UILabel *gameLabel = (UILabel *)[cell viewWithTag:101];
    gameLabel.text = player.game;

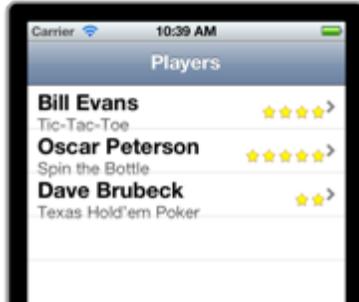
    UIImageView * ratingImageView = (UIImageView *)
        [cell viewWithTag:102];
    ratingImageView.image = [self imageForRating:player.rating];

    return cell;
}
```

This uses a new method, `imageForRating`:

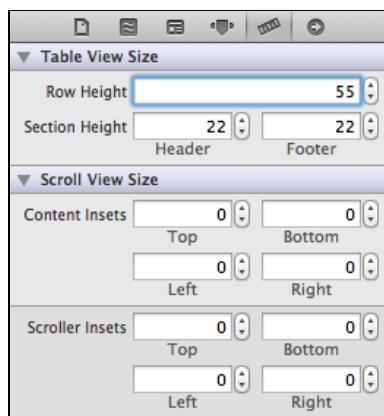
```
- (UIImage *)imageForRating:(int)rating
{
    switch (rating)
    {
        case 1: return [UIImage imageNamed:@"1StarSmall"];
        case 2: return [UIImage imageNamed:@"2StarsSmall"];
        case 3: return [UIImage imageNamed:@"3StarsSmall"];
        case 4: return [UIImage imageNamed:@"4StarsSmall"];
        case 5: return [UIImage imageNamed:@"5StarsSmall"];
    }
    return nil;
}
```

That should do it. Now run the app again. It is possible that the app shows up like this:



Hmm, that doesn't look quite right, the cells appear to overlap one another. You did change the height of the prototype cell but the table view doesn't necessarily take that into consideration. There are two ways to fix it: you can change the table view's Row Height attribute or implement the `heightForRowAtIndexPath` method. The former is much easier, so let's do that.

Back in **MainStoryboard.storyboard**, in the Size inspector of the Table View, set Row Height to 55:



If you run the app now, it looks a lot better!

By the way, if you changed the height of the cell by dragging its handle rather than typing in the value, then the table view's Row Height property was automatically changed already. So it may have worked correctly for you the first time around.

Note: You would use `heightForRowAtIndexPath` if you did not know the height of your cells in advance, or if different rows can have different heights.

Using a Subclass for the Cell

The table view already works pretty well but I'm not a big fan of using tags to access the labels and other subviews of the prototype cell. It would be much more handy if you could connect these labels to outlets and then use the corresponding properties. As it turns out, you can.

Add a new file to the project, with the **Objective-C class** template. Name it **PlayerCell** and make it a subclass of **UITableViewCell**.

Change **PlayerCell.h** to:

```
@interface PlayerCell : UITableViewCell

@property (nonatomic, weak) IBOutlet UILabel *nameLabel;
@property (nonatomic, weak) IBOutlet UILabel *gameLabel;
@property (nonatomic, weak) IBOutlet UIImageView
    *ratingImageView;

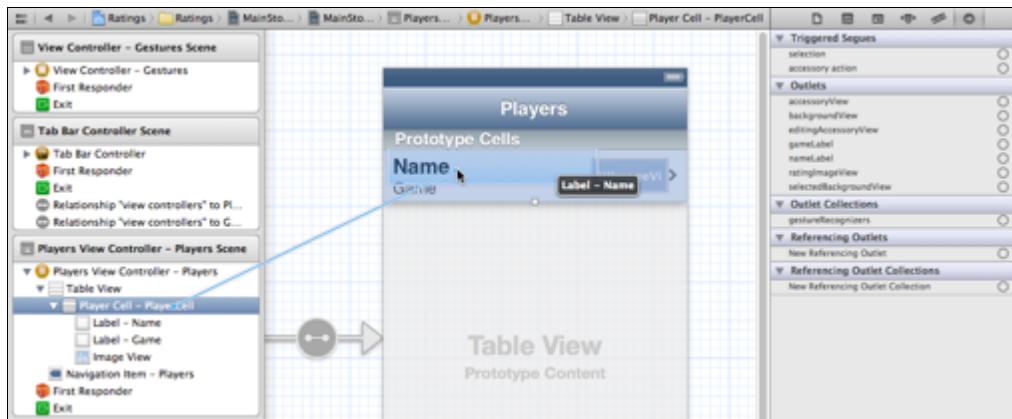
@end
```

The class itself doesn't do much, it just adds properties for `nameLabel`, `gameLabel` and `ratingImageView`, all of which are `IBOutlets`.

Back in **MainStoryboard.storyboard**, select the prototype cell and change its Class to **PlayerCell** on the Identity inspector. Now whenever you ask the table view for a new cell with `dequeueReusableCellWithIdentifier:`, it returns a `PlayerCell` instance instead of a regular `UITableViewCell`.

Note that you gave this class the same name as its reuse identifier – they're both called "PlayerCell" – but whether you want to do that is a matter of personal preference (I like to keep things consistent). The class name and reuse identifier have nothing to do with each other, so you could name them differently if you wanted to.

Now connect the labels and the image view to these outlets. Either select the label and drag from its Connections inspector to the table view cell, or do it the other way around, ctrl-drag from the table view cell back to the label:



Important: You should hook up the controls to the table view cell, not to the view controller! You see, whenever your data source asks the table view for a new cell with `dequeueReusableCellWithIdentifier`, the table view doesn't give you the actual prototype cell but a *copy* (or one of the previous cells is recycled if possible).

This means there will be more than one instance of `PlayerCell` at any given time. If you were to connect a label from the cell to an outlet on the view controller, then several copies of the label will try to use the same outlet. That's just asking for trouble. (On the other hand, connecting the prototype cell to actions on the view controller is perfectly fine. You would do that if you had custom buttons or other `UIControls` on your cell that can trigger actions.)

Now that you've hooked up the properties, you can simplify the data source code one more time. First import the `PlayerCell` class in **PlayersViewController.m**:

```
#import "PlayerCell.h"
```

And then change `cellForRowAtIndexPath` to:

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

    Player *player = (self.players)[indexPath.row];

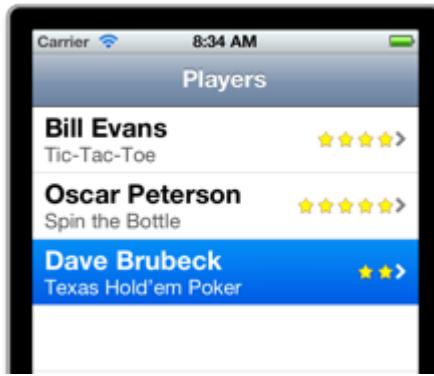
    cell.nameLabel.text = player.name;
    cell.gameLabel.text = player.game;
    cell.ratingImageView.image = [self
        imageForRating:player.rating];
```

```
    return cell;
}
```

That's more like it. You now cast the object that you receive from `dequeueReusableCellWithIdentifier:` to a `PlayerCell`, and then you can simply use the properties that are wired up to the labels and the image view. Isn't it great how using prototype cells makes table views a whole lot less messy!

Run the app and try it out. When you run the app it should still look the same as before, but behind the scenes it's now using your own table view cell subclass!

Here are some free design tips. There are a couple of things you need to take care of when you design your own table view cells. First off, you should set the `Highlighted color` attribute of the labels so that they look good when the user taps the row:

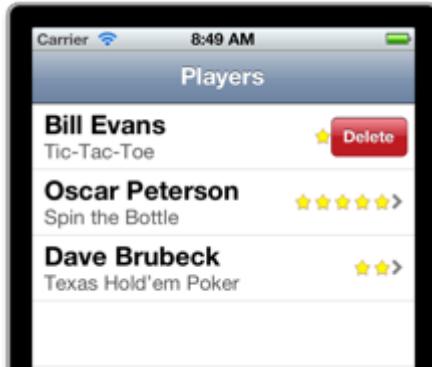


Second, you should make sure that the content you add is flexible so that when the table view cell resizes, the content sizes along with it. Cells will resize when you add the ability to delete or move rows, for example.

Add the following method to `PlayersViewController.m`:

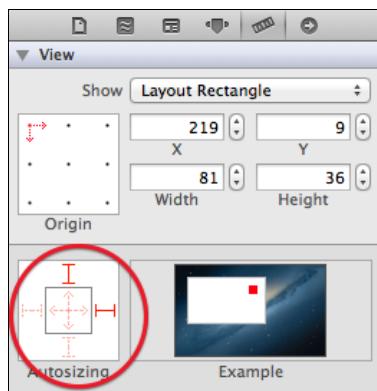
```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        [self.players removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:@[indexPath]
                           withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

When this method is present, swipe-to-delete is enabled on the table. Run the app and swipe a row to see what happens.

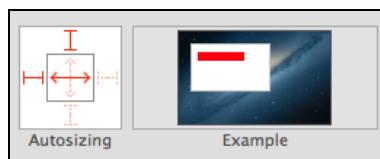


The Delete button slides into the cell but partially overlaps the stars image. What actually happens is that the cell resizes to make room for the Delete button, but the image view doesn't follow along.

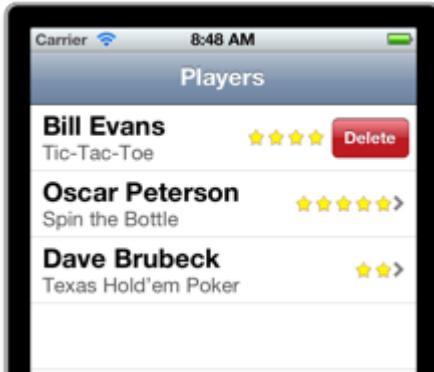
To fix this, open **MainStoryboard.storyboard**, select the image view in the table view cell, and in the Size inspector change the Autosizing so it sticks to its superview's right edge:



Autosizing for the labels should be set up as follows, so they'll shrink when the cell shrinks:



With those changes, the Delete button appears to push aside the stars:



You could also make the stars disappear altogether to make room for the Delete button, but that's left as an exercise for the reader. The important point is that you should keep these details in mind when you design your own table view cells.

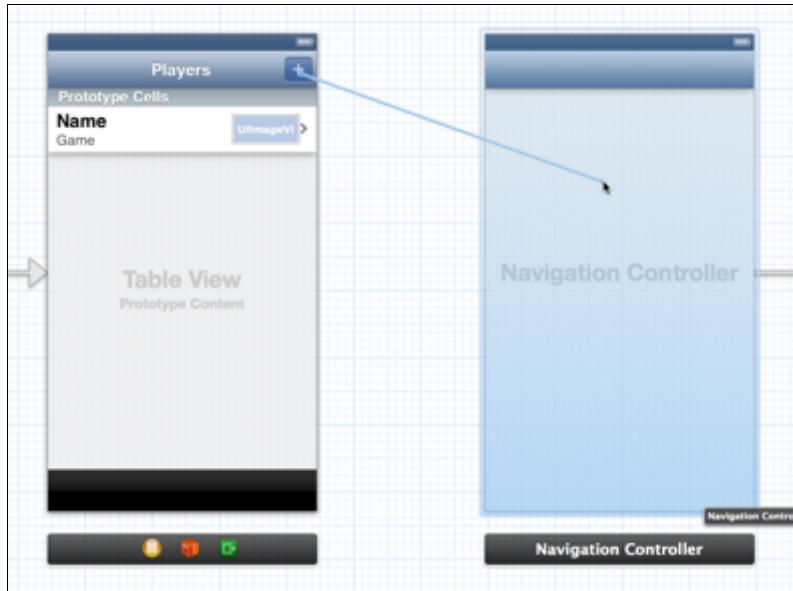
Segues

It's time to add more view controllers to the storyboard. You're going to create a screen that allows users to add new players to the app.

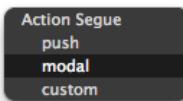
Drag a Bar Button Item into the right slot of the navigation bar on the Players screen. In the Attributes inspector change its Identifier to Add to make it a standard + button. When you tap this button the app will make a new modal screen pop up that lets you enter the details for a new player.

Drag a new Navigation Controller into the canvas, to the right of the Players screen. Remember that you can double-click the canvas to zoom out so you have more room to work with. This new Navigation Controller comes with a Table View Controller attached, so that is handy.

Here's the trick: Select the + button that you just added on the Players screen and ctrl-drag to the new Navigation Controller:



Release the mouse button and a small popup menu shows up:



Choose Modal. This places a new arrow between the Players screen and the Navigation Controller:

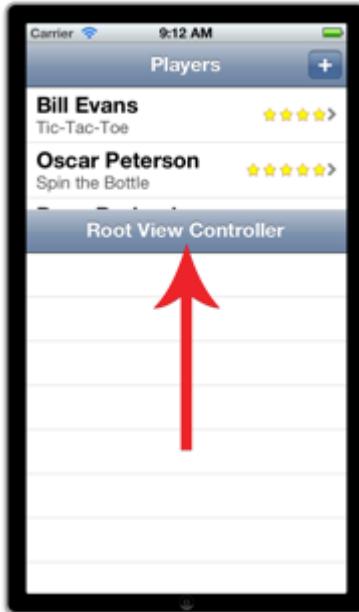


This type of connection is known as a *segue* (pronounce: seg-way) and represents a transition from one screen to another. The storyboard connections you've seen so far were relationships and they described one view controller containing another. A segue, on the other hand, changes what is on the screen. They are triggered by taps on buttons, table view cells, gestures, and so on.

The cool thing about using segues is that you no longer have to write any code to present the new screen, nor do you have to hook up your buttons to `IBActions`.

What you just did, dragging from the Bar Button Item to the next screen, is enough to create the transition. (Note: If your control already had an `IBAction` connection, then the segue overrides that.)

Run the app and press the + button. A new table view will slide up the screen!



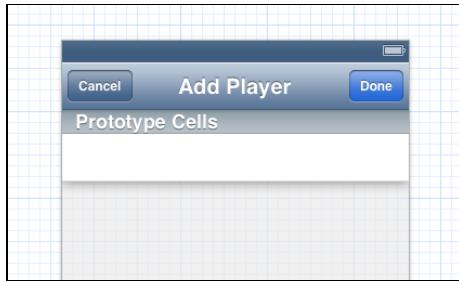
This is a so-called “modal” segue. The new screen completely obscures the previous one. The user cannot interact with the underlying screen until they close the modal screen first. Later on you’ll also see “push” segues that push new screens on the navigation stack of a Navigation Controller.

The new screen isn’t very useful yet – you can’t even close it to go back to the main screen!

Segues only go one way, from the Players screen to this new one. To go back, you have to use the delegate pattern. For that, you first have to give this new scene its own class. Add a new file to the project and name it `PlayerDetailsViewController`, subclass of `UITableViewController`.

To hook this new class up to the storyboard, switch back to **MainStoryboard**, select the new Table View Controller scene and in the Identity inspector set its Class to **PlayerDetailsViewController**. I always forget this very important step, so to make sure you don’t, I’ll keep pointing it out.

While you’re there, change the title of the screen to “Add Player” (by double-clicking in the navigation bar). Also add two Bar Button Items to the navigation bar. In the Attributes inspector, set the Identifier of the button to the left to Cancel, and the one on the right to Done.



Then change **PlayerDetailsViewController.h** to the following:

```
@class PlayerDetailsViewController;

@protocol PlayerDetailsViewControllerDelegate <NSObject>
- (void)playerDetailsViewControllerDidCancel:
    (PlayerDetailsViewController *)controller;
- (void)playerDetailsViewControllerDidSave:
    (PlayerDetailsViewController *)controller;
@end

@interface PlayerDetailsViewController : UITableViewController

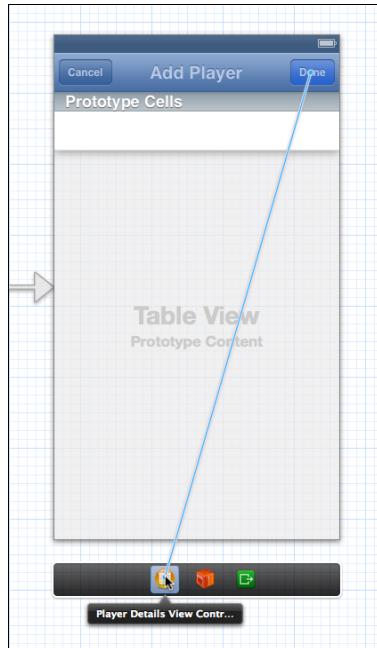
@property (nonatomic, weak) id
    <PlayerDetailsViewControllerDelegate> delegate;

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

@end
```

This defines a new delegate protocol that you'll use to communicate back from the Add Player screen to the main Players screen when the user taps Cancel or Done.

Switch back to Interface Builder, and hook up the Cancel and Done buttons to their respective action methods. One way to do this is to ctrl-drag from the bar button to the view controller and then picking the correct action name from the popup menu:



Make sure you pick the `done:` and `cancel:` actions from the Sent Actions section of the popup menu – don't create a new segue!

In **PlayerDetailsViewController.m**, add the following two methods at the bottom of the file:

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

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

These are the action methods for the two bar buttons. For now, they simply let the delegate know what just happened. It's up to the delegate to actually close the screen. (That is not a requirement, but that's how I like to do it. Alternatively, you could make the Add Player screen close itself before or after it has notified the delegate.)

Note: It is customary for delegate methods to include a reference to the object in question as their first (or only) parameter, in this case the `PlayerDetailsView-Controller`. That way the delegate always knows which object sent the message.

Now that you've given the `PlayerDetailsViewController` a delegate protocol, you still need to implement that protocol somewhere. Obviously, that will be in `PlayersViewController` since that is the view controller that presents the Add Player screen. Add the following to **PlayersViewController.h**:

```
#import "PlayerDetailsViewController.h"

@interface PlayersViewController : UITableViewController
    <PlayerDetailsViewControllerDelegate>

. . .
```

And to the end of **PlayersViewController.m**:

```
#pragma mark - PlayerDetailsViewControllerDelegate

- (void)playerDetailsViewControllerDidCancel:
    (PlayerDetailsViewController *)controller
{
    [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)playerDetailsViewControllerDidSave:
    (PlayerDetailsViewController *)controller
{
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

Currently these delegate methods simply close the screen. Later you'll make them do more interesting things.

The `dismissViewControllerAnimated:completion:` method is new in iOS 5. You may have used `dismissModalViewControllerAnimated:` before. That method still works but the new version is the preferred way to dismiss view controllers from now on (it also gives you the handy ability to execute additional code after the screen has been dismissed).

There is only one thing left to do to make all of this work: the Players screen has to tell the `PlayerDetailsViewController` that it is now its delegate. That seems like something you could set up in Interface Builder just by dragging a line between the two. Unfortunately, that is not possible. To pass data to the new view controller during a segue, you still need to write some code.

Add the following method to **PlayersViewController.m** (it doesn't really matter where):

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender
```

```
{  
    if ([segue.identifier isEqualToString:@"AddPlayer"])  
    {  
        UINavigationController *navigationController =  
            segue.destinationViewController;  
  
        PlayerDetailsViewController *playerDetailsViewController  
            = [navigationController viewControllers][0];  
  
        playerDetailsViewController.delegate = self;  
    }  
}
```

The `prepareForSegue` method is invoked whenever a segue is about to take place. The new view controller has been loaded from the storyboard at this point but it's not visible yet, and you can use this opportunity to send data to it.

Note: You never call `prepareForSegue` yourself, it's a message from UIKit to let you know that a segue has just been triggered.

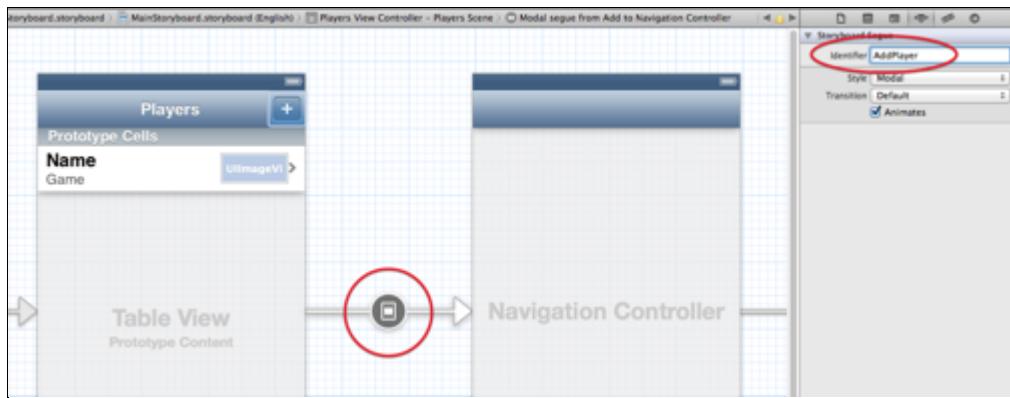
The destination of this particular segue is the Navigation Controller, because that is what you connected to the Bar Button Item. To get the `PlayerDetailsViewController` instance, you have to dig through the Navigation Controller's array of `viewControllers` to find it.

Run the app, press the + button, and try to close the Add Player screen. It still doesn't work!

That's because you never gave the segue an identifier. The code from `prepareForSegue` checks for that identifier ("AddPlayer"). It is recommended to always do such a check because you may have multiple outgoing segues from one view controller and you'll need to be able to distinguish between them (something that you'll do later in these chapters).

To fix this issue, open **MainStoryboard.storyboard** and click on the segue between the Players screen and the Navigation Controller. Notice that the Bar Button Item now lights up, so you can easily see which control triggers this segue.

In the Attributes inspector, set Identifier to **AddPlayer**:



If you run the app again, tapping Cancel or Done will now properly close the screen and return you to the list of players.

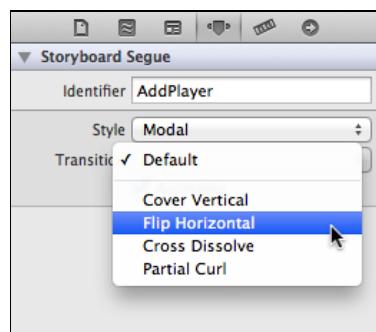
Note: It is perfectly possible to call `dismissViewControllerAnimated:completion:` from the modal screen. There is no requirement that says the delegate must do this. I personally prefer to let the delegate handle this but if you want the modal screen to close itself, then go right ahead.

There's one thing you should be aware of. If you previously used,

```
[self.parentViewController
    dismissModalViewControllerAnimated:YES]
```

to close the screen, then that may no longer work because the "parent" view controller now means something else. Instead of using `self.parentViewController`, simply call the method on `self` or on `self.presentingViewController`, which is a new property that was introduced with iOS 5.

By the way, the Attributes inspector for the segue also has a Transition field. You can choose different animations:



Play with them to see which one you like best. Don't change the Style setting, though. For this screen it should be Modal – any other option will crash the app!

You'll be using the delegate pattern a few more times in this tutorial. Here's a handy checklist for setting up the connections between two scenes:

4. Create a segue from a button or other control on the source scene to the destination scene. (If you're presenting the new screen modally, then often the destination will be a Navigation Controller.)
5. Give the segue a unique Identifier. (It only has to be unique in the source scene; different scenes can use the same identifier.)
6. Create a delegate protocol for the destination scene.
7. Call the delegate methods from the Cancel and Done buttons, and at any other point your destination scene needs to communicate with the source scene.
8. Make the source scene implement the delegate protocol. It should dismiss the destination view controller when Cancel or Done is pressed.
9. Implement the `prepareForSegue` method in the source view controller and do `destination.delegate = self;`.

Delegates are necessary because on iOS 5 there is no such thing as a "reverse segue". When a segue is triggered it always creates a new instance of the destination view controller. You can certainly make a segue back from the destination to the source, but that may not do what you expect.

If you were to make a segue back from the Cancel button to the Players screen, for example, then that wouldn't close the Add Player screen and return to Players, but it creates a new instance of the Players screen. You've started an infinite cycle of creating new view controllers over and over that only ends when the app runs out of memory.

Remember: Segues only go one way, they are only used to open a new screen. To go back you dismiss the screen (or pop it from the navigation stack), usually from a delegate. The segue is employed by the source controller only, the destination view controller doesn't even know that it was invoked by a segue.

Note: Does creating a delegate protocol for each screen that you want to reach through a segue sound like a lot of work? That's what the creators of UIKit thought too, so in iOS 6 they introduced a new concept: the unwind segue. With this new feature you can create segues that close the screen and go back to the previous one. That is what the green Exit icon is for in the storyboard. Unfortunately, unwind segues are an iOS 6-only feature – it won't work on iOS 5 and therefore isn't covered in this book. If you want to learn more about unwind segues, we've dedicated a chapter to it in *iOS 6 by Tutorials*.

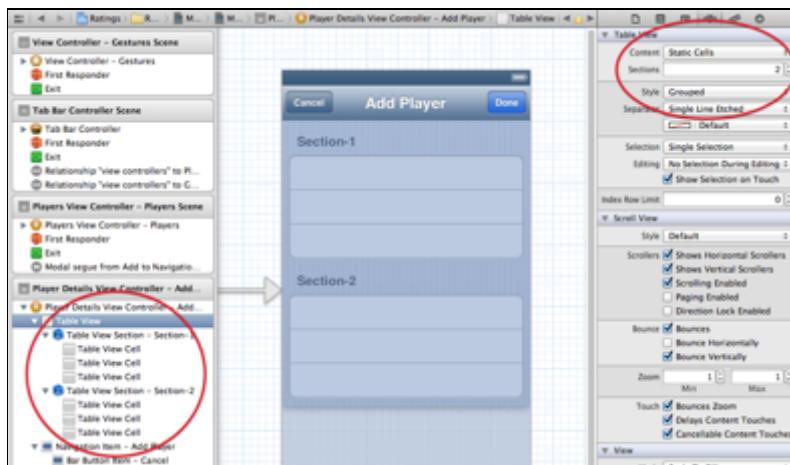
Static Cells

When you're finished with this section, the Add Player screen will look like this:



That's a grouped table view, of course, but the new thing is that you don't have to create a data source for this table. You can design it directly in Interface Builder – no need to write `cellForRowAtIndexPath` for this one. The new feature that makes this possible is *static cells*.

Select the table view in the Add Player screen and in the Attributes inspector change Content to Static Cells. Set Style to Grouped and Sections to 2.



The Add Player screen will have only one row in each section, so select the superfluous cells and delete them.

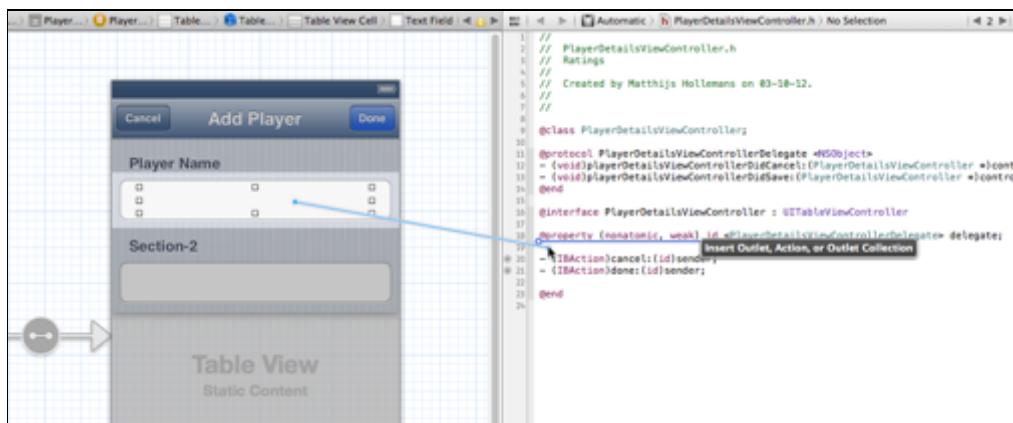
Select the top-most section. In its Attributes inspector, give the Header field the value "Player Name".



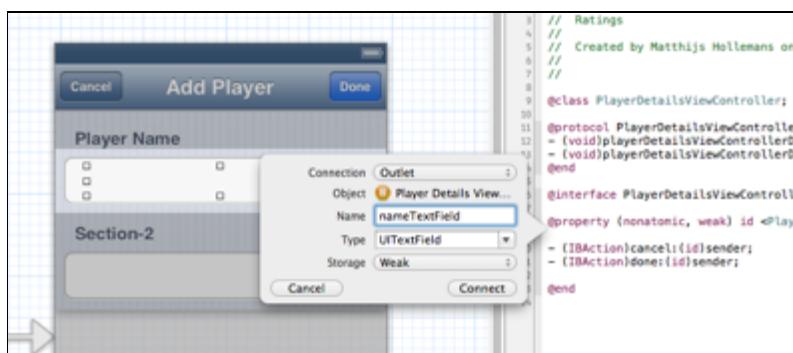
Drag a new Text Field into the cell for this section. Remove its border so you can't see where the text field begins or ends. Set the Font to System 17 and uncheck Adjust to Fit.

You're going to make an outlet for this text field on the `PlayerDetailsViewController` using the Assistant Editor feature of Xcode. Open the **Assistant Editor** with the button from the toolbar (the one that looks like a tuxedo / alien face). It should automatically open on `PlayerDetailsViewController.h`.

Select the text field and ctrl-drag into the .h file:



Let go of the mouse button and a popup appears:



Name the new outlet `nameTextField` and choose Weak for storage. After you click Connect, Xcode will add the following property to **PlayerDetailsViewController.h**:

```
@property (weak, nonatomic) IBOutlet UITextField *nameTextField;
```

Note: When you make connections with the Assistant editor, the current version of Xcode also adds a `viewDidUnload` method to the .m file that sets the new outlet property to `nil`. This is no longer required. In fact, `viewDidUnload` is deprecated as of iOS 6 and Apple recommends that you no longer use it, even for iOS 5 apps. Feel free to remove this method from the code.

Creating outlets for views on your table cells is exactly the kind of thing I said you shouldn't try with prototype cells, but for static cells it is OK. There will be only one instance of each static cell (unlike prototype cells, they are never copied) and so it's perfectly acceptable to connect their subviews to outlets on the view controller.

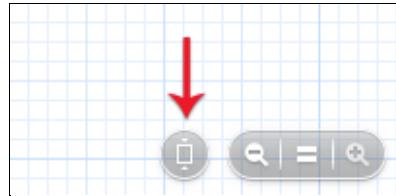
Set the Style of the static cell in the second section to Right Detail. This gives you a standard cell style to work with. Change the label on the left to read "Game" and give the cell a disclosure indicator accessory. Make an outlet for the label on the right (the one that says "Detail") and name it `detailLabel`. The labels on this cell are just regular `UILabel` objects. Remove the header text for the section.

The final design of the Add Player screen looks like this:



Tip: The screens you have designed so far in this storyboard all have the height of the 4-inch screen of the iPhone 5, which is 568 points tall as opposed to the 480 points of the previous iPhone models. You can toggle between these two form

factors using the left-most button from the little floating panel that sits at the bottom of the canvas.



Obviously, your app should work properly with both screen sizes. You can accomplish this with autosizing masks or the new Auto Layout technology from iOS 6. For the Ratings app, you don't have to do anything fancy. It only uses table view controllers and they automatically resize to fit the extra screen space on the iPhone 5.

Back to the Add Player screen. When you use static cells, your table view controller does not need a data source. Because you used an Xcode template to create the `PlayerDetailsViewController` class, it still has some placeholder code for the data source and that will prevent the static cells from working properly. Delete anything between the lines,

```
#pragma mark - Table view data source
```

and:

```
#pragma mark - Table view delegate
```

That should silence Xcode about the warnings it has been giving ever since you added this class to the project.

Run the app and check out the new screen with the static cells. All without writing a line of code – in fact, you threw away a bunch of code!

You can't avoid writing code altogether, though. When you dragged the text field into the first cell, you probably noticed it didn't fit completely, there is a small margin of space around the text field. The user can't see where the text field begins or ends, so if they tap in the margin and the keyboard doesn't appear, they'll be confused.

To avoid that, you should let a tap anywhere in that row bring up the keyboard. That's pretty easy to do, just add or replace the `tableView:didSelectRowAtIndexPath:` method with the following:

```
- (void)tableView:(UITableView *)tableView  
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    if (indexPath.section == 0)  
        [self.nameTextField becomeFirstResponder];
```

```
}
```

This just says that if the user tapped in the first cell, the app should activate the text field (there is only one cell in the section so you only need to test for the section index). Making the text field the “first responder” will automatically bring up the keyboard. It’s just a little tweak, but one that can save users a bit of frustration.

You should also set the Selection Style for the cell to None (instead of Blue) in the Attributes inspector, otherwise the row becomes blue if the user taps in the margin around the text field.

All right, that’s the design of the Add Player screen. Now let’s actually make it work.

The Add Player Screen At Work

For now you will ignore the Game row and just let users enter the name of the player, nothing more.

When the user presses the Cancel button the screen should close and whatever data they entered will be lost. That part already works. The delegate (the Players screen) receives the “did cancel” message and simply dismisses the view controller.

When the user presses Done, however, you should create a new `Player` object and fill in its properties. Then you should tell the delegate that you’ve added a new player, so it can update its own screen.

Inside **PlayerDetailsViewController.m**, first add an import:

```
#import "Player.h"
```

Then change the `done:` method to:

```
- (IBAction)done:(id)sender
{
    Player *player = [[Player alloc] init];
    player.name = self.nameTextField.text;
    player.game = @"Chess";
    player.rating = 1;
    [self.delegate playerDetailsViewController:self
                                         didAddPlayer:player];
}
```

The `done:` method now creates a new `Player` instance and sends it to the delegate. The delegate protocol currently doesn’t have this method, so change its definition in **PlayerDetailsViewController.h** file to:

```

@class PlayerDetailsViewController;
@class Player;

@protocol PlayerDetailsViewControllerDelegate <NSObject>

- (void)playerDetailsViewControllerDidCancel:
    (PlayerDetailsViewController *)controller;

- (void)playerDetailsViewController:
    (PlayerDetailsViewController *)controller
didAddPlayer:(Player *)player;

@end

```

The “didSave” method declaration is gone. Instead, there is now a “didAddPlayer”.

The last thing to do is to add the implementation for this method in **PlayersViewController.m**:

```

- (void)playerDetailsViewController:
    (PlayerDetailsViewController *)controller
didAddPlayer:(Player *)player
{
    [self.players addObject:player];

    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:
        ([self.players count] - 1) inSection:0];
    [self.tableView insertRowsAtIndexPaths:@[indexPath]
        withRowAnimation:UITableViewRowAnimationAutomatic];

    [self dismissViewControllerAnimated:YES completion:nil];
}

```

This first adds the new `Player` object to the array of players. Then it tells the table view that a new row was added (at the bottom), because the table view and its data source must always be in sync.

You could have just done a `[self.tableView reloadData]` but it looks nicer to insert the new row with an animation. `UITableViewRowAnimationAutomatic` is a new constant in iOS 5 that automatically picks the proper animation, depending on where you insert the new row, very handy.

Try it out, you should now be able to add new players to the list!

If you’re wondering about performance of these storyboards, then you should know that loading a whole storyboard at once isn’t a big deal. The Storyboard doesn’t instantiate all the view controllers right away, only the initial view controller. Because your initial view controller is a Tab Bar Controller, the two view controllers

that it contains are also loaded (the Players scene from the first tab and the scene from the second tab).

The other view controllers are not instantiated until you segue to them. When you close these view controllers they are immediately deallocated again, so only the actively used view controllers are in memory, just as if you were using separate nibs.

Let's see that in practice. Add these methods to **PlayerDetailsViewController.m**:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        NSLog(@"init PlayerDetailsViewController");
    }
    return self;
}

- (void)dealloc
{
    NSLog(@"dealloc PlayerDetailsViewController");
}
```

You're overriding the `initWithCoder:` and `dealloc` methods and making them log a message to the Xcode Debug pane. Now run the app again and open the Add Player screen. You should see that this view controller did not get allocated until that point.

When you close the Add Player screen, either by pressing Cancel or Done, you should see the `NSLog` from `dealloc`. If you open the screen again, you should also see the message from `initWithCoder:` again. This should reassure you that view controllers are loaded on-demand only, just as they would if you were loading nibs by hand.

One more thing about static cells, they only work in `UITableViewController`. Even though Interface Builder will let you add them to a Table View object inside a regular `UIViewController`, this won't work during runtime. The reason for this is that `UITableViewController` provides some extra magic to take care of the data source for the static cells. Xcode even prevents you from compiling such a project with the error message: "Illegal Configuration: Static table views are only valid when embedded in `UITableViewController` instances".

Prototype cells, on the other hand, work just fine in a table view that you place inside a regular view controller. Neither work for nibs, though. At the moment, if you want to use prototype cells or static cells, you'll have to use a storyboard.

It is not unthinkable that you might want to have a single table view that combines both static cells and regular dynamic cells, but this isn't very well supported by the

SDK. If this is something you need to do in your own app, then see this post on the Apple Developer Forums [<https://devforums.apple.com/message/505098>] for a possible solution.

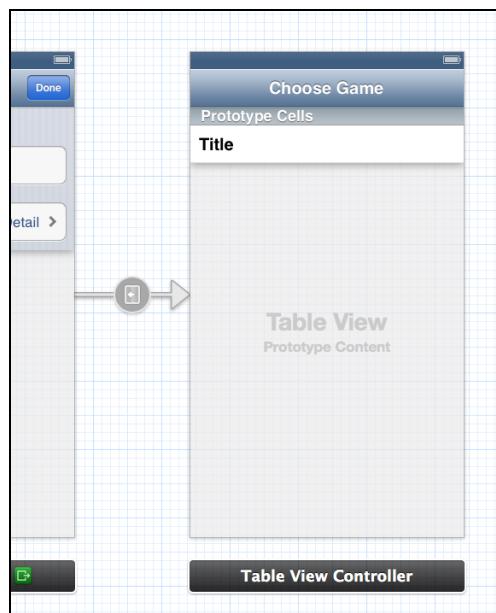
Note: If you're building a screen that has a lot of static cells – more than can fit in the visible frame – then you can scroll through them in Interface Builder with the scroll gesture on the mouse or trackpad (2 finger swipe). This might not be immediately obvious, but it works quite well.

The Game Picker Screen

Tapping the Game row in the Add Player screen should open a new screen that lets the user pick a game from a list. That means you'll be adding yet another table view controller, although this time you're going to push it on the navigation stack rather than show it modally.

Drag a new Table View Controller into the storyboard. Select the Game table view cell in the Add Player screen (be sure to select the entire cell, not one of the labels) and ctrl-drag to the new Table View Controller to create a segue between them. Make this a Push segue (under Selection Segue in the popup, not Accessory Action) and give it the identifier "PickGame".

Double-click the navigation bar and name this new scene "Choose Game". Set the Style of the prototype cell to Basic, and give it the reuse identifier **GameCell**. That's all you need to do for the design of this screen:



Add a new file to the project and name it **GamePickerController**, subclass of **UITableViewController**. Don't forget to set the Class in the storyboard so that your new GamePickerController object is associated with the Table View Controller.

First let's give this new screen some data to display. Add a new instance variable to **GamePickerController.m**:

```
@implementation GamePickerController
{
    NSArray *_games;
}
```

Fill up this array in `viewDidLoad`:

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

    _games = @[@"Angry Birds",
               @"Chess",
               @"Russian Roulette",
               @"Spin the Bottle",
               @"Texas Hold'em Poker",
               @"Tic-Tac-Toe"];
}
```

Note: This uses the new notation for creating array literals `@[...]` that was added to Objective-C recently. Another timesaver!

Replace the data source methods from the template with:

```
#pragma mark - Table view data source

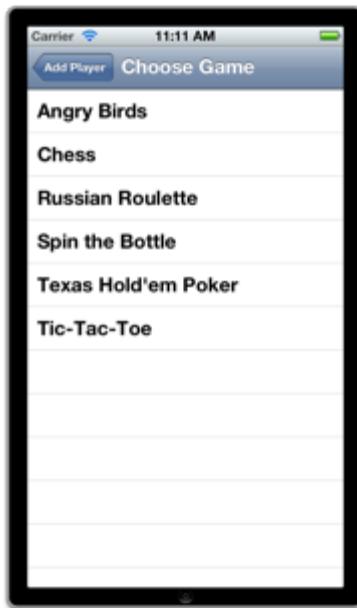
- (NSInteger)numberOfSectionsInTableView:
    (UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [_games count];
```

```
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"GameCell"];
    cell.textLabel.text = _games[indexPath.row];
    return cell;
}
```

That should do it as far as the data source is concerned. Run the app and tap the Game row. The new Choose Game screen will slide into view. Tapping the rows won't do anything yet, but because this screen is presented on the navigation stack you can always press the back button to return to the Add Player screen.



This is pretty cool, huh? You didn't have to write any code to invoke this new screen. You just ctrl-dragged from the static table view cell to the new scene and that was it.

Important: The table view delegate method `didSelectRowAtIndexPath` in `PlayerDetailsViewController` is still called when you tap the Game row, so make sure you don't do anything there that will conflict with the segue.

Of course, this new screen isn't very useful if it doesn't send any data back, so you'll have to add a new delegate for that. Replace `GamePickerController.h` with:

```

@class GamePickerController;

@protocol GamePickerControllerDelegate <NSObject>
- (void)gamePickerController:
    (GamePickerController *)controller
    didSelectGame:(NSString *)game;
@end

@interface GamePickerController : UITableViewController

@property (nonatomic, weak) id
    <GamePickerControllerDelegate> delegate;
@property (nonatomic, strong) NSString *game;

@end

```

You've added a delegate protocol with just one method, and a property that will hold the name of the currently selected game.

Add a new instance variable, `_selectedIndex`, to **GamePickerController.m**:

```

@implementation GamePickerController
{
    NSArray *_games;
    NSUInteger _selectedIndex;
}

```

Then add the following line to the bottom of `viewDidLoad`:

```

_selectedIndex = [_games indexOfObject:self.game];

```

The name of the selected game will be set in `self.game`. Here you figure out what the index is for that game in the list of games. You'll use that index to set a checkmark in the table view cell. For this to work, `self.game` must be filled in before the view is loaded. That will be no problem because you will do this in the caller's `prepareForSegue`, which takes place before `viewDidLoad`.

Change `cellForRowAtIndexPath` to:

```

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"GameCell"];
    cell.textLabel.text = _games[indexPath.row];

    if (indexPath.row == _selectedIndex)

```

```
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    else
        cell.accessoryType = UITableViewCellAccessoryNone;

    return cell;
}
```

This sets a checkmark on the cell that contains the name of the currently selected game. Small gestures such as these will be appreciated by the users of the app.

Replace the placeholder didSelectRowAtIndexPath method from the template with:

```
#pragma mark - Table view delegate

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];

    if (_selectedIndex != NSNotFound)
    {
        UITableViewCell *cell = [tableView
            cellForRowAtIndexPath:[NSIndexPath
                indexPathForRow:_selectedIndex inSection:0]];
        cell.accessoryType = UITableViewCellAccessoryNone;
    }

    _selectedIndex = indexPath.row;
    UITableViewCell *cell = [tableView
        cellForRowAtIndexPath:indexPath];
    cell.accessoryType = UITableViewCellAccessoryCheckmark;

    NSString *game = _games[indexPath.row];
    [self.delegate gamePickerController:self
        didSelectGame:game];
}
```

First this deselects the row after it was tapped. That makes it fade from the blue highlight color back to the regular white. Then it removes the checkmark from the cell that was previously selected, and puts it on the row that was just tapped. Finally, the method returns the name of the chosen game to the delegate.

Run the app now to test that this works. Tap the name of a game and its row will get a checkmark. Tap the name of another game and the checkmark moves along with it. The screen ought to close as soon as you tap a row but that doesn't happen yet because you haven't actually hooked up the delegate.

In **PlayerDetailsViewController.h**, add an import:

```
#import "GamePickerController.h"
```

And add the delegate protocol to the @interface line:

```
@interface PlayerDetailsViewController : UITableViewController  
    <GamePickerControllerDelegate>
```

In **PlayerDetailsViewController.m**, add the `prepareForSegue` method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue  
    sender:(id)sender  
{  
    if ([segue.identifier isEqualToString:@"PickGame"])  
    {  
        GamePickerController *gamePickerController =  
            segue.destinationViewController;  
        gamePickerController.delegate = self;  
        gamePickerController.game = _game;  
    }  
}
```

This is similar to what you did before. This time the destination view controller is the game picker screen. Remember that this happens after `GamePickerController` is instantiated but before its view is loaded.

The `_game` variable is new. This is a new instance variable:

```
@implementation PlayerDetailsViewController  
{  
    NSString *_game;  
}
```

You use this variable to remember the selected game so you can store it in the `Player` object later. It should get a default value. The `initWithCoder:` method is a good place for that:

```
- (id)initWithCoder:(NSCoder *)aDecoder  
{  
    if ((self = [super initWithCoder:aDecoder]))  
    {  
        NSLog(@"init PlayerDetailsViewController");  
        _game = @"Chess";  
    }  
    return self;  
}
```

If you've worked with nibs before, then `initWithCoder:` will be familiar. That part has stayed the same with storyboards; `initWithCoder:`, `awakeFromNib`, and `viewDidLoad` are still the methods to use. You can think of a storyboard as a collection of nibs with additional information about the transitions and relationships between them. But the views and view controllers inside the storyboard are still encoded and decoded in the same way.

Change `viewDidLoad` to display the name of the game in the cell:

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

All that remains is to implement the delegate method:

```
- (void)gamePickerController:
    (GamePickerController *)controller
didSelectGame:(NSString *)game
{
    _game = game;
    self.detailLabel.text = _game;

    [self.navigationController popViewControllerAnimated:YES];
}
```

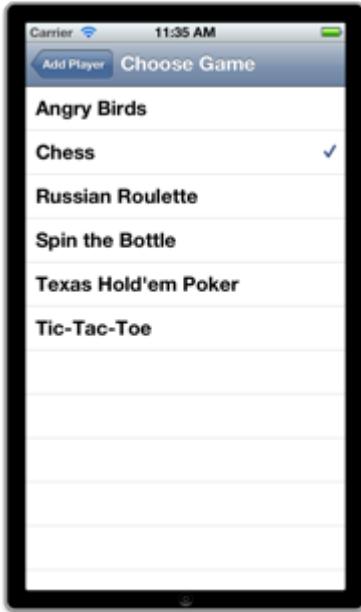
This is pretty straightforward: you put the name of the new game into the `_game` instance variable and also the cell's label, and then close the Choose Game screen. Because it's a push segue, you have to pop this screen off the navigation stack to close it.

The `done:` method can now put the name of the chosen game into the new `Player` object, rather than the hardcoded value you've been using so far:

```
- (IBAction)done:(id)sender
{
    Player *player = [[Player alloc] init];
    player.name = self.nameTextField.text;
    player.game = _game;
    player.rating = 1;

    [self.delegate playerDetailsViewController:self
        didAddPlayer:player];
}
```

Awesome, you now have a functioning Choose Game screen!



Where To Go From Here?

Congratulations, you now know the basics of using storyboards, and can create apps with multiple view controllers transitioning between each other with segues!

If you want to learn more about storyboards in iOS 5, keep reading the next chapter, where we'll cover:

- How to change the `PlayerDetailsViewController` so that it can also edit existing `Player` objects.
- How to have multiple outgoing segues to other scenes, and how to make your view controllers re-usable so they can handle multiple incoming segues.
- How to perform segues from disclosure buttons, gestures, and any other event you can think of.
- How to make custom segues – you don't have to be limited to the standard Push and Modal animations!
- How to use storyboards on the iPad, with a split-view controller and popovers.
- And finally, how to manually load storyboards and use more than one storyboard inside an app.

Chapter 5: Intermediate Storyboards

By Matthijs Hollemans

In the last chapter, you got some basic experience with storyboarding. You learned how to add view controllers into a storyboard, transition between them with segues, and even create custom table view cells quite easily!

In this chapter, you're going to learn even more cool things you can do with storyboards in iOS 5. We'll show you how to modify the app to edit players, add multiple segues between scenes, implement custom segue animations, use storyboards on the iPad, and much more!

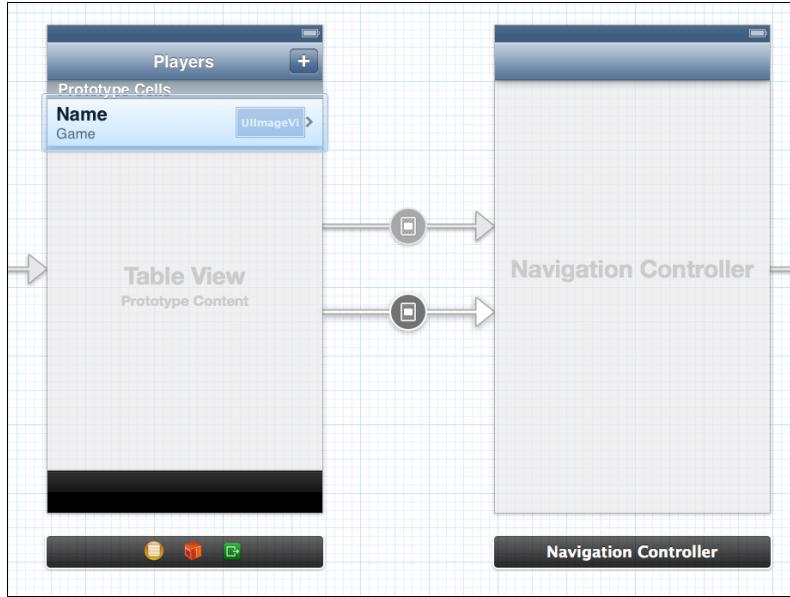
This chapter picks up where you left off last time. Open up your Ratings project in Xcode and let's get started!

Editing Existing Players

It's always a good idea to give users of your app the ability to edit the data they've added. In this section you will extend the `PlayerDetailsViewController` so that besides adding new players it can also edit existing ones.

Ctrl-drag from the prototype cell in the Players screen to the Navigation Controller that is attached to the Add Player screen and add a new modal segue. Name this segue **EditPlayer**.

There are now two segues between these scenes:



It is possible to distinguish between these two segues because you've given them unique names, AddPlayer and EditPlayer. If you get confused as to which one is which, you can simply click on the segue icon and the control that triggers it will be highlighted with a blue box. That is the + button for the AddPlayer segue, and the prototype cell for the EditPlayer segue.

In **PlayersViewController.m**, extend `prepareForSegue:` to:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                  sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"AddPlayer"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        PlayerDetailsViewController *playerDetailsViewController
            = [navigationController viewControllers][0];
        playerDetailsViewController.delegate = self;
    }
    else if ([segue.identifier isEqualToString:@"EditPlayer"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        PlayerDetailsViewController *playerDetailsViewController
            = [navigationController viewControllers][0];
        playerDetailsViewController.delegate = self;
    }

    NSIndexPath *indexPath = [self.tableView
                             indexPathForCell:sender];
}
```

```
        Player *player = self.players[indexPath.row];
        playerDetailsViewController.playerToEdit = player;
    }
}
```

The if-statement that checks for the “EditPlayer” segue is new. What happens is very similar to the “AddPlayer” segue, except that you now pass along a `Player` object in the new `playerToEdit` property.

To find the index-path for the cell that was tapped, you do:

```
NSIndexPath *indexPath = [self.tableView
                        indexPathForCell:sender];
```

The “sender” parameter from `prepareForSegue` contains a pointer to the control that initiated the segue. In the case of the `AddPlayer` segue that is the + `UIBarButtonItem`, but for `EditPlayer` it is a table view cell. Because you put the segue on the prototype cell, it can be triggered from any cell that is copied from the prototype, and you use `sender` to tell apart which particular cell that was.

Add the new `playerToEdit` property to **PlayerDetailsViewController.h**:

```
@property (strong, nonatomic) Player *playerToEdit;
```

In **PlayerDetailsViewController.m**, change `viewDidLoad` to:

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

    if (self.playerToEdit != nil)
    {
        self.title = @"Edit Player";
        self.nameTextField.text = self.playerToEdit.name;
        _game = self.playerToEdit.game;
    }

    self.detailLabel.text = _game;
}
```

If the `playerToEdit` property is set, then this screen no longer functions as the `Add Player` screen but it becomes `Edit Player`. It also fills in the text field and the game label with the values from the existing `Player` object.

Run the app and tap on a player to open the `Edit Player` screen:



Of course, you're not quite done with the changes. If you were to press Done now, a new `Player` object would still be added to the list. You have to change that part of the logic to update the existing `Player` object instead.

Add a new method to the delegate protocol in `PlayerDetailsViewController.h`:

```
- (void)playerDetailsViewController:  
    (PlayerDetailsViewController *)controller  
    didEditPlayer:(Player *)player;
```

Then in `PlayerDetailsViewController.m` change the `done:` action to call this new delegate method when the user is editing an existing player object:

```
- (IBAction)done:(id)sender  
{  
    if (self.playerToEdit != nil)  
    {  
        self.playerToEdit.name = self.nameTextField.text;  
        self.playerToEdit.game = _game;  
  
        [self.delegate playerDetailsViewController:self  
            didEditPlayer:self.playerToEdit];  
    }  
    else  
    {  
        Player *player = [[Player alloc] init];  
        player.name = self.nameTextField.text;  
        player.game = _game;  
        player.rating = 1;  
    }  
}
```

```
[self.delegate playerDetailsViewController:self  
    didAddPlayer:player];  
}  
}
```

Finally, implement the delegate method in **PlayersViewController.m**:

```
- (void)playerDetailsViewController:  
    (PlayerDetailsViewController *)controller  
    didEditPlayer:(Player *)player  
{  
    NSUInteger index = [self.players indexOfObject:player];  
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index  
        inSection:0];  
    [self.tableView reloadRowsAtIndexPaths:@[indexPath]  
        withRowAnimation:UITableViewRowAnimationAutomatic];  
  
    [self dismissViewControllerAnimated:YES completion:nil];  
}
```

This simply reloads the cell for that player so that its labels get updated and then closes the Edit Player screen.

That's all there is to it. With a few small changes you were able to re-use the existing `PlayerDetailsViewController` class to also edit `Player` objects. There are two segues going to this scene, `AddPlayer` and `EditPlayer`, and which mode is used, adding or editing, depends on which segue is triggered.

Remember that performing a segue always creates a new instance of the destination view controller, so if you do Add Player first and then Edit Player some time later, you are interacting with separate instances of this class.

I have mentioned a few times that `prepareForSegue` is called before `viewDidLoad`. Here you use that to your advantage to set the `playerToEdit` property on the destination view controller. In `viewDidLoad` you read the values from `playerToEdit` and put them into the labels.

Note: There is no such thing as a "didPerformFromSegue" method on the destination scene that lets the scene know it was invoked by a segue. In fact, the new view controller doesn't know anything about the segue at all.

To tell the destination view controller that it was launched from a segue, or to perform additional configuration, you'll need to set a property or call a method from `prepareForSegue`.

You can also override the setter for your data object in the destination view controller. For example:

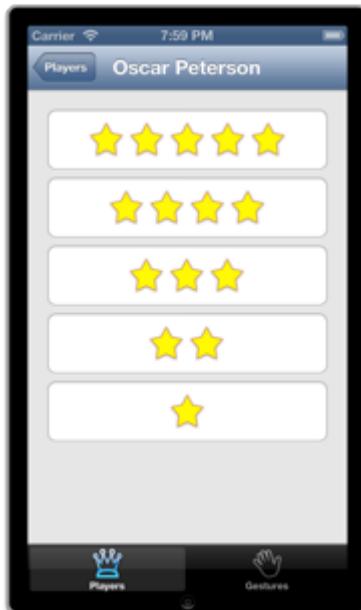
```
- (void)setPlayerToEdit:(Player *)newPlayerToEdit
{
    if (playerToEdit != newPlayerToEdit)
    {
        playerToEdit = newPlayerToEdit;

        // do additional configuration here
        // ...

        self.invokedFromSegue = YES;
    }
}
```

The Rating Screen

The app is called Ratings but so far you haven't done much with those ratings except show a few stars here and there. You will now add a new screen that lets you pick a rating for a player:



Drag a new View Controller into the canvas and put it roughly below the Add Player screen. This is a regular view controller, not a table view controller.

There is a bit of a problem, you need to invoke this new Rate Player screen from the list of players, but tapping a row in that table already brings up the Edit Player screen. So first there needs to be a way to distinguish between rating a player and editing a player.

The way you're going to do it is as follows: tapping a row will now bring up the Rate Player screen, but tapping the detail disclosure button goes to the Edit Player screen.

Select the prototype cell in the Players view controller. Change its accessory to Detail Disclosure so it becomes a blue button instead of just a chevron.

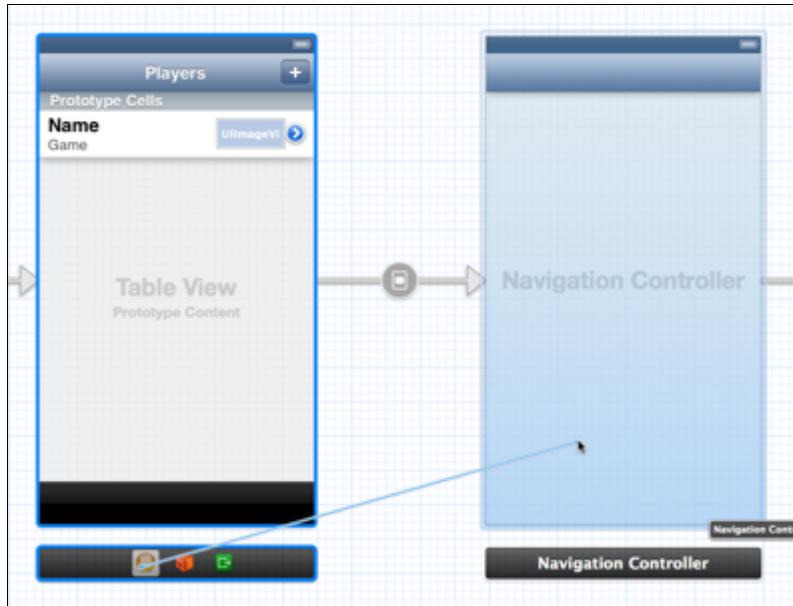
Delete the existing EditPlayer segue. The idea is that you should make a new segue from the detail disclosure button to the Add/Edit Player screen. Here's the rub: making segues from the detail disclosure button is not supported on iOS 5, but as of version 4.5, Xcode does provide a means to do so.

If you ctrl-drag from the table view cell to the Navigation Controller, Xcode gives you the option to create a "Selection Segue" or an "Accessory Action". The last one sounds like what you need, but it works on iOS 6 only and it will crash the app on iOS 5 with the following error message:

```
*** Terminating app due to uncaught exception
'NSUnknownKeyException', reason: '[<PlayerCell 0x6d35530>
setValue:forUndefinedKey:]: this class is not key value coding-
compliant for the key accessoryActionSegueTemplate.'
```

So instead, because this app should be backwards compatible to iOS 5, what you'll do is put the segue on the view controller itself and then trigger it programmatically.

Ctrl-drag from the view controller icon in the dock to the Navigation Controller and add a new Modal segue. As before, give it the identifier **EditPlayer**. Note that this segue is connected to the Players view controller itself, not to any specific control inside it. There is no button or other control that you can tap to trigger the segue.



While you're here, ctrl-drag from the prototype cell to the new view controller that you just added. Make this a push segue named **RatePlayer**. There are now three outgoing segues from the Players screen.

Double-click its navigation bar to title the new screen "Rate Player".

Back to the disclosure button. If you've worked with these before you know that there is a special table view delegate method for handling taps on disclosure buttons. You're going to add this method to **PlayersViewController.m** and trigger the EditPlayer segue manually.

```
- (void)tableView:(UITableView *)tableView
    accessoryButtonTappedForRowWithIndexPath:
    (NSIndexPath *)indexPath
{
    [self performSegueWithIdentifier:@"EditPlayer"
        sender:indexPath];
}
```

That's all you have to do. This will load the `PlayerDetailsViewController` from the storyboard (and the Navigation Controller that contains it), and present it modally on the screen.

Of course, before the new screen is displayed, `prepareForSegue` is still called. You need to make a small change to that method. Previously, the `sender` parameter contained the `UITableViewCell` object that triggered the segue. Now, however, the segue is not being triggered from the table view cell, so instead you're sending along an `NSIndexPath` (because that's what you just put in the `sender` parameter of the call to `performSegueWithIdentifier:sender:`).

In `prepareForSegue` the line,

```
NSIndexPath *indexPath = [self.tableView  
    indexPathForCell:sender];
```

now simply becomes:

```
NSIndexPath *indexPath = sender;
```

If you trigger a segue programmatically by calling `performSegueWithIdentifier`, then you can pass along anything you want as the `sender`. I chose to send the `NSIndexPath` because that was the least amount of work. (You could also have sent the `Player` object from that row, for example, or look up the `UITableViewCell` and send that.)

Run the app. Tapping the blue disclosure button now brings up the Edit Player screen (modally, sliding up from below) and tapping the row slides in the Rate Player screen from the side (pushed on the navigation stack).

Let's finish building the Rate Player screen. Add a new file for a `UIViewController` subclass to the project and name it **RatePlayerViewController** (remember, this is a regular view controller, not a table view controller).

Set the Class for the Rate Player screen in the Identity inspector. This is something I always forget and then I spend five minutes puzzling over why my screen doesn't do the things it's supposed to – until I realize I didn't actually tell the storyboard to use my subclass. Don't be as foolish as me and remember to fill in the Class field!

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

```
@class RatePlayerViewController;  
@class Player;  
  
@protocol RatePlayerViewControllerDelegate <NSObject>  
- (void)ratePlayerViewController:  
    (RatePlayerViewController *)controller  
    didPickRatingForPlayer:(Player *)player;  
@end  
  
@interface RatePlayerViewController : UIViewController  
  
@property (nonatomic, weak) id  
    <RatePlayerViewControllerDelegate> delegate;  
@property (nonatomic, strong) Player *player;  
  
- (IBAction)rateAction:(UIButton *)sender;  
@end
```

This should look familiar by now. Again you're using the delegate pattern to communicate back to the source view controller.

Add an import at the top of **RatePlayerViewController.m**:

```
#import "Player.h"
```

Change viewDidLoad to:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.title = self.player.name;
}
```

This sets the name of the chosen player in the navigation bar title (instead of the text "Rate Player").

The interesting part in this view controller is the rateAction method, so add it next:

```
- (IBAction)rateAction:(UIButton *)sender
{
    self.player.rating = sender.tag;
    [self.delegate ratePlayerViewController:self
                                    didPickRatingForPlayer:self.player];
}
```

This puts the new rating into the `Player` object and then lets the delegate know about it. The rating comes from `sender.tag`, where the sender is a `UIButton`.

What you're going to do is add five `UIButton` objects to the view controller – one star, two stars, three stars, etc. – and give each of them a tag value that corresponds to the number of stars on the button. All of these buttons will be connected to the same action method. That's a quick 'n' easy way to make this work.

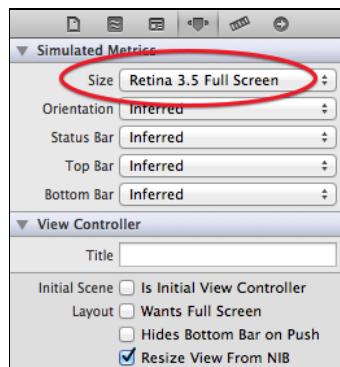
Drag five Buttons into the Rate Player screen and make the layout look like this:



The images for the buttons have already been added to the project (they are inside the Images folder). Use "1Star.png", "2Stars.png", and so on.

Connect each button's Touch Up Inside event to the `rateAction:` method. Set the `tag` attribute for the 5-stars button to 5, for the 4-stars button to 4, etc. The tag value should correspond to the number of stars on the button.

Make the background color for the scene's main view light gray so the buttons stand out a bit more. When placing the buttons, keep in mind that the screen has to work both on the new iPhone 5 and the smaller older models. Under Simulated Metrics, you can toggle the form factor for how this view controller appears in Interface Builder, which will help you make a layout that fits on all screen sizes:



This screen doesn't need Cancel or Done buttons in the navigation bar because it's pushed on the navigation stack.

The final step is to set the delegate so that the buttons actually have somewhere to send their messages. Here are the changes to **PlayersViewController.h**:

```
#import "RatePlayerViewController.h"
```

```
@interface PlayersViewController : UITableViewController
<PlayerDetailsViewControllerDelegate,
RatePlayerViewControllerDelegate>
```

And **PlayersViewController.m**:

```
#pragma mark - RatePlayerViewControllerDelegate

- (void)ratePlayerViewControllerAnimated:
    (RatePlayerViewController *)controller
    didPickRatingForPlayer:(Player *)player
{
    NSUInteger index = [self.players indexOfObject:player];
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index
                                                inSection:0];
    [self.tableView reloadRowsAtIndexPaths:@[indexPath]
        withRowAnimation:UITableViewRowAnimationAutomatic];

    [self.navigationController popViewControllerAnimated:YES];
}
```

Again, you simply redraw the table view cell for the player that was changed, but this time you pop the Rate Player screen off the navigation stack rather than dismissing it.

Of course, you can't forget `prepareForSegue:`

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender
{
    // ...existing code...

    else if ([segue.identifier isEqualToString:@"RatePlayer"])
    {
        RatePlayerViewController *ratePlayerViewControllerAnimated =
            segue.destinationViewController;
        ratePlayerViewControllerAnimated.delegate = self;

        NSIndexPath *indexPath = [self.tableView
            indexPathForCell:sender];
        Player *player = self.players[indexPath.row];
        ratePlayerViewControllerAnimated.player = player;
    }
}
```

Because this scene has three outgoing segues, there are also three if-statements in `prepareForSegue`.

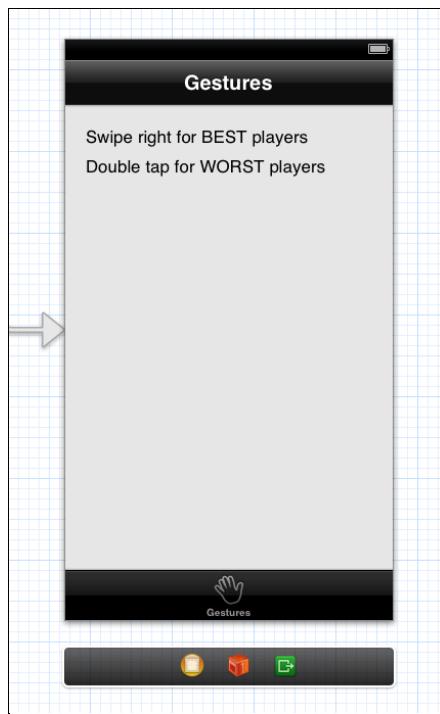
Run the app and verify that you can now pick ratings for the players.

Gestures

We've been neglecting the second tab of the app. Let's give it something to do. The project has a class named `viewController` that was originally generated by the Xcode template, but you haven't used it until now. Using Xcode's **Edit\Refactor\Rename** menu, rename that class to **GesturesViewController**.

In the storyboard, go to the view controller that is hooked up to the second tab and set its Class field to **GesturesViewController**.

Drag in some labels and a Navigation Bar to make it look somewhat like this:



You're not going to push any screens on the navigation stack here, so you don't need to embed this scene inside a Navigation Controller. Just putting a Navigation Bar subview at the top is enough.

As the text on the labels indicates, you're going to add gestures to this screen. A swipe to the right will pop up a new screen that lists all the best players (5 stars); a double-tap will list the worst players (1 star) instead. The app needs some place to list those best/worst players, and you'll add a new table view controller for that.

Drag a Navigation Controller from the Object Library into the canvas and place it to the right of the Gestures screen. This actually gives you two new scenes: the Navigation Controller itself and a default Root View Controller attached to it. Humor me and delete that Root View Controller. There's nothing wrong with it, but I want to show you how you can connect your own view controller to the Navigation Controller.

Now drag a new Table View Controller into the canvas, next to that new Navigation Controller. Ctrl-drag from the Navigation Controller to the Table View Controller and choose **Relationship Segue – root view controller** to connect the two. You may need to reshuffle your scenes a bit to make this all fit.

We will designate this new Table View Controller the Ranking screen. Give it that title in its Navigation Item, so you don't get confused as to which scene is which. The storyboard is already getting quite big!

Back to the Gestures scene. Triggering a segue based on a gesture is actually pretty simple. In the Object Library there are several gesture recognizer objects:

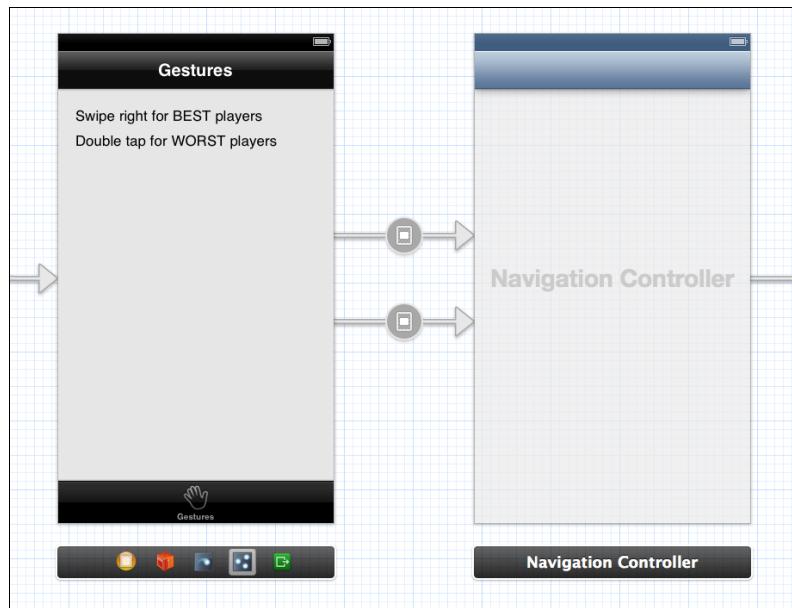


Drag the Swipe Gesture Recognizer into the Gestures screen. This will add an icon for the gesture recognizer to the dock:



Now ctrl-drag from this icon into the Navigation Controller next door and pick the Modal segue option. Give this segue the identifier **BestPlayers**.

Also drag in a Tap Gesture Recognizer. Create a segue for that one too and name it **WorstPlayers**. In the Attributes inspector for the tap gesture recognizer, set the number of taps to 2 so it will detect double taps.



Run the app, perform the gesture, and the segue should either happen – or the app may actually crash. ☹

There is a bug in iOS 5 where adding a gesture recognizer to a view controller inside a Tab Bar Controller crashes the app. As a workaround, you can connect the gesture recognizer objects to a strong outlet in order to keep them alive.

Add the following properties to **GesturesViewController.h**:

```
@interface GesturesViewController : UIViewController

@property (nonatomic, strong) IBOutlet UISwipeGestureRecognizer *swipeGestureRecognizer;
@property (nonatomic, strong) IBOutlet UITapGestureRecognizer *tapGestureRecognizer;

@end
```

In the storyboard, ctrl-drag from the Gestures View Controller to the two gesture recognizer symbols in the dock to connect them to these properties. Try the app again: the crash should be history.

Now that you can make it appear, let's make the Ranking screen do something. Create a new `UITableView` subclass named **RankingViewController**.

Replace **RankingViewController.h** with:

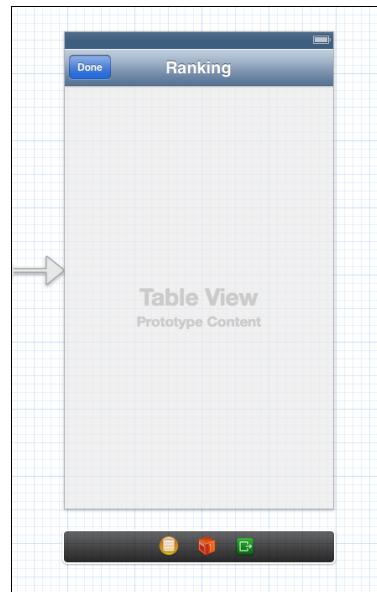
```
@interface RankingViewController : UITableViewController  
  
@property (nonatomic, strong) NSMutableArray *rankedPlayers;  
  
- (IBAction)done:(id)sender;  
  
@end
```

Set the Class for the Ranking screen in the storyboard to **RankingViewController**. Add a Done bar button to its navigation bar and connect it to the `done:` action.

Tip: You can simply ctrl-drag from the Done button to the status bar. That will always select the view controller as the target.

Delete the prototype cell from the table view. For this view controller you're going to build cells the old fashioned way. The old method of making table view cells still works, and you can even combine them with prototype cells if you want to. Some of the cells in your table view can be based on prototype cells while others are old school handmade cells. (Combining static cells with your own is also possible, but kinda tricky.)

The final design of the Ranking screen is about as simple as it gets:



For the changes in **RankingViewController.m**, first import the `Player` header:

```
#import "Player.h"
```

Replace the table view data source methods with the following:

```
#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:
    (UITableView *)tableView
{
    return 1;
}

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

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

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
    {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier];
    }

    Player *player = self.rankedPlayers[indexPath.row];
    cell.textLabel.text = player.name;
    cell.detailTextLabel.text = player.game;

    return cell;
}
```

Finally, add the implementation of the `done:` action method:

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

There is no delegate for this screen. We don't really have anything useful to send back to the view controller that invoked the Ranking screen, so it simply dismisses itself when the user presses Done.

Run the app. You should be able to open and close the Ranking screen, even though it doesn't display anything yet. You still need to give it the list of ranked players.

Add the following property to **GesturesViewController.h**:

```
@property (nonatomic, strong) NSArray *players;
```

GesturesViewController.m needs these additional imports:

```
#import "RankingViewController.h"
#import "Player.h"
```

Add a new `prepareForSegue` method for setting up the segues in response to the gestures:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                 sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"BestPlayers"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        RankingViewController *rankingViewController =
            [navigationController viewControllers][0];
        rankingViewController.rankedPlayers =
            [self playersWithRating:5];
        rankingViewController.title = @"Best Players";
    }
    else if ([segue.identifier isEqualToString:@"WorstPlayers"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        RankingViewController *rankingViewController =
            [navigationController viewControllers][0];
        rankingViewController.rankedPlayers =
            [self playersWithRating:1];
        rankingViewController.title = @"Worst Players";
    }
}
```

For both segues this first gets the Navigation Controller that is on the other end of the segue, and from that you can obtain the `RankingViewController` instance. Then you give it the list of ranked players and a title.

The `playersWithRating:` method is also new:

```

- (NSMutableArray *)playersWithRating:(int)rating
{
    NSMutableArray *rankedPlayers = [NSMutableArray
        arrayWithCapacity:[self.players count]];

    for (Player *player in self.players)
    {
        if (player.rating == rating)
            [rankedPlayers addObject:player];
    }

    return rankedPlayers;
}

```

This simply loops through the list of players and only adds those with the specified rating to a new array.

Now the question is, where does `GesturesViewController` get its own list of players from in the first place? From `AppDelegate`, of course. The App Delegate is the owner of the data model for this app.

Add the following import to `AppDelegate.m`:

```
#import "GesturesViewController.h"
```

Add the following to the bottom of `didFinishLaunchingWithOptions`:

```

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // ...existing code...

    GesturesViewController *gesturesViewController =
        [tabBarController viewControllers][1];
    gesturesViewController.players = _players;

    return YES;
}

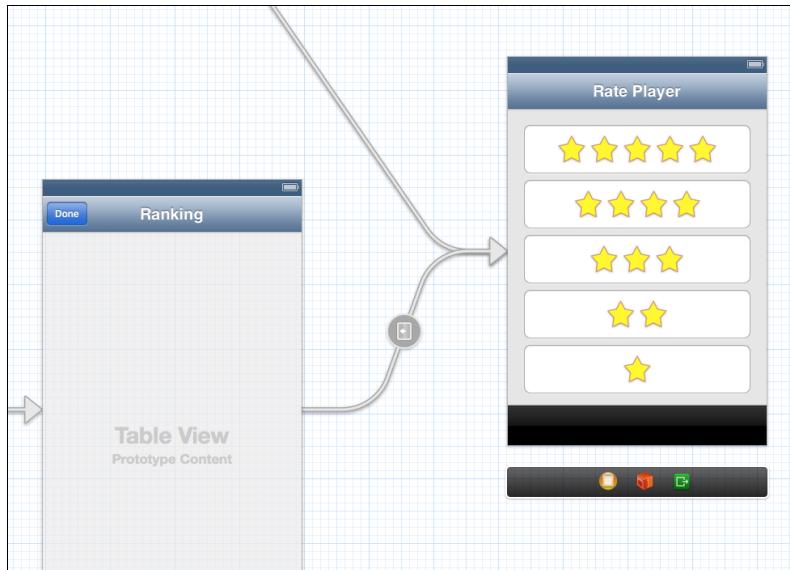
```

Now all the data model stuff is hooked up and you can run the app. Doing a swipe right will show all the players with 5 stars, double-tapping shows all the players with 1 star.

You're not done yet. You are also going to connect the Ranking screen to the Rate Player screen. Not for any good reason, but simply because you can.

In the storyboard, ctrl-drag from the Ranking view controller to the Rate Player screen and create a Push segue. Name it **RatePlayer**. There are now two segues

going to the Rate Player scene, both named “RatePlayer”, from different source view controllers:



To the `RatePlayerViewController`, it doesn't really matter how many incoming segues it has or which classes are at the other ends of those segues. It just expects to receive a `Player` object in its `player` property, and then uses a delegate to communicate back to the view controller that invoked it. In fact, it doesn't even know (or care) that it was invoked by a segue.

Note: If instead of using a delegate you had hardcoded the relationship between `RatePlayerViewController` and `PlayersViewController` then it would have been much harder to also segue to it from the `RankingViewController` or any other screen.

But `RatePlayerViewController` doesn't see `PlayersViewController` or `RankingViewController` at all. It only knows that there is some object that conforms to the `RatePlayerViewControllerDelegate` protocol that it can send messages to. This kind of design keeps your code modular, reusable, and free of unwanted side effects.

Because there is no prototype cell in the Ranking screen, you'll have to perform the segue manually.

In `RankingViewController.m`, change `didSelectRowAtIndexPath` to the following:

```
#pragma mark - Table view delegate

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
```

```

    Player *player = self.rankedPlayers[indexPath.row];
    [self performSegueWithIdentifier:@"RatePlayer"
                                sender:player];
}

```

First this finds the `Player` object in question and then sends it along as the `sender` parameter of `performSegueWithIdentifier`. That may be abusing the intent of the "sender" parameter a little (it's supposed to contain the object that initiated the segue), but it works just fine in practice.

Of course you also need a `prepareForSegue` method:

```

- (void)prepareForSegue:(UIStoryboardSegue *)segue
              sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"RatePlayer"])
    {
        RatePlayerViewController *ratePlayerViewController =
            segue.destinationViewController;
        ratePlayerViewController.delegate = self;
        ratePlayerViewController.player = sender;
    }
}

```

In **RankingViewController.h**, add an import and make the class conform to the delegate:

```

#import "RatePlayerViewController.h"

@interface RankingViewController : UITableViewController
<RatePlayerViewControllerDelegate>

```

Finally, add the delegate method to **RankingViewController.m**. It just closes the screen:

```

#pragma mark - RatePlayerViewControllerDelegate

- (void)ratePlayerViewController:
    (RatePlayerViewController *)controller
    didPickRatingForPlayer:(Player *)player
{
    [self.navigationController popViewControllerAnimated:YES];
}

```

Run the app. You should now be able to rate a player from the Ranking screen.

Note: You can also use `performSegueWithIdentifier` to trigger segues based on input from the accelerometer or gyroscope, or for any other events that cannot be expressed by Interface Builder. Your imagination is the limit (but don't take it too far or your users may start to question your sanity).

To let this app make at least some sense, it has to remove the player from the Ranking screen if the rating changes, because when that happens he or she by definition is no longer a best (5-star) or worst (1-star) player.

Add a new property to the **RankingViewController.h**:

```
@property (nonatomic, assign) int requiredRating;
```

For the list of best players this property's value will be set to 5, for the list of worst players it will be set to 1.

Change the implementation of the `RatePlayerViewControllerDelegate` method to the following:

```
- (void)ratePlayerViewController:
    (RatePlayerViewController *)controller
    didPickRatingForPlayer:(Player *)player
{
    if (player.rating != self.requiredRating)
    {
        NSUInteger index = [self.rankedPlayers
                             indexOfObject:player];
        [self.rankedPlayers removeObjectAtIndex:index];

        NSIndexPath *indexPath = [NSIndexPath
                                  indexPathForRow:index inSection:0];
        [self.tableView deleteRowsAtIndexPaths:@[indexPath]
                           withRowAnimation:UITableViewRowAnimationFade];
    }
    [self.navigationController popViewControllerAnimated:YES];
}
```

If the rating of the selected `Player` changed, this removes the `Player` object from the array and from the table.

GesturesViewController.m has to set `requiredRating` to the proper value before it transitions to the new screen:

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

```
if ([segue.identifier isEqualToString:@"BestPlayers"])
{
    // ...existing code...
    rankingViewController.requiredRating = 5;
}
else if ([segue.identifier isEqualToString:@"WorstPlayers"])
{
    // ...existing code...
    rankingViewController.requiredRating = 1;
}
```

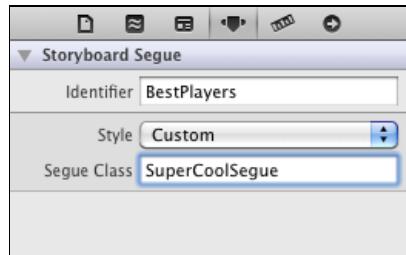
Try it out and see what happens.

Note: The app should really refresh the contents of the Players screen too, but that's left as an exercise for the reader.

Custom Segues

You've seen two types of segues already: Modal and Push. These will do fine for most apps but you have to admit they are a little boring. Fortunately, you can also create your own segue animations to liven things up a little.

Let's replace the transition from the Gestures screen to the Ranking screen with a transition of our own. Select the BestPlayers segue and set its Style to Custom. This lets you enter the name for the segue class that you're going to write in a moment. Set Segue Class to **SuperCoolSegue**. Do the same thing for the WorstPlayers segue.



To create your own segue, you have to extend the `UIStoryboardSegue` class. Add a new **Objective-C class** file to the project, named **SuperCoolSegue**, subclass of **UIStoryboardSegue**.

All you need to add to this class is a "perform" method. To start, add the simplest possible implementation inside **SuperCoolSegue.m**:

```
#import "SuperCoolSegue.h"
```

```
@implementation SuperCoolSegue

- (void)perform
{
    [self.sourceViewController
        presentViewController:self.destinationViewController
        animated:NO completion:nil];
}

@end
```

This immediately presents the destination view controller on top of the source controller (modally), without an animation of any sort.

Note: Previously you may have used the `presentModalViewControllerAnimated:` method for showing modal screens, but as of iOS 5 the new method `presentViewController:animated:completion:` is the preferred way to modally present view controllers.

Try it out. When you perform the gesture, the Ranking screen appears as before, but without the usual animation. That's not much fun, and a little abrupt, so let's give it a cool animated effect.

Replace the contents of **SuperCoolSegue.m** with:

```
#import <QuartzCore/QuartzCore.h>
#import "SuperCoolSegue.h"

@implementation SuperCoolSegue

- (void)perform
{
    UIViewController *source = self.sourceViewController;
    UIViewController *destination =
        self.destinationViewController;

    // Create a UIImage with the contents of the destination
    UIGraphicsBeginImageContext(destination.view.bounds.size);
    [destination.view.layer renderInContext:
        UIGraphicsGetCurrentContext()];
    UIImage *destinationImage = UIG
        raphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
}
```

```
// Add this image as a subview to the tab bar controller
UIImageView *destinationImageView = [[UIImageView alloc]
                                      initWithImage:destinationImage];
[source.parentViewController.view
         addSubview:destinationImageView];

// Scale the image down and rotate it 180 degrees
// (upside down)
CGAffineTransform scaleTransform =
    CGAffineTransformMakeScale(0.1, 0.1);
CGAffineTransform rotateTransform =
    CGAffineTransformMakeRotation(M_PI);
destinationImageView.transform =
    CGAffineTransformConcat(scaleTransform, rotateTransform);

// Move the image outside the visible area
CGPoint oldCenter = destinationImageView.center;
CGPoint newCenter = CGPointMake(oldCenter.x -
    destinationViewController.bounds.size.width, oldCenter.y);
destinationImageView.center = newCenter;

// Start the animation
[UIView animateWithDuration:0.5f delay:0
    options:UIViewAnimationOptionCurveEaseOut
    animations:^(void)
{
    destinationViewController.transform =
        CGAffineTransformIdentity;
    destinationViewController.center = oldCenter;
}]
completion: ^ (BOOL done)
{
    // Remove the image as we no longer need it
    [destinationImageView removeFromSuperview];

    // Properly present the new screen
    [source presentViewController:destination
        animated:NO completion:nil];
} ];
}

@end
```

The trick is to make a snapshot of the new view controller's view hierarchy before starting the animation, which gives you a `UIImage` with the contents of the screen, and then animate that `UIImage`. It's possible to do the animation directly on the

actual views but that may be slower and it doesn't always give the results you would expect. Built-in controllers such as the navigation controller don't lend themselves very well to these kinds of manipulations.

You add that `UIImage` as a temporary subview to the Tab Bar Controller, so that it will be drawn on top of everything else. The initial state of the image view is scaled, rotated, and outside of the visible screen, so that it will appear to tumble into view when the animation starts.

After the animation is done, you remove the image view again and properly present the view controller. This transition from the image to the actual view is seamless and unnoticeable to the user because they both contain the same contents.

Give it a whirl. If you don't think this animation is cool enough, then have a go at it yourself. See what kind of effects you can come up with... It can be a lot of fun to play with this stuff. If you also want to animate the source view controller to make it fly out of the screen, then I suggest you make a `UIImage` for that view as well.

When you close the Ranking screen, it still uses the regular sink-to-the-bottom animation. It's perfectly possible to perform a custom animation there as well, but remember that this transition is not a segue. The delegate is responsible for handling this, but the same principles apply. Set the `animated` flag to `no` and do your own animation instead.

Note: With the new unwind segues in iOS 6 you can also create your own `UIStoryboardSegue` subclass to perform the closing animation. You can read more about that in the book *iOS 6 by Tutorials*.

Storyboards and the iPad

You're going to make the Ratings app universal so that it also runs on the iPad.

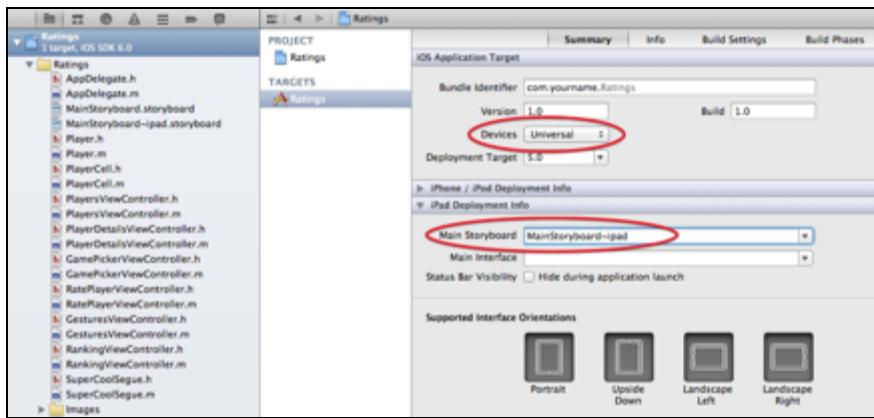
In the project's Target Summary screen, under **iOS Application Target**, change the **Devices** setting to **Universal**. That adds a new iPad Deployment Info section to the screen.

Add a new Storyboard file to the project. Create a new file, and from the **User Interface** section choose the **Storyboard template**, and set the Device Family to iPad. Save it in the `en.lproj` folder as `MainStoryboard~ipad.storyboard`.

Open this new storyboard in the editor and drag a View Controller into it. Notice that this is now an iPad-sized view controller. Drag a Label into this new view controller and give it some text, just for testing.

Note: Disable Auto Layout in the storyboard's File inspector to make sure this app can still run on iOS 5.

Back in your target settings, in the iPad Deployment Info section on the Target Summary screen, choose **MainStoryboard~ipad** as the Main Storyboard:



Also make sure to enable all interface orientations, because iPad apps are expected to support both portrait and landscape.

Then change **AppDelegate.m** to the following:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    _players = [NSMutableArray arrayWithCapacity:20];

    // ...existing code...

    if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad)
    {
        UITabBarController *tabBarController =
            (UITabBarController *)self.window.rootViewController;
        UINavigationController *navigationController =
            [tabBarController viewControllers][0];

        PlayersViewController *playersViewController =
            [navigationController viewControllers][0];
        playersViewController.players = _players;

        GesturesViewController *gesturesViewController =
            [tabBarController viewControllers][1];
        gesturesViewController.players = _players;
    }
}
```

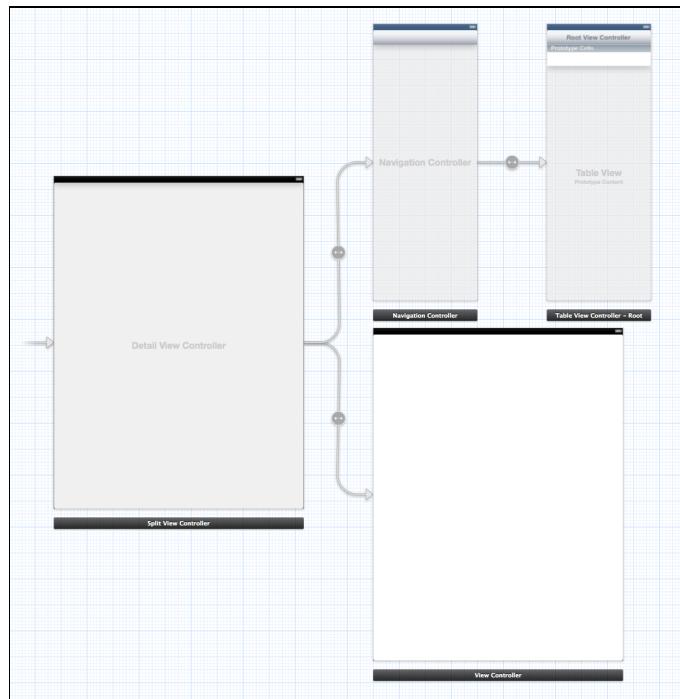
```
    return YES;  
}
```

The new if-statement checks whether the app runs on a regular iPhone or iPod touch, and then proceeds to do the things it did before, i.e. passing the `_players` array to the two view controllers inside the Tab Bar Controller. The if-statement is necessary because you don't want to do any of that stuff on the iPad version, because the iPad storyboard doesn't have a Players or Gestures view controller (yet).

Now run the app on the iPad Simulator. Instead of the tab bar interface from before you should see the new view controller with the test label from the new storyboard. The iPad version of the app successfully loaded its own storyboard.

There really aren't that many differences between making storyboards for the iPhone and the iPad, except that the iPad storyboards will be a lot bigger. You also have two additional segue types: Popover and Replace.

Let's get serious with this iPad stuff. Get rid of the view controller from the iPad storyboard and drag a new Split View Controller into the canvas. The Split View Controller comes with three other scenes attached... I told you, you need a big monitor!



By default this Split View Controller is oriented in portrait, but if you set the Orientation field from its Simulated Metrics to Landscape, then you can see both the master and detail panes.

Note that the arrows between these scenes are all relationship connections. Just like the Navigation and Tab Bar Controllers, a Split View Controller is a container of other view controllers. On the iPhone only one scene from the storyboard is visible at a time when you run the app, but on the iPad several scenes may be visible simultaneously. The master and detail panes of the Split View Controller are an example of that.

If you run the app now it doesn't work very well yet. All you get is a white screen that doesn't rotate when you flip the device (or the simulator) over.

Add a new file to the project and name it **DetailViewController**, subclass of **UIViewController**. This is the class for the big scene that goes into the right pane of the Split View Controller.

Change **DetailViewController.h** to:

```
@interface DetailViewController : UIViewController
<UISplitViewControllerDelegate>

@property (nonatomic, weak) IBOutlet UIToolbar *toolbar;

@end
```

You're making this object the delegate for the Split View Controller so it will be notified whenever the device is rotating.

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

```
#import "DetailViewController.h"

@implementation DetailViewController
{
    UIPopoverController *_masterPopoverController;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    return YES;
}

#pragma mark - UISplitViewControllerDelegate

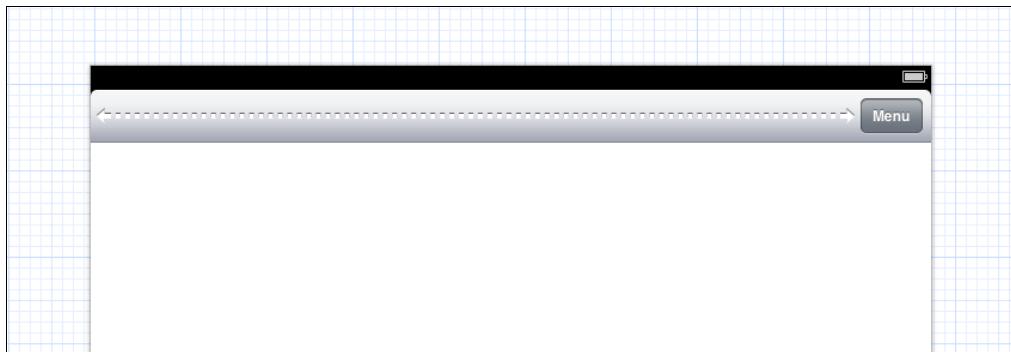
- (void)splitViewController:
    (UISplitViewController *)splitViewController
willHideViewController:(UIViewController *)viewController
withBarButtonItem:(UIBarButtonItem *)barButtonItem
forPopoverController:(UIPopoverController *)popoverController
```

```
{  
    barButtonItem.title = @"Master";  
    NSMutableArray *items = [[self.toolbar items] mutableCopy];  
    [items insertObject:barButtonItem atIndex:0];  
    [self.toolbar setItems:items animated:YES];  
    _masterPopoverController = popoverController;  
}  
  
- (void)splitViewController:  
    (UISplitViewController *)splitController  
willShowViewController:(UIViewController *)viewController  
invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem  
{  
    NSMutableArray *items = [[self.toolbar items] mutableCopy];  
    [items removeObject:barButtonItem];  
    [self.toolbar setItems:items animated:YES];  
    _masterPopoverController = nil;  
}  
  
@end
```

This is the minimal amount of stuff you need to do to support a Split View Controller in your app.

In the storyboard, set the Class of the big scene to **DetailViewController**. Drag a Toolbar into this scene, at the top. The toolbar comes with a default Bar Button Item named simply Item. Rename it to “Menu” (this is for a popover you’ll be adding later) and add a flexible space in front of it.

The design should look like this:



Connect the Toolbar to the view controller’s `toolbar` outlet. Also make sure the autosizing for the toolbar is set up as follows:



By default, toolbars are made to stick to the bottom of the screen but you want this one to sit at the top at all times, or it won't look right when the device is rotated.

You're not done yet. You also have to make a class for the "master" view controller (the Table View Controller inside the Navigation Controller), i.e. what goes on the left pane of the split-view. The only reason for doing this is so you can override the `shouldAutorotateToInterfaceOrientation` method and make it return `YES`. On the iPad all visible view controllers need to agree on the rotation or the app won't rotate properly.

Create a new `UITableViewController` subclass and name it **MasterViewController**. Add the method to **MasterViewController.m**:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    return YES;
}
```

Open **MainStoryboard~ipad.storyboard** again, select the scene named Root View Controller in the storyboard and set its Class to **MasterViewController**. Xcode will give some warnings about the missing data source methods from this new class, but you don't have to worry about that.

There is one more thing to do. The `DetailViewController` class is the delegate for the split view controller, but you haven't set up that delegate relationship anywhere yet. As you know you cannot make these kinds of connections directly in Interface Builder (unfortunately!) so you'll have to write some code in the App Delegate.

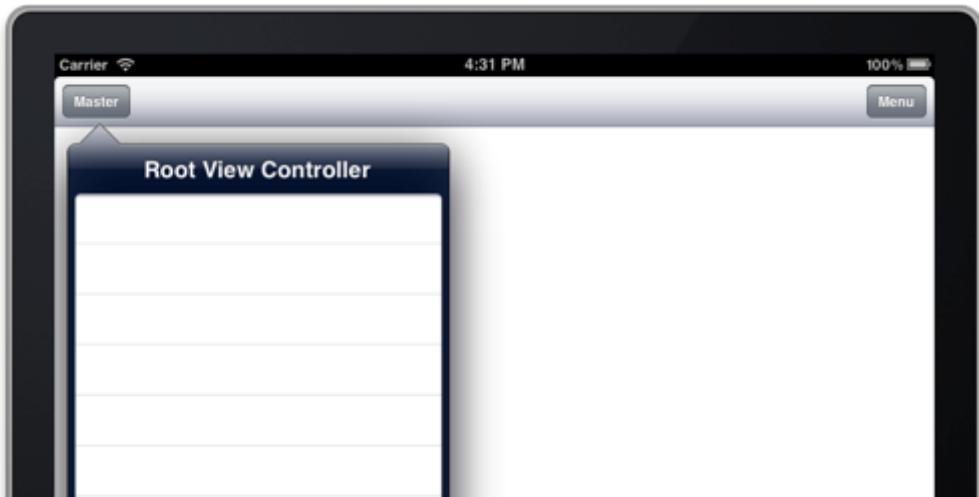
Switch to **AppDelegate.m**, and change the `didFinishLaunchingWithOptions` method to:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // ...existing code...

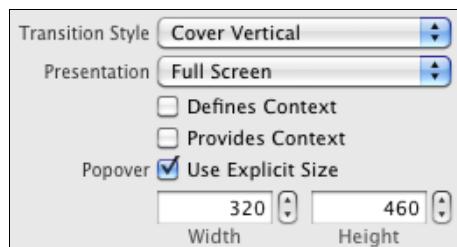
    if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad)
    {
        // ...existing code...
    }
    else
    {
```

```
UISplitViewController *splitViewController =  
    (UISplitViewController *)self.window.rootViewController;  
  
splitViewController.delegate =  
    [splitViewController.viewControllers lastObject];  
}  
  
return YES;  
}
```

Run the app and you should now have a fully functional split view controller!



You can configure the popover size for the master view controller in the Storyboard Editor. The Attributes inspector for the Split View Controller has a setting for the popover size:



As of iOS 5.1, the master pane no longer appears in a popover but is a full-height panel that slides in from the left:



Xcode comes with a Master-Detail Application template that already sets all of this up for you, but it's good to know how to do it from scratch as well.

Note: The way autorotation works has changed a bit on iOS 6. The trick with making the `MasterViewController` class to override the `shouldAutorotateToInterfaceOrientation` method is no longer necessary if your app is meant for iOS 6 and up.

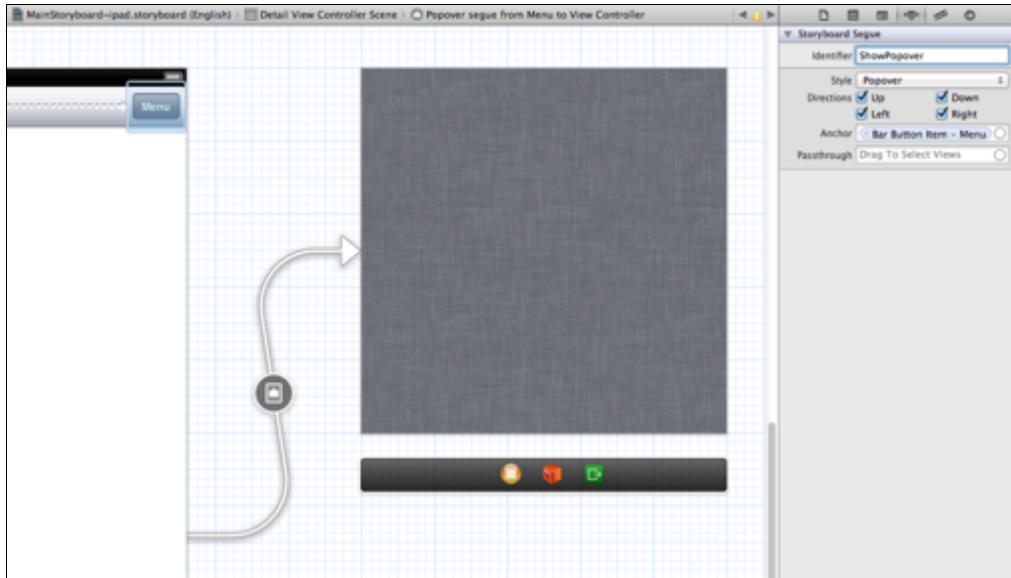
Popovers

You can also easily create your own popovers. You add a new scene to the storyboard and then simply link a "Popover" segue to it.

Drag a new View Controller into the canvas. This will become the content controller of the popover. It's a little too big so under Simulated Metrics change its Size from Inferred to Freeform. Also remove the simulated status bar.

Now you can resize its view in the Size inspector. Make it 400 by 400 points. Just so you can see that the popover actually works, change the Background Color of the view to something other than white (for example Scroll View Textured Background).

Ctrl-drag from the Menu bar button item on the Detail View Controller to this new view controller and choose the Popover segue. Name it **ShowPopover**. Notice that the Attributes inspector for a popover segue has quite a few options that correspond to the properties that you can set on `UIPopoverController`:



Run the app and you have a working popover! Talk about easy...

The segue that presents a popover is the `UIStoryboardSegue`, a subclass of `UIStoryboardSegue`. It adds a new property to the segue object, `popoverController`, that refers to the `UIPopoverController` that manages the popover. You should really capture that `popoverController` object in an instance variable so that you can dismiss it later if necessary.

Add the `UIPopoverControllerDelegate` protocol to the `@interface` declaration in **DetailViewController.h**:

```
@interface DetailViewController : UIViewController
<UISplitViewControllerDelegate, UIPopoverControllerDelegate>
```

In **DetailViewController.m**, add a new instance variable:

```
@implementation DetailViewController
{
    UIPopoverController *_masterPopoverController;
    UIPopoverController *_menuPopoverController;
}
```

Add the by now very familiar `prepareForSegue` method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                    sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"ShowPopover"])
    {
        _menuPopoverController =
            ((UIStoryboardSegue *)segue).popoverController;
```

```
        _menuPopoverController.delegate = self;
    }
}
```

Here you put the value from the segue's popoverController property into your own `_menuPopoverController` variable and make the `DetailViewController` the delegate for the popover controller.

Add the delegate method:

```
#pragma mark - UIPopoverControllerDelegate

- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)popoverController
{
    _menuPopoverController.delegate = nil;
    _menuPopoverController = nil;
}
```

It simply sets the instance variable back to `nil` when the popover is dismissed.

Now that you have an instance variable that refers to the segue's popover controller when it is visible, you can dismiss the popover when the device rotates with the following method:

```
- (void)willAnimateRotationToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
duration:(NSTimeInterval)duration
{
    if (_menuPopoverController != nil &&
        _menuPopoverController.popoverVisible)
    {
        [_menuPopoverController dismissPopoverAnimated:YES];
        _menuPopoverController = nil;
    }
}
```

Try it out.

You may have noticed a small problem: every time you tap the Menu button, a new popover is opened but the previous one isn't closed first. Tap repeatedly on the button and you end up with a whole stack of popovers. This is annoying (and might even cause your app to be rejected if you ship with this!) but fortunately the workaround is easy.

It is not possible to cancel a segue once it has started so that is not an option, but when a popover is open you do have a reference to it in the `_menuPopoverController` variable. Change `prepareForSegue` to:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
                  sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"ShowPopover"])
    {
        if (_menuPopoverController != nil &&
            _menuPopoverController.popoverVisible)
        {
            [_menuPopoverController dismissPopoverAnimated:NO];
        }

        _menuPopoverController =
            ((UIStoryboardPopoverSegue *)segue).popoverController;

        _menuPopoverController.delegate = self;
    }
}
```

No matter how many times you tap the Menu button now, only one popover is ever visible.

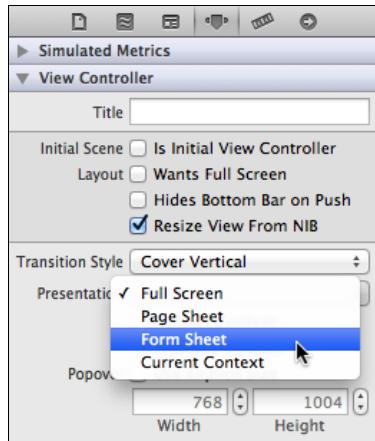
Note: On iOS 6 it is a little easier to prevent multiple instances of the popover from showing using the new method `shouldPerformSegueWithIdentifier:sender:` that is called just before the segue happens. For example:

```
- (BOOL)shouldPerformSegueWithIdentifier:(NSString *)identifier
                                  sender:(id)sender
{
    if ([identifier isEqualToString:@"ShowPopover"] &&
        _menuPopoverController != nil &&
        _menuPopoverController.popoverVisible)
        return NO;
    else
        return YES;
}
```

Besides the Popover segue, iPad storyboards can also have a “Replace” segue. You use this to replace the master or detail view controllers in a Split View Controller. Like the built-in Settings app, you could have a table view in the master pane with each row having its own detail view. You can put a Replace segue between each

row and its associated detail scene, to swap out the detail controller when you tap such a row.

You can also use segues for presenting modal form sheets and page sheets. To do this you simply set the Presentation attribute for the destination view controller to the style you want to use:

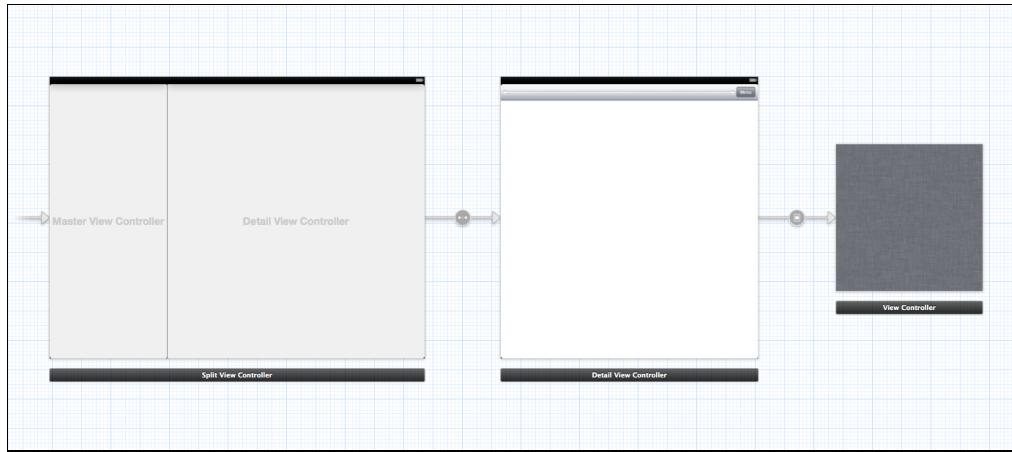


Manually Loading Storyboards

The iPad version of the Ratings app doesn't have a lot to do with ratings yet, so let's put the screens from the iPhone storyboard into the master pane of the Split View Controller. You're going to do that programmatically as there is no way you can link from one storyboard to another.

Note: I'm not saying that this is what you should do in your own universal apps. It may make more sense to keep the storyboards for the iPhone and iPad apps completely separate. However, for the purposes of this tutorial it is a good demonstration of how to load additional storyboards by hand, just in case you ever need to write an app that uses more than one storyboard.

Delete the Navigation Controller and Root View Controller scenes from the iPad storyboard. You can also delete the `MasterViewController` class files from the project if you want. The storyboard now looks like this:



You're going to load the iPhone storyboard in the App Delegate and then put its Tab Bar Controller into the master pane of the Split View Controller.

Storyboards are represented by the `UIStoryboard` class. The main storyboard file is loaded automatically when your app starts, but you can load additional storyboards by calling `[UIStoryboard storyboardWithName:bundle:]`.

There is only one problem, if you attempt to load the iPhone storyboard using,

```
UIStoryboard *storyboard = [UIStoryboard
    storyboardWithName:@"MainStoryboard" bundle:nil];
```

then this will actually load the **MainStoryboard~ipad** file. Even though you only specified "MainStoryboard" as the filename, the app will automatically append "~ipad" to it because this is a universal app running in iPad mode. The app will get totally confused. For this section of the chapter you'd better rename the iPad storyboard file. Call it **iPadMainStoryboard.storyboard** instead.

Don't forget to change the Main Storyboard setting in the iPad Deployment Info section under Target Summary. (It's also a good idea to do a clean build and to remove the app from the Simulator before you continue just to make sure it doesn't use a cached version of the old storyboard.)

In **AppDelegate.m**, change `didFinishLaunchingWithOptions` to:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // ...code to make the Player objects...

    UITabBarController *tabBarController;

    if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad)
    {
```

```
tabBarController = (UITabBarController *)  
    self.window.rootViewController;  
}  
else  
{  
    UISplitViewController *splitViewController =  
        (UISplitViewController *)  
            self.window.rootViewController;  
    id detailViewController =  
        [splitViewController.viewControllers lastObject];  
  
    UIStoryboard *storyboard = [UIStoryboard  
        storyboardWithName:@"MainStoryboard" bundle:nil];  
    tabBarController = [storyboard  
        instantiateInitialViewController];  
  
    NSArray *viewControllers = @[tabBarController,  
                                detailViewController];  
    splitViewController.viewControllers = viewControllers;  
    splitViewController.delegate = detailViewController;  
}  
  
UINavigationController *navigationController =  
    [tabBarController viewControllers][0];  
PlayersViewController *playersViewController =  
    [navigationController viewControllers][0];  
playersViewController.players = _players;  
  
GesturesViewController *gesturesViewController =  
    [tabBarController viewControllers][1];  
gesturesViewController.players = _players;  
  
return YES;  
}
```

The new part is this:

```
UIStoryboard *storyboard = [UIStoryboard  
    storyboardWithName:@"MainStoryboard" bundle:nil];  
  
tabBarController = [storyboard  
    instantiateInitialViewController];
```

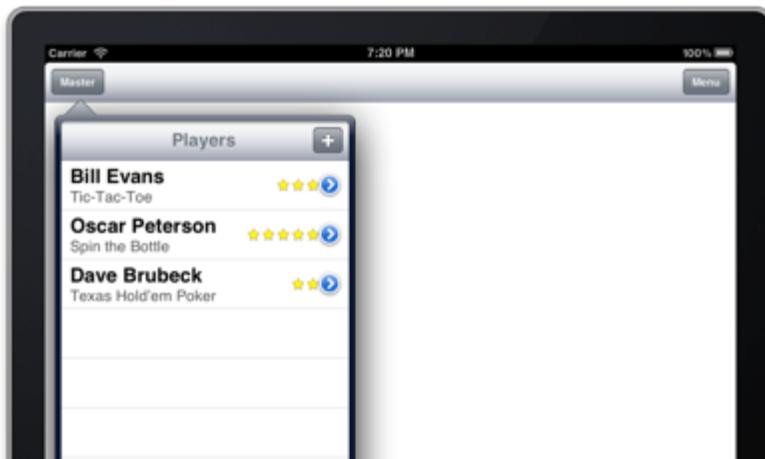
This loads the **MainStoryboard** file into a new `UIStoryboard` object and then calls `instantiateInitialViewController` to load its initial view controller, which in your case is the Tab Bar Controller.

Once you have the Tab Bar Controller, you need to put it into the Split View Controller's master pane. Currently the split-view only contains the Detail View Controller, so you add the Tab Bar Controller to its `viewControllers` property:

```
NSArray *viewControllers = @*[tabBarController,
                               detailViewController];

splitViewController.viewControllers = viewControllers;
```

Run the app and you should see the screens from the iPhone version of the app in the split-view popover:



If you start tapping on stuff you'll notice the integration isn't as seamless as it could be. That's because you never configured the scenes from your original storyboard to work on the iPad.

Go through the `.m` files for all the view controllers and add the `shouldAutorotateToInterfaceOrientation` method:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
        return YES;

    return (interfaceOrientation !=
        UIInterfaceOrientationPortraitUpsideDown);
}
```

Now the app will properly rotate to landscape mode.

Note: Overriding `shouldAutorotateToInterfaceOrientation`: in all your view controllers is no longer necessary on iOS 6. But if you still want your app to be compatible with iOS 5, then doing so is unavoidable.

Interface Builder doesn't let you set the popover size for view controllers in an iPhone storyboard (which is understandable because the iPhone doesn't have popovers), so you'll have to do that in code. Add to `application:didFinishLaunchingWithOptions:`:

```
tabBarController.contentSizeForViewInPopover = CGSizeMake(320, 460);
```

Now the popover from the Split View Controller is just as big as the contents of the iPhone screen. This setting only has an effect on iOS 5.0. As of 5.1, the split-view master pane is no longer a popover but a panel that always takes up the full height of the screen.

There is one more minor problem with the app and that is that the modal view controllers are being presented full screen, which looks a little weird when you tap the + button to add a new player. This is related to the modal presentation style setting of the view controllers.

For example, if in **PlayersViewController.m** in `prepareForSegue`, you do,

```
navigationController.modalPresentationStyle =
    UIModalPresentationCurrentContext;
playerDetailsViewController.contentSizeForViewInPopover =
    CGSizeMake(320, 423);
```

then the modal scene doesn't take over the whole screen – the modal view will appear in the master controller only.

`UIStoryboard's instantiateInitialViewController` method is not the only way to load a view controller from a storyboard. You can also ask for a specific view controller using `instantiateViewControllerWithIdentifier:`. That is useful if you don't have a segue to that view controller in the storyboard.

To demonstrate this feature, open the iPhone storyboard and delete the `EditPlayer` segue from the Players scene. This segue was formerly triggered by the disclosure button.

Replace the `accessoryButtonTappedForRowWithIndexPath:` method in **PlayersViewController.m** with the following:

```
- (void)tableView:(UITableView *)tableView
    accessoryButtonTappedForRowWithIndexPath:
    (NSIndexPath *)indexPath
```

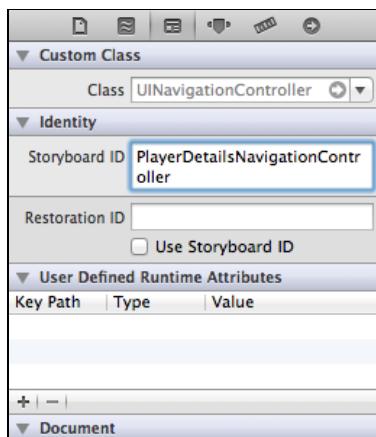
```
{  
    UINavigationController *navigationController =  
        [self.storyboard instantiateViewControllerWithIdentifier:  
            @"PlayerDetailsNavController"];  
  
    PlayerDetailsViewController *playerDetailsViewController =  
        [navigationController viewControllers][0];  
    playerDetailsViewController.delegate = self;  
  
    Player *player = self.players[indexPath.row];  
    playerDetailsViewController.playerToEdit = player;  
  
    [self presentViewController:navigationController  
        animated:YES completion:nil];  
}
```

This is very similar to what you did in `prepareForSegue`, except for the first line:

```
UINavigationController *navigationController =  
[self.storyboard instantiateViewControllerWithIdentifier:  
    @"PlayerDetailsNavController"];
```

The `self.storyboard` property refers to the `UIStoryboard` object that this view controller was loaded from. You can ask this `UIStoryboard` object to instantiate a specific view controller, in this case the Navigation Controller that contains the Player Details scene.

In order for this to work, you have to set an identifier (**PlayerDetailsNavController**) on the view controller in its Identity inspector:



Note: You want to instantiate the Navigation Controller, so set this identifier on the Navigation Controller, not on the Player Details view controller.

Run the app (from the iPhone simulator) and try editing a player. You'll see that this still works even though the storyboard no longer has the EditPlayer segue.

Last Words

Localizing your storyboards is pretty easy; it works just like localizing any other resource. Simply select the storyboard and under Localization in the File inspector you add a new language. Done.

You've seen that each scene in the storyboard has a dock area that contains the top-level objects for that scene, usually just the First Responder, the View Controller, and an Exit icon that is used with unwind segues on iOS 6. Gesture recognizers and table view sections are also added to the dock.

In theory you can drag any other view objects that you want into the dock, just like you can with nibs. The problem is that the Interface Builder doesn't give you any means to edit these objects.

For example, if you want to change the background image for a static cell, you can drag a `UIImageView` into the dock and hook it up to the cell's `backgroundImage` property. That works but you can only edit that Image View through the inspectors, it is not visible in the canvas anywhere. I hope the storyboard team adds in this functionality because it's really handy to load extra objects that way.

Another thing that storyboards don't support very well is custom container view controllers. Having a `UITabBarController` or `UINavigationController` is fine and dandy, but what if you want to make your own "tab bar" or some other container using the new view containment APIs?

As of iOS 6 you can use the Container View element to embed the contents of one view controller inside another in your storyboards (see the book *iOS 6 by Tutorials* for more info), but that's only half a solution and it obviously doesn't work on iOS 5. For the time being, you will have to handle view controller containment programmatically and not in your storyboards.

Tip: You can put the standard view controllers such as the `UIImagePickerController` and `MFMailComposeViewController` in your storyboards with a little trick. Drag a regular view controller onto the canvas and set its class to **UIImagePickerController**. In `prepareForSegue`, set the properties on the new controller and off you go. I'm not really sure if this is supported behavior, but it seems to work fairly well. ;-)

Where To Go From Here?

Congratulations on making it through – you now have a lot of hands-on experience with using storyboards to create an app! I think storyboards are a great addition to iOS and I look forward to using them in my new projects. Especially prototype cells and static cells are bound to be big time savers!

Should you change your existing projects to use Storyboards? Probably not, but for any new apps that you want to make iOS 5+ only it's a good idea.

In case you want to migrate your existing projects anyway, it is possible to copy the contents of your nibs into the storyboard. First drag a new View Controller into the storyboard and delete its main view. Then open your nib file in a new window and drag its view into the new View Controller on the storyboard. This will take over the existing design and save you some time recreating it. You may need to reconnect the outlets and actions, though.

Have fun storyboarding!

6

Chapter 6: Beginning iCloud

By Cesare Rocchi

iCloud is meant to solve a big problem which affects our daily lives as users of multiple devices: data synchronization. We all have stuff we use on our devices regularly like documents, pictures, videos, emails, calendars, music, and address books. Now that mobile devices are becoming more and more common, we often find our stuff scattered in many places. Some might be on our PC, some on our iPhone, and some on our iPad. How many times have you tried to quickly open a document and realized "argh, I have it saved on another device"?

A common workaround is to keep data you need often on some shared folder (like Dropbox) or send yourself emails with recent versions of documents. There are ways to address these issues, but they are, as I said, workarounds that require some effort, like remembering to send yourself an email or to upload recently changed documents to the shared folder.

iCloud is a service to which you can delegate this "remembering". It is a set of central servers that store your documents, and make the latest version available to every device/app compatible with iCloud (iPhone, iPod, iPad, Mac, or even PC).

Here is a common scenario: you enter an appointment in the calendar on the Mac, you forget to synch it with the iPhone, pick up your iPhone and get out of the office. When will you see the notification? When you get back to the office, it might be too late! Before iCloud you'd be in trouble, but now the calendar app on the iPhone is integrated with iCloud! iCloud will automatically pick changes to your calendar and push them to all the other devices connected to your account. It is all in one place (the cloud) and anywhere (your devices) at the same time!

To some extent this is a solution very similar to the IMAP protocol, which allows keeping emails in synch on different devices, due to the flexibility of the protocol and the fact that messages are stored on a central server and copied on clients upon request.

Before the release of iOS5, iTunes has been a sort of solution to the "synchronization problem", with the huge drawback mentioned above: you have to remember to physically connect your device to your Mac or PC and hit the synch button. Moreover, you have to repeat the operation for each of your devices. Now

iCloud solves everything automatically without the need to remember, because some magic process takes care of synchronizing documents, pictures, preferences, contacts and calendars.

iCloud is great news for developers. By using a set of new APIs, you can configure your app to store and retrieve data on iCloud as well. Can you think of the advantages? The possibilities are just endless!

In this tutorial, you'll investigate iCloud by implementing a set of simple applications that interact with cloud servers to read, write and edit documents. In the process, you'll learn about the new `UIDocument` class, querying iCloud for files, autosaving, and much more!

To get the most out of this tutorial, you will need two physical devices for testing, such as an iPhone and an iPad. The simulator does not currently have iCloud support.

Note: There's an important note about Apple's data saving policy and iCloud that you should be aware of. Whether or not you enable iCloud in your application, all the contents stored in local "Documents" directory of your application are automatically backed up on iCloud if the user chooses to enable backup for applications.

If your data can be recreated somehow, don't store it in "Documents", otherwise backups are not efficient and your app wastes user's iCloud storage space. In fact, Apple is getting more strict about checking for this kind of thing, and your app might be rejected if you do this.

Temporary data can be stored in `<Application_Home>/tmp`. Data that can be downloaded again, like copies of magazines, can be stored in `<Application_Home>/Library/Caches`.

Under the hood

Before you begin, let's talk about how iCloud works.

In iOS each application has its data stored in a local directory, and can only access data in its own directory. This prevents apps from reading or modifying data from other apps.

iCloud allows you to upload your local data to central servers on the network, and receive updates from other devices. The replication of content across different devices is achieved by means of a continuous background process (referred to as *daemon*), which detects changes to a resource (document) and uploads them to the central storage.

If you ever tried to create something like this on your own, you know there are several major challenges when implementing this:

1. **Conflict resolution.** What happens if you modify a document on your iPhone, and modify the same document on your iPad at the same time? Somehow you have to reconcile these changes. iCloud allows you to break your documents into chunks to prevent many merge conflicts from being a problem (because if you change chunk A on device 1, and chunk B on device 2, since chunk A and B are different you can just combine them). For cases when it truly is a problem, it allows you as a developer fine-grained control over how to handle the problem (and you can always ask the user what they would like to do).
2. **Background management.** iOS apps only have limited access to running tasks in the background, but keeping your documents up-to-date is something you want to always be doing. The good news is that, since iCloud synchronization is running in a background daemon, it's always active!
3. **Network bandwidth costs.** Continuously pushing documents between devices can take a lot of network bandwidth. As mentioned above, iCloud helps reduce the costs by breaking each document into chunks. When you first create a document, every chunk is copied to the cloud. When subsequent changes are detected only the chunks affected are uploaded to the cloud, to minimize the usage of bandwidth and processing. A further optimization is based on a peer-to-peer solution. That happens when two devices are connected to the same iCloud account and the same wireless network. In this case data take a shortcut and move directly between devices.

The mechanisms described so far are enabled by a smart management of metadata like file name, size, modification date, version etc. This metadata is aggressively pushed to the servers, and iCloud uses this information to determine what needs to be pulled down to each device.

Note that devices pull data from the cloud when "appropriate". The meaning of this depends on the OS and platform. For example an iPhone has much less power and "battery dependency" than an iMac plugged into a wall. In this case iOS might decide to notify just the presence of a new file, without downloading it, whereas Mac OS X might start the download immediately after the notification.

The important aspect is that an application is always aware of the existence of a new file, or changes to an already existing file, and through an API the developer is free to implement the synchronization policy. In essence the API allows an app to know the "situation" on iCloud even if the files are not yet local, leaving the developer free to choose whether (and when) to download an updated version.

Note. While you're going through this tutorial, if you get stuck with a bug or unexpected behavior it is suggested to start from scratch with a fresh app install and make sure no previous data is on your device. This is because previous versions of the app (especially if you use the same provisioning and bundle id) might conflict with the version you are working on.

To do this, uninstall the application from your device and delete the data stored by your app from Settings/iCloud/Storage & Backup/Manage Storage. Most of the examples create an "Unknown" item in this list.

Configuring iCloud

When you first set up an iOS device, you'll be asked to configure an iCloud account by providing or creating an Apple ID. Configuration steps will also allow you to set which services you want to synchronize (calendar, contacts, etc.). Those configurations are also available under Settings\iCloud on your device.

Before you proceed any further with this tutorial, make sure that you have two test devices, and that iCloud is working properly on both devices. One easy way to test this is to add a test entry into your calendar, and verify that it synchronizes properly between your various devices. You can also visit <http://www.icloud.com> to see what's in your calendar.

Once you're sure iCloud is working on your device, you'll try it out in an app of your own creation!

Enabling iCloud in your application

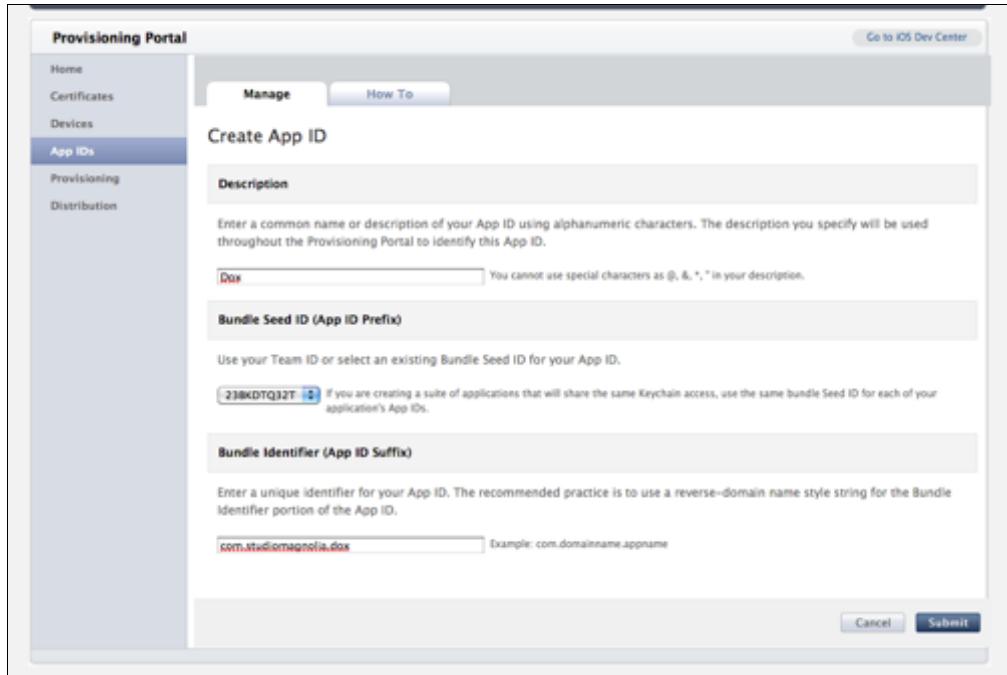
In this tutorial, you'll be creating a simple app that manages a shared iCloud document called "dox". The app will be universal and will be able to run on both iPhone and iPad, so you can see changes made on one device propagated to the other.

There are three steps to use iCloud in an app, so let's try them out as you start this new project.

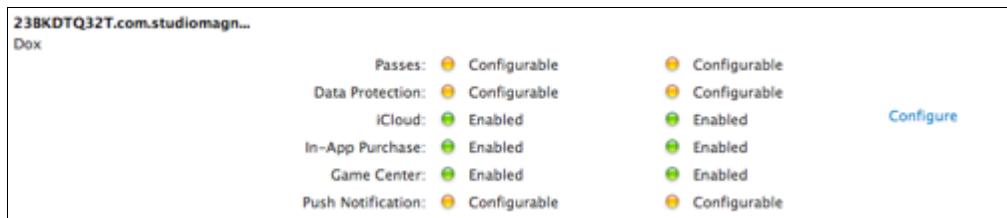
1. Create an iCloud-enabled App ID

To do this, visit the iOS Developer Center and log onto the iOS Provisioning Portal. Create a new App ID for your app similar to the below screenshot (but replace the bundle identifier with a unique name of your own).

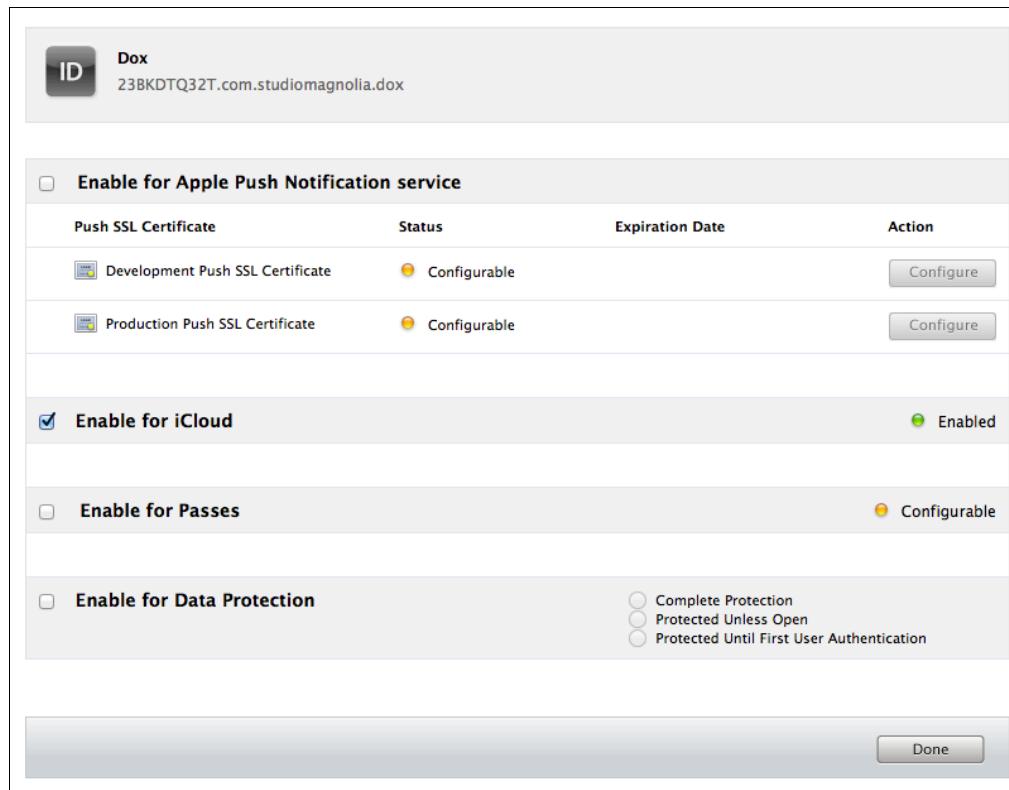
Note. Be sure to end your App ID with "dox" in this tutorial, because that is what you will be naming the project. For example, you could enter com.yourname.dox.



After you create the App ID, you will see that Push Notifications and Game Center are automatically enabled, but iCloud requires you to manually enable it. Click the Configure button to continue.

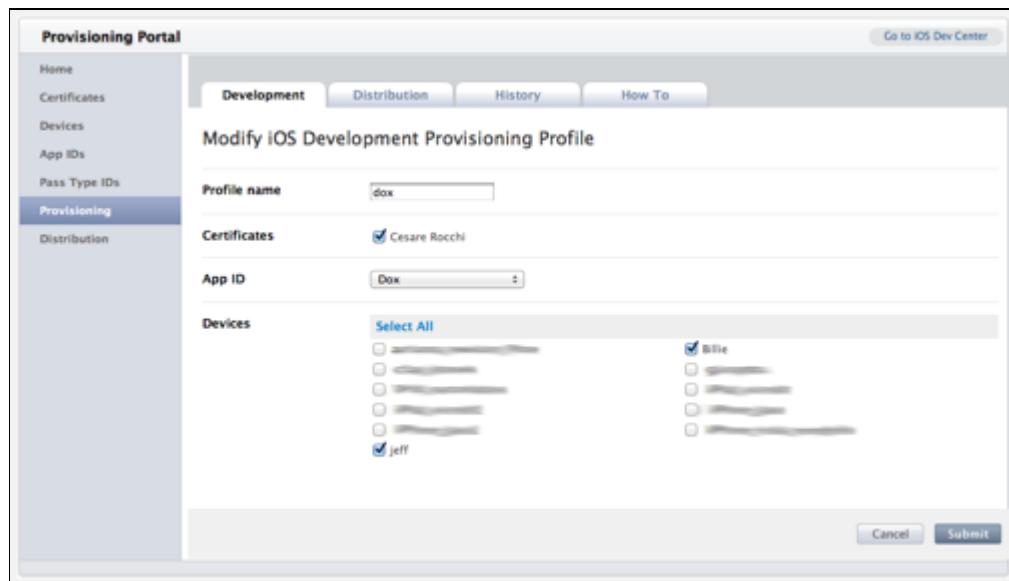


On the next screen, click the checkbox next to **Enable for iCloud** and click OK when the popup appears. If all works well, you will see a green icon next to the word Enabled. Then just click **Done** to finish.

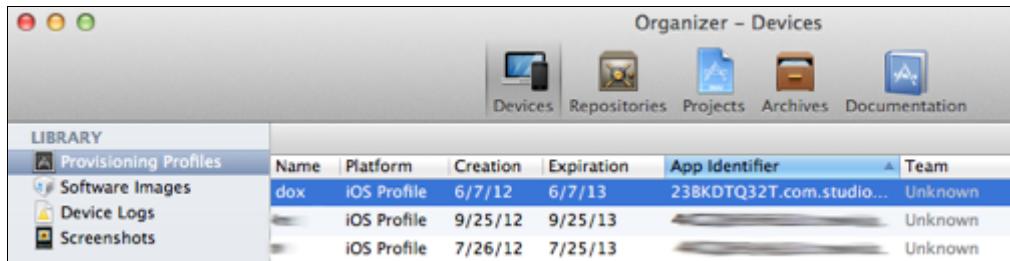


2. Create a provisioning profile for the App ID

Still in the iOS Provisioning Portal, switch to the Provisioning section, and click New Profile. Select the App ID you just created from the dropdown, and fill out the rest of the information, similar to the screenshot below.

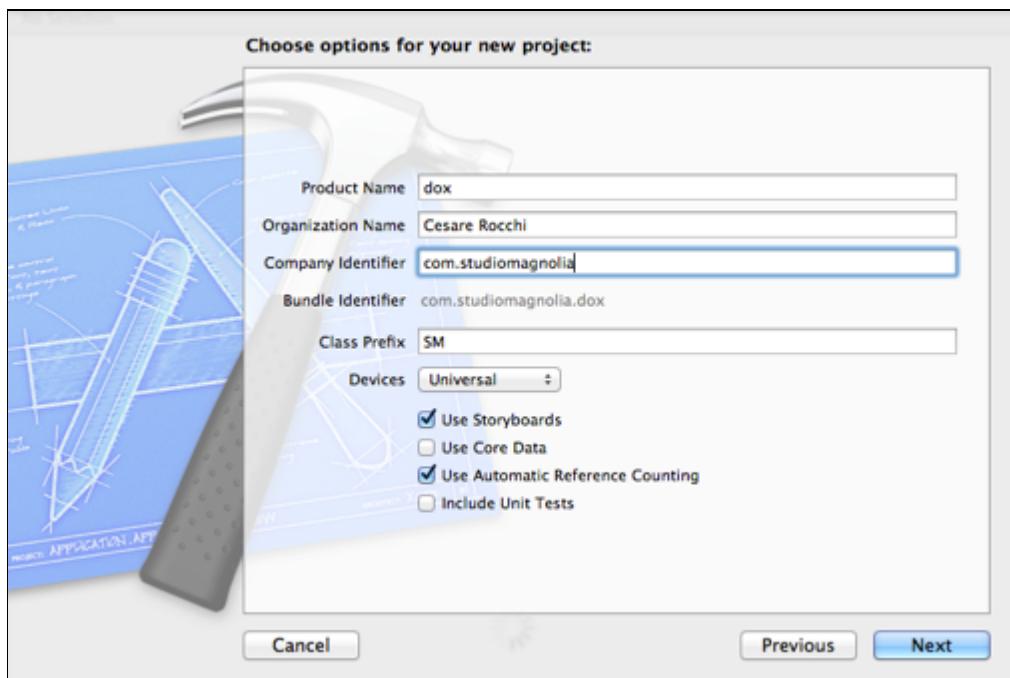


After creating the profile, refresh the page until it is ready for download, and then download it to your machine. Once it's downloaded, double click it to bring it into Xcode, and verify that it is visible in Xcode's Organizer.



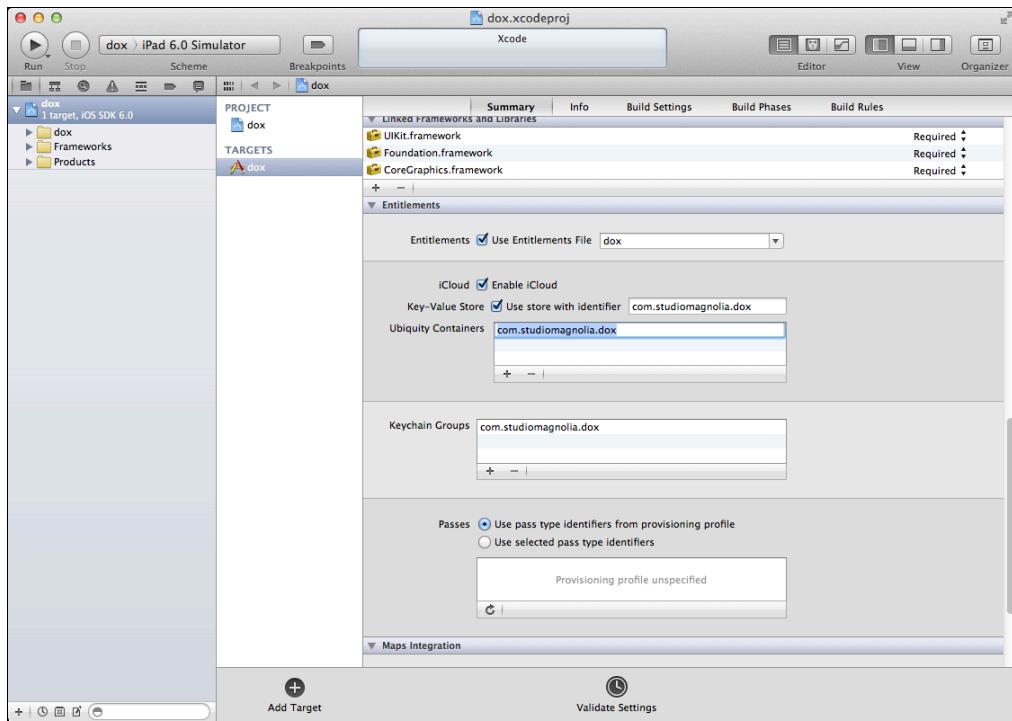
3. Configure your Xcode project for iCloud

Start up Xcode and create a new project with the **iOS\Application\Master-Detail Application** template. Enter **dox** for the product name, enter the company identifier you used when creating your App ID, enter **SM** for the class prefix, set the device family to **Universal**, and make sure **Use Automatic Reference Counting** and **Use Storyboards** are checked (but leave the other checkboxes unchecked).



Once you've finished creating the project, select your project in the Project Navigator and select the dox target. Select the Summary tab, and scroll way down to the Entitlements section.

Once you're there, check the **Entitlements** and **Enable iCloud** checkboxes, and populate the fields based on your App ID, as shown below.



This is what the fields here mean:

- The **Entitlements File** points to a property list file which, much like the `info.plist` file, includes specifications about application entitlements.
- The iCloud **Key-Value Store** represents the unique identifier which points to the key-value store in iCloud. You will learn more about the key-value store more in the next chapter.
- The **Ubiquity Containers** section represents "directories" in the cloud in which your applications can read/write documents. Yes, you have read correctly, I said applications (plural), for a user's container can be managed by more than one application. The only requirement is that applications have to be created by the same team (as set up in the iTunes Developer Center).
- The **Keychain Groups** includes keys needed by applications that are sharing keychain data. The keychain is beyond the scope of this chapter.

You don't have to change anything from the defaults for this tutorial, so you are ready to go! If you like you can edit the same settings by editing the file **dox.entitlements** that is included in your project.

Checking for iCloud availability

When building an application that makes use of iCloud, the best thing to do is to check the availability of iCloud as soon as the application starts. Although iCloud is available on all iOS 5 and iOS 6 devices, the user might not have configured it. To

avoid possible unintended behaviors or crashes, you should check if iCloud is available before using it. Let's see how this works.

Open up the **SAppDelegate.m** file of the project you have just created and add the following code at the bottom of `application:didFinishLaunchingWithOptions` (before the return statement).

```
id currentToken = [[NSFileManager defaultManager]
    ubiquityIdentityToken];
if (currentToken) {
    NSLog(@"iCloud access on with id %@", currentToken);
} else {
    NSLog(@"No iCloud access");
}
```

Here you use a new method you haven't seen yet called `ubiquityIdentityToken`. This method allows you to retrieve the token associated with the user's iCloud account. If this token is `nil` that means the user has not activated iCloud on his device, otherwise you are good to go and you can use iCloud features in your app.

Note: The token identifies a single user account, so if the user switches accounts by entering new iCloud's credentials, you'll get a separate token. You can use this to tweak your application accordingly.

Compile and run your project (on a device, because iCloud does not work on the simulator), and if all works well, you should see a message in your console like this:

```
iCloud access on with id <1b10b44e 0cc644b0 63843d43 976b46b5
0189c5d3>
```

Note that this method is new to iOS 6 - unlike iOS 5, there is no need to call `URLForUbiquityContainerIdentifier` and pass the id of the ubiquity container.

Note: The project up to this point is in the resources for this chapter as **dox-00**.

iCloud API overview

Before you go on with the code, let's take a few minutes to give an overview of the APIs you'll be using to work with iCloud documents.

To store documents in iCloud, you can do things manually if you'd like, by moving files to/from the iCloud directory with new methods in `NSFileManager` and the new

`NSFilePresenter` and `NSFileCoordinator` classes. However doing this is fairly complex and unnecessary in most cases, because iOS5 has introduced a new class to make working with iCloud documents much easier: `UIDocument`.

`UIDocument` acts as middleware between the file itself and the actual data (which in your case will be the text of a note). In your apps, you'll usually create a subclass of `UIDocument` and override a few methods on it that we'll discuss below.

`UIDocument` implements the `NSFilePresenter` protocol for you and does its work in the background, so the application is not blocked when opening or saving files, and the user can continue working with it. Such a behavior is enabled by a double queue architecture.

The first queue, the main thread of the application, is the one where your code is executed. Here you can open, close and edit files. The second queue is on the background and it is managed by UIKit.

For example let's say you want to open a document, which has been already created on iCloud. You'd send a message to an instance of `UIDocument` like the following:

```
[doc openWithCompletionHandler:^(BOOL success) {
    // Code to run when the open has completed
}];
```

This triggers a 'read' message into the background queue. You can't call this method directly, for it gets called when you execute `openWithCompletionHandler:`. Such an operation might take some time (for example, the file might be very big, or not downloaded locally yet).

In the meantime you can do something else on the user application so the application is not blocked. Once the reading is done you are free to load the data returned by the read operation.

This is exactly where `UIDocument` comes in handy, because you can override the `loadFromContents:ofType:error:` method to read the data into your `UIDocument` subclass. Here's a simplified version what it will look like for your simple notes app:

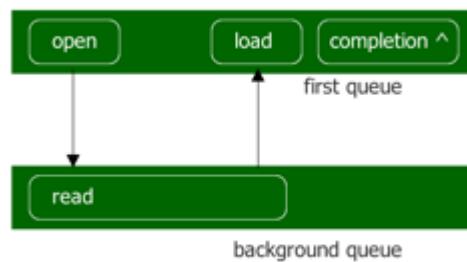
```
- (BOOL)loadFromContents:(id)contents
    ofType:(NSString *)typeName
    error:(NSError **)outError
{
    self.noteContent = [[NSString alloc]
        initWithBytes:[contents bytes]
        length:[contents length]
        encoding:NSUTF8StringEncoding];

    return YES;
}
```

```
}
```

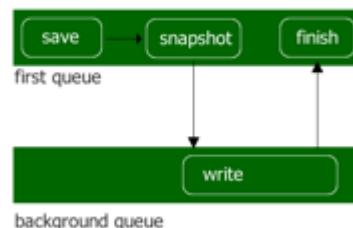
This method is called by the background queue whenever the read operation has been completed. The most important parameter here is `contents`, which is typically an `NSData` containing the actual data which you can use to create or update your model. You'd typically override this method to parse the `NSData` and pull out your document's information, and store it in some instance variables in your `UIDocument` subclass, like shown here.

After `loadFromContents:ofType:error:` completes, you'll receive the callback you provided in the `openWithCompletionHandler:` block, as shown in the diagram below.



To sum up, when you open a file you receive two callbacks: first in your `UIDocument` subclass when data has been read, and secondly when open operation is completely finished.

The write operation is pretty similar and it exploits the same double queue. The difference is that when opening a file you have to parse an `NSData` instance, but while closing you have to convert your document's data to `NSData` and provide it to the background queue.



To save a document, you can either manually initiate the process by writing code, or via the auto saving feature implemented in `UIDocument` (more on this below).

If you want to save manually, you'd call a method like this:

```
[doc saveToURL:[doc fileURL]
    forSaveOperation:UIDocumentSaveForCreating]
```

```
completionHandler:^(BOOL success) {
    // Code to run when the save has completed
}];
```

Just like when opening a file, there is a completion handler that is called when the writing procedure is done. When asked to write, the background queue asks for a snapshot of the contents of your `UIDocument` subclass. This is accomplished by overriding another method of `contentsForType:error:`.

Here you should return an `NSData` instance that describes the current model to be saved. In your notes application, you'll be returning an `NSData` representation of a string as follows:

```
- (id)contentsForType:(NSString *)typeName
                 error:(NSError **)outError
{
    return [NSData dataWithBytes:[self.noteContent UTF8String]
                           length:[self.noteContent length]];
}
```

The rest is taken care of in the background queue, which manages the storing of data. Once done the code in the completion handler will be executed.

For sake of completeness I should mention that in both reading and writing, instead of `NSData` you can use `NSFileWrapper`. While `NSData` is meant to manage flat files `NSFileWrapper` can handle **packages**, that is a directory with files treated as a single file. You'll learn how to use `NSFileWrapper` in the next chapter.

As I mentioned earlier, the save operation can be called explicitly via code or triggered automatically. The `UIDocument` class implements a saveless model, where data is saved automatically at periodic intervals or when certain events occur. This way there is no need for the user to tap a 'save' button anymore, because the system manages that automatically, e.g. when you switch to another document.

Under the hood the `UIKit` background queue calls a method on `UIDocument` called `hasUnsavedChanges`, which returns whether the document is "dirty" and needs to be saved. In case of positive response the document is automatically saved. There is no way to directly set the value for such a method but there are two ways to influence it.

The first way is to explicitly call the `updateChangeCount:` method. This notifies the background queue about changes. As an alternative you can use the undo manager, which is built in the `UIDocument` class. Each instance of this class (or subclasses) has in fact a property `undoManager`. Whenever changes are registered via an undo or redo action the `updateChangeCount:` is called automatically. You'll learn about this topic later in the chapter.

It is important to remember that in either case the propagation of changes might not be immediate. By sending these messages you are only providing 'hints' to the background queue, which will start the update process when it's appropriate according to the device and the type of connection.

Subclassing UIDocument

Now that you have a good overview of `UIDocument`, let's create a subclass for your note application and see how it works!

Create a new file with the **iOS\Cocoa Touch\Objective-C** class template. Name the class `SMNote`, and make it a subclass of `UIDocument`.

To keep things simple, your class will just have a single property to store the note as a string. To add this, replace the contents of **SMNote.h** with the following:

```
@interface SMNote : UIDocument
@property (strong) NSString * noteContent;
@end
```

As you have learned above you have two override points, one when you read and one when you write. Add the implementation of these by replacing **SMNote.m** with the following:

```
#import "SMNote.h"

@implementation SMNote

// Called when the application reads data from the file system
- (BOOL)loadFromContents:(id)contents
    ofType:(NSString *)typeName error:(NSError **)outError
{
    if ([contents length] > 0) {
        self.noteContent = [[NSString alloc]
            initWithBytes:[contents bytes]
            length:[contents length]
            encoding:NSUTF8StringEncoding];
    } else {
        // When the note is created, assign default content
        self.noteContent = @"Empty";
    }

    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"noteModified"
        object:self];
}
```

```
        return YES;
    }

// Called when the application (auto)saves the content of a note
- (id)contentsForType:(NSString *)typeName
    error:(NSError **)outError
{

    if ([self.noteContent length] == 0) {
        self.noteContent = @"Empty";
    }

    return [NSData dataWithBytes:[self.noteContent UTF8String]
                           length:[self.noteContent length]]];
}

@end
```

When you load a file you need a procedure to 'transform' the `NSData` contents returned by the background queue into a string. Conversely, when you save you have to encode your string into an `NSData` object. In both cases you do a quick check and assign a default value in case the string is empty. This happens the first time that the document is created.

Believe it or not, the code you need to model the document is already over! Now you can move to the code related to loading and updating.

Opening an iCloud file

First of all you should decide a file name for your document. For this tutorial, you'll start by creating a single filename. Add the following `#define` at the top of **SMMasterViewController.m**:

```
#define kFILENAME @"mydocument.dox"
```

Next, let's extend the class to keep track of your document, and a metadata query to look up the document in iCloud. Modify **SMMasterViewController.h** to look like the following:

```
#import "SMNote.h"

@class SMDetailViewController;
```

```
@interface SMMasterViewController : UITableViewController

@property (strong, nonatomic)
    SMDetailViewController *detailViewController;
@property (strong, nonatomic) SMNote * doc;
@property (strong, nonatomic) NSMetadataQuery *query;

- (void)loadDocument;
- (void)loadData:(NSMetadataQuery *)query;

@end
```

When the view is loaded you want to call the method `loadDocument`. In **SMMasterViewController.m** you add the following as the last row of `viewDidLoad`.

```
[self loadDocument];
```

Next you define the `loadDocument` method. Let's put it together bit by bit so you can understand the code one piece at a time. Start by adding the following:

```
- (void)loadDocument {

    NSMetadataQuery *query = [[NSMetadataQuery alloc] init];
    _query = query;

}
```

Note that before you can load a document from iCloud, you first have to check what's there. If you ever worked with the Spotlight API on the Mac, you'll be familiar with the class `NSMetadataQuery`. It is a class to represent results of a query related to the properties of an object, such as a file.

In building such a query you have the possibility to specify parameters and scope, i.e. what you are looking for and where. In the case of iCloud files the scope is always `NSMetadataQueryUbiquitousDocumentsScope`. You can have multiple scopes, so you have to build an array containing just one item.

So continue `loadDocument` as follows:

```
- (void)loadDocument {

    NSMetadataQuery *query = [[NSMetadataQuery alloc] init];
    _query = query;
    [query setSearchScopes:[NSArray arrayWithObject:
        NSMetadataQueryUbiquitousDocumentsScope]];

}
```

Now you can provide the parameters of the query. If you ever worked with Core Data or even arrays you probably know the approach. Basically, you build a predicate and set it as parameter of a query/search.

In your case you are looking for a file with a particular name, so the keyword is `NSMetadataItemFSNameKey`, where 'FS' stands for file system. Add the code to create and set the predicate next:

```
- (void)loadDocument {

    NSMetadataQuery *query = [[NSMetadataQuery alloc] init];
    _query = query;
    [query setSearchScopes:[NSArray arrayWithObject:
        NSMetadataQueryUbiquitousDocumentsScope]];

    NSPredicate *pred = [NSPredicate predicateWithFormat:
        @"%K == %@", NSMetadataItemFSNameKey,
        kFILENAME];
    [query setPredicate:pred];

}
```

You might not have seen the `%K` substitution before. It turns out predicates treat formatting characters a bit differently than you might be used to with `NSString`'s `stringWithFormat::`. When you use `%@` in predicates, it wraps the value you provide in quotes. You don't want this for keypaths, so you use `%K` instead to avoid wrapping it in quotes. For more information, see the [Predicate Format String Syntax](#) in Apple's documentation here:

- <https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Predicates/Articles/pSyntax.html>

Now the query is ready to be run, but since it is an asynchronous process you need to set up an observer to catch a notification when it completes. The specific notification you want has a long name:

`NSMetadataQueryDidFinishGatheringNotification`. This is posted when the query has finished gathering info from iCloud.

```
- (void)loadDocument {

    NSMetadataQuery *query = [[NSMetadataQuery alloc] init];
    _query = query;
    [query setSearchScopes:[NSArray arrayWithObject:
        NSMetadataQueryUbiquitousDocumentsScope]];

    NSPredicate *pred = [NSPredicate predicateWithFormat:
        @"%K == %@", NSMetadataItemFSNameKey,
```

```
        NSMetadataItemFSNameKey,
        kFILENAME];
[query setPredicate:pred];

[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(queryDidFinishGathering:)
name:NSMetadataQueryDidFinishGatheringNotification
object:query];

[query startQuery];

}
```

Now that this is in place, add the code for the method that will be called when the query completes:

```
- (void)queryDidFinishGathering:(NSNotification *)notification {
    NSMetadataQuery *query = [notification object];
    [query disableUpdates];
    [query stopQuery];

    [[NSNotificationCenter defaultCenter]
removeObserver:self
name:NSMetadataQueryDidFinishGatheringNotification
object:query];

    _query = nil;

    [self loadData:query];
}
```

Note that once you run a query, if you don't stop it, it runs forever or until you quit the application. Especially in a cloud environment things can change often. It might happen that while you are processing the results of a query, due to live updates, the results change! So it is important to stop this process by calling `disableUpdates` and `stopQuery`. In particular the first prevents live updates and the second allows you to stop a process without deleting already collected results.

You then remove ourselves as an observer to ignore further notifications, and finally call a method to load the document, passing the `NSMetadataQuery` as a parameter.

Add the starter implementation of this method next (add this above `queryDidFinishGathering:`):

```
- (void)loadData:(NSMetadataQuery *)query {  
  
    if ([query resultCount] == 1) {  
  
        NSMetadataItem *item = [query resultAtIndex:0];  
  
    }  
}
```

As you can see here, a `NSMetadataQuery` wraps an array of `NSMetadataItems` which contain the results. In your case, you are working with just one file so you are just interested in the first element.

An `NSMetadataItem` is like a dictionary, storing keys and values. It has a set of predefined keys that you can use to look up information about each file:

- `NSMetadataItemURLKey`
- `NSMetadataItemFSNameKey`
- `NSMetadataItemDisplayNameKey`
- `NSMetadataItemIsUbiquitousKey`
- `NSMetadataUbiquitousItemHasUnresolvedConflictsKey`
- `NSMetadataUbiquitousItemIsDownloadedKey`
- `NSMetadataUbiquitousItemIsDownloadingKey`
- `NSMetadataUbiquitousItemIsUploadedKey`
- `NSMetadataUbiquitousItemIsUploadingKey`
- `NSMetadataUbiquitousItemPercentDownloadedKey`
- `NSMetadataUbiquitousItemPercentUploadedKey`

In your case, you are interested in `NSMetadataItemURLKey`, which points to the URL that you need to build your Note instance. Continue the `loadData:` method as follows:

```
- (void)loadData:(NSMetadataQuery *)query {  
  
    if ([query resultCount] == 1) {  
  
        NSMetadataItem *item = [query resultAtIndex:0];  
        NSURL *url = [item  
                      valueForAttribute:NSMetadataItemURLKey];  
  
        SMNote *doc = [[SMNote alloc] initWithFileURL:url];  
        self.doc = doc;  
  
    }  
}
```

```
}
```

When you create a `UIDocument` (or a subclass of `UIDocument` like `SMNote`), you always have to use the `initWithFileURL:` initializer and give it the URL of the document to open. You call that here, passing in the URL of the located file, and store it away in an instance variable.

Now you are ready to open the note. As explained previously you can open a document with the `openWithCompletionHandler:` method, so continue `loadData:` as follows:

```
- (void)loadData:(NSMetadataQuery *)query {

    if ([query resultCount] == 1) {

        NSMetadataItem *item = [query resultAtIndex:0];
        NSURL *url = [item
                      valueForAttribute:NSMetadataItemURLKey];

        SMNote *doc = [[SMNote alloc] initWithFileURL:url];
        self.doc = doc;

        [self.doc openWithCompletionHandler:^(BOOL success) {
            if (success) {
                NSLog(@"iCloud document opened");
                [self.tableView reloadData];
            } else {
                NSLog(@"failed opening document from iCloud");
            }
        }];
    }
}
```

You can run the app now, and it seems to work, except it never prints out either of the above messages! This is because there is currently no document in your container in iCloud, so the search isn't finding anything (and the result count is 0).

Since the only way to add a document on the iCloud is via an app, you need to write some code to create a document. You will append this to the `loadData:` method that you defined a few seconds ago. When the query returns zero results, you should:

1. Retrieve the local iCloud directory
2. Initialize an instance of document in that directory
3. Call the `saveToURL:forSaveOperation:completionHandler:` method

4. When the save is successful you can call `openWithCompletionHandler:`.

So add an `else` case in `loadData:` as follows:

```
else {

    NSURL *ubiq = [[NSFileManager defaultManager]
                    URLForUbiquityContainerIdentifier:nil];
    NSURL *ubiqPackage = [[ubiq
        URLByAppendingPathComponent:@"Documents"
        URLByAppendingPathComponent:kFILENAME];

    SMNote *doc = [[SMNote alloc] initWithFileURL:
                    ubiqPackage];
    self.doc = doc;

    [doc saveToURL:[doc fileURL]
        forSaveOperation:UIDocumentSaveForCreating
        completionHandler:^(BOOL success) {
        if (success) {
            [doc openWithCompletionHandler:^(BOOL success) {
                NSLog(@"newly created document opened");
                [self.tableView reloadData];
            }];
        }
    }];
}
```

Compile and run your app, and you should see the "new document" message arrive the first time you run it, and "iCloud document opened" in subsequent runs. Big success!

You can even try this on a second device (I recommend temporarily commenting out the `else` case first though to avoid creating two documents due to timing issues), and you should see the "iCloud document opened" message show up on the second device (because the document already exists on iCloud now!)

Now your application is almost ready. The iCloud part is over, and you just need to set up the user interface!

Setting up the user interface

The Xcode project template you chose already set up a bunch of stuff for us. There are two storyboards (both for iPhone and iPad) and the navigation logic is already in place. So whenever you tap a note in the table, its details are shown. You will

add some code to show the a single note in `SMMasterViewController` and you will add a `UITextView` to show the content of the note in `SMDetailViewController`.

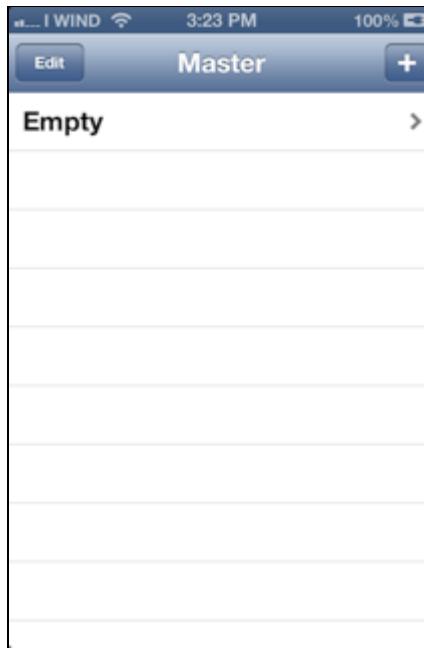
Let's start by modifying the **SMMasterViewController.m**. Tweak the table view data source methods to display the note as follows:

```
- (NSInteger)numberOfSectionsInTableView:  
    (UITableView *)tableView  
{  
    return 1;  
}  
  
- (NSInteger)tableView:(UITableView *)tableView  
 numberOfRowsInSection:(NSInteger)section  
{  
    return 1;  
}  
  
- (UITableViewCell *)tableView:(UITableView *)tableView  
    cellForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    UITableViewCell *cell = [tableView  
dequeueReusableCellWithIdentifier:@"Cell" forIndexPath:indexPath];  
  
    cell.textLabel.text = self.doc.noteContent;  
    return cell;  
}  
  
- (void)tableView:(UITableView *)tableView  
didSelectRowAtIndexPath:(NSIndexPath *)indexPath  
{  
  
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==  
UIUserInterfaceIdiomPad) {  
        self.detailViewController.detailItem = self.doc;  
    }  
}  
- (void)prepareForSegue:(UIStoryboardSegue *)segue  
    sender:(id)sender  
{  
    if ([[segue identifier] isEqualToString:@"showDetail"]) {  
  
        [[segue destinationViewController]  
        setDetailItem:self.doc];  
    }  
}
```

```
}
```

```
}
```

Now you can quickly check if the application is working. Compile and run on one device and you should see that the table is populated with an empty note.



This is the note created by default the first time you run the application. Next you need to tweak the detail view to show the content's note. Open the **SMDetailViewController.h** and edit as follows.

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

@interface SMDetailViewController : UIViewController
<UISplitViewControllerDelegate>

@property (strong, nonatomic) SMNote *detailItem;
@property (weak, nonatomic) IBOutlet UITextView *noteTextView;

@end
```

With respect to the generated code you have simply specified that `detailItem` is of type `SMNote` and you have added an outlet for a text view to display the note's content.

Next switch to **SMDetailViewController.m** and refactor two methods as follows:

```
- (void)setDetailItem:(SMNote *)newDetailItem
```

```

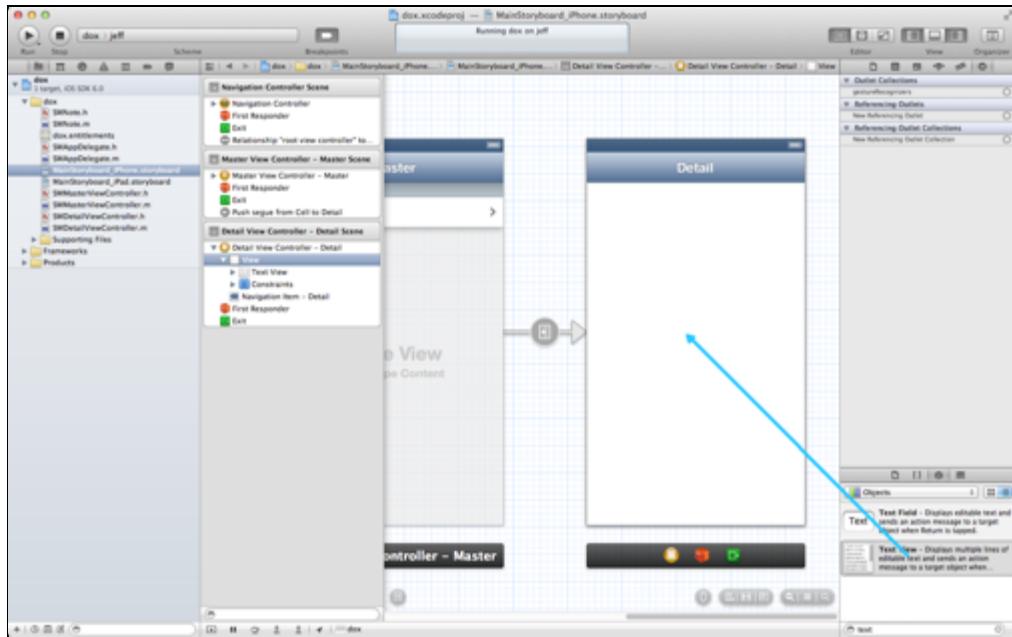
{
    if (_detailItem != newDetailItem) {
        _detailItem = newDetailItem;
        [self configureView];
    }

    if (self.masterPopoverController != nil) {
        [self.masterPopoverController
            dismissPopoverAnimated:YES];
    }
}

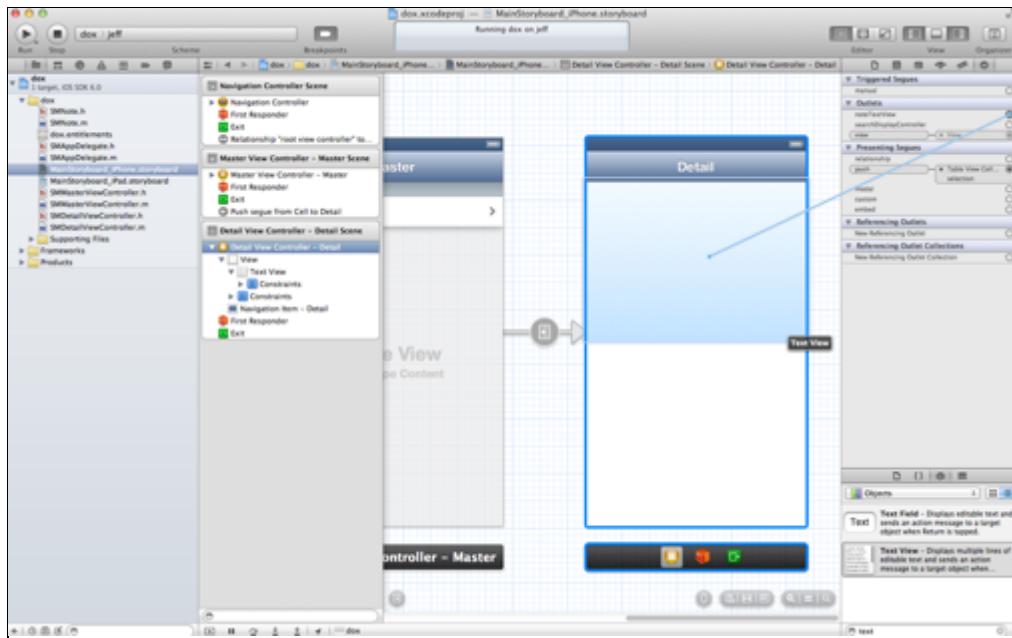
- (void)configureView
{
    if (self.detailItem) {
        self.noteTextView.text = self.detailItem.noteContent;
    }
}

```

Let's now configure the storyboard. Open **MainStoryboard_iPhone.storyboard** and scroll right to see the detail view controller. Delete the existing label and drag a Text View as follows:



Don't forget to link the outlet to the code. Select the controller from the scene on the left and connect the **noteTextView** outlet to the text view.



Repeat the same steps for **MainStoryboard_iPad.storyboard**. Now you are ready for the first real run of your application!

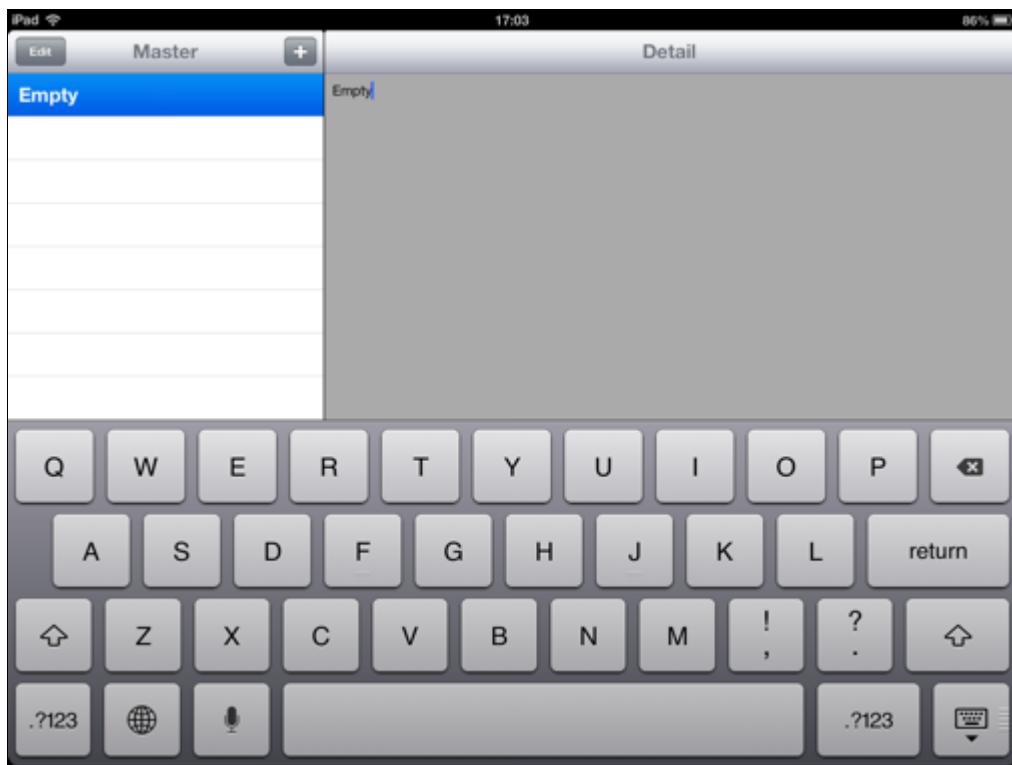
Note: At this point, for testing purposes you should start from a clean configuration.

If you still have data from the previous run of the app you should delete them from Settings/iCloud/Storage & Backup/Manage Storage. Select the item dox and then tap "Edit" on the top right to enable the deletion of items.

After making sure you're at a clean state, install the application on your iPhone. As previously the application will generate the file from scratch. If you tap the note you'll see its content in the detail view controller:

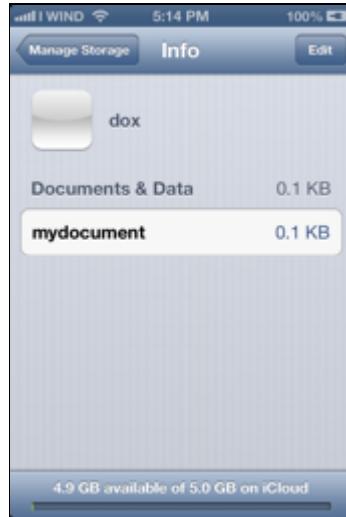


Now install the app on the iPad and run. This time the file won't be created because it's already on iCloud and the application will populate the table view with your item. If you select it the text view will show its content.

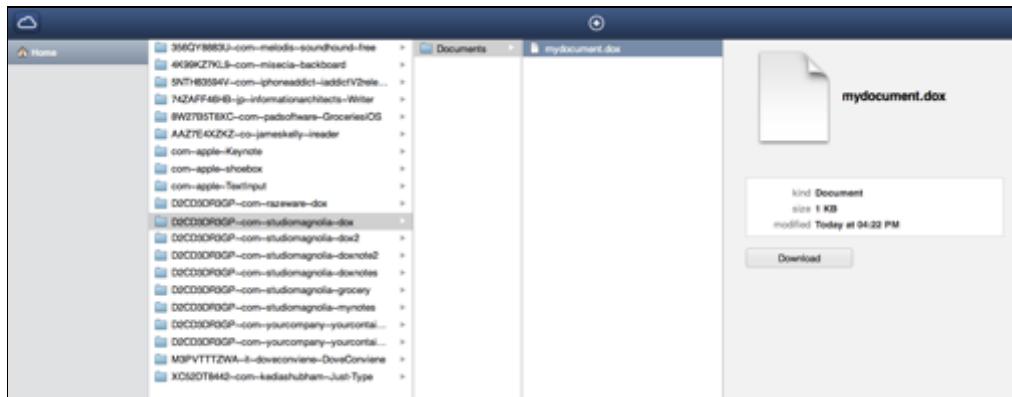


If you change the text, it will not synchronize to the other device yet – you'll do that later.

Another way to check the result is to browse the list of files in your iCloud Console, which is a bit hidden in the menu. Here is the sequence: Settings -> iCloud -> Storage and Backup -> Manage Storage -> Documents & Data -> dox. Here you will see the file as you have named it and the quota it takes on iCloud.



A third way to check the storage on iCloud is to visit: developer.icloud.com. Enter your credentials and you'll see (read only) all the data stored on iCloud by the applications you have installed.



Congrats! You have built your first iCloud-enabled application!

Note: The project up to this point is in the resources for this chapter as **dox-01**.

Handling multiple documents

Cool, your example works and you are a bit more acquainted with the capabilities of iCloud. But what you have right now isn't enough to impress your users, or build an application that makes sense. Who wants to manage just one document?!

Next you are going to extend your application to manage more than one document at a time. The most natural development of your current prototype is to transform it into a notes application, as follows:

- Each note will have a unique id
- The table will show a list of notes
- Users can then edit the content
- The list of notes is updated when you launch the application

To do this, you will have to reorganize your code a bit. The architecture of view controllers will be the same, one master and one detail view. Let's start with the master controller.

You add an array to keep track of all the notes. Open **SMMasterViewController.h** and replace the contents with the following:

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

@class SMDetailViewController;

@interface SMMasterViewController : UITableViewController

@property (strong, nonatomic)
    SMDetailViewController *detailViewController;
@property (strong, nonatomic) NSMetadataQuery *query;
@property (strong, nonatomic) NSMutableArray *notes;

- (void)loadDocument;

@end
```

You have added an array to store the notes, while the rest is unchanged. Next switch over to **SMMasterViewController.m** and change `viewDidLoad` like this:

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

    self.navigationItem.leftBarButtonItem = self.editButtonItem;
```

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

self.detailViewController = (SMDetailViewController *)
[[self.splitViewController.viewControllers lastObject]
 topViewController];

self.notes = [NSMutableArray array];

[[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(loadDocument)
 name:UIApplicationDidBecomeActiveNotification
 object:nil];

}
```

Here you have added the initialization for the new array and an observer to listen when the application becomes active, so that each time you resume it notes get reloaded.

Unlike the previous project that had just one document (so always used the same filename each time), this time you're storing multiple documents (one for each note created), so you need a way to generate unique file names. As an easy solution, you will use the creation date of the file and prepend the 'Note_' string.

So change the implementation of `insertNewObject:` as follows:

```
- (void)insertNewObject:(id)sender
{
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"yyyyMMdd_hhmmss"];

    NSString *fileName = [NSString stringWithFormat:@"Note_%@",
        [formatter stringFromDate:[NSDate date]]];

    NSURL *ubiq = [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil];

    NSURL *ubiquitousPackage =
    [[ubiq URLByAppendingPathComponent:@"Documents"]
        URLByAppendingPathComponent:fileName];
```

```
SMNote *doc = [[SMNote alloc]
    initWithFileURL:ubiquitousPackage];

[doc saveToURL:[doc fileURL]
    forSaveOperation:UIDocumentSaveForCreating
    completionHandler:^(BOOL success) {

    if (success) {
        [self.notes addObject:doc];
        [self.tableView reloadData];
    }
}];
}
```

You should be pretty familiar with this code. The file name is generated by combining the current date and hour. You call the `saveToURL:forSaveOperation:completionHandler:` method and, in case of success, you add the newly created note to the array which populates the table view.

Note: If you target from iOS6 up, instead of using date and hour, you can use the new class `NSUUID` to generate a unique id.

Almost done with the ability to add notes - just need to add the code to populate the table view with the contents of the notes array. Implement the table view data source and delegate methods like the following:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return self.notes.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"Cell"
        forIndexPath:indexPath];
    cell.textLabel.text = self.notes[indexPath.row].title;
    cell.detailTextLabel.text = self.notes[indexPath.row].content;
    return cell;
}
```

```

        forIndexPath:indexPath];
    SMNote * note = self.notes[indexPath.row];
    cell.textLabel.text = note.fileURL.lastPathComponent;
    return cell;
}

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad)
    {
        SMNote *selectedNote = self.notes[indexPath.row];
        self.detailViewController.detailItem = selectedNote;
    }
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showDetail"]) {

        NSIndexPath *indexPath = [self.tableView
                                  indexPathForSelectedRow];
        SMNote *selectedNote = self.notes[indexPath.row];
        self.detailViewController.detailItem = selectedNote;
        [[segue destinationViewController]
         setDetailItem:selectedNote];

    }
}
}

```

Now it's time to tweak the loading methods to deal with the new data model. Still in the **SMMasterViewController.m** let's go to the `loadDocument` method. You will follow a similar strategy to what you did earlier when loading a single note. However, this time you don't know the exact file name, so you have to tweak your search predicate to look for a file name like "Note_*".

```

- (void)loadDocument {

    NSURL *ubiq = [[NSFileManager defaultManager]
                   URLForUbiquityContainerIdentifier:nil];

    if (ubiq) {

```

```
self.query = [[NSMetadataQuery alloc] init];
[self.query setSearchScopes:
 [NSArray arrayWithObject:
 NSMetadataQueryUbiquitousDocumentsScope]];

NSPredicate *pred = [NSPredicate predicateWithFormat:
 @"%K like 'Note_*'",
 NSMetadataItemFSNameKey];
[self.query setPredicate:pred];
[[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(queryDidFinishGathering:)
 name:NSMetadataQueryDidFinishGatheringNotification
 object:self.query];

[self.query startQuery];

} else {

 NSLog(@"No iCloud access");

}

}
```

You might be tempted to place a call to this method in `viewDidLoad`. That would be correct, but that is executed on initial app startup. You want to reload data really each time the app is opened (even from the background), and you have already added the observer to listen when the application becomes active.

Next you need to tweak the `loadData:` method.

```
- (void)loadData:(NSMetadataQuery *)query {

[self.notes removeAllObjects];

for (NSMetadataItem *item in [query results]) {

NSURL *url = [item
               valueForAttribute:NSMetadataItemURLKey];
SMNote *doc = [[SMNote alloc] initWithFileURL:url];

[doc openWithCompletionHandler:^(BOOL success) {
    if (success) {

        [self.notes addObject:doc];
    }
}];
```

```
        [self.tableView reloadData];

    } else {
        NSLog(@"failed to open from iCloud");
    }

}];
}
```

This method populates the array of notes according to the results of the query. The implementation here fully reloads the list of notes as returned by iCloud.

The last step is to add a “Save” button to the detail view so the user can store changes to the selected note. Open **SMDetailViewController.m** and change viewDidLoad as follows:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [self configureView];
    self.noteTextView.backgroundColor =
        [UIColor lightGrayColor];

    UIBarButtonItem *saveButton = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemSave
        target:self
        action:@selector(saveEdits:)];
}

self.navigationItem.rightBarButtonItem = saveButton;
```

This will add a button named “Save” on the top right of the detail view. Add the method that it will call when you tap it next:

```
- (void) saveEdits:(id)sender {

    self.detailItem.noteContent = self.noteTextView.text;

    [self.detailItem saveToURL:[self.detailItem fileURL]
        forSaveOperation:UIDocumentSaveForOverwriting
        completionHandler:nil];

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad) {
```

```
    self.detailItem = nil;
    self.noteTextView.text = @"";
} else {
    [self.navigationController popViewControllerAnimated:YES];
}
}
```

Note to save the edits here, you call the `saveToURL:forSaveOperation:completeHandler` method on `UIDocument`. There is another way to get it to save automatically which you'll learn about in the next chapter.

That's it! Compile and run the app on one device, and create a few notes. Then edit one or two of them, and be sure to tap Save when you're done. Then start the app on another device, and see that the table is correctly populated with the same notes!

Note: The project up to this point is in the resources for this chapter as **dox-02.**

Where to go from here?

Congratulations, you now have hands-on experience with the basics of using iCloud and the new `UIDocument` class to create an iCloud-enabled app with multi-document support.

You've just scratched the surface of iCloud - stay tuned for the next chapter, where you'll learn how to handle conflict resolution, use `NSFileWrapper`, store simple key-value pairs, and use Core Data with iCloud!

Chapter 7: Intermediate iCloud

By Cesare Rocchi

In the last chapter, you learned the basics of synchronizing your application data with iCloud using the new `UIDocument` class. You also learned how to work with multiple documents and search the current documents on iCloud with `NSMetadataQuery`.

In this chapter, you're going to learn a lot more about iCloud, such as:

- How to switch between iCloud and local storage
- How to handle multi-file documents with `NSFileWrapper`
- How to easily store key-value information on iCloud
- How to delete files from iCloud
- How to deal with file changes and conflicts
- How to use the undo manager with iCloud
- How to export data (via URLs) with iCloud
- How to use Core Data with iCloud

By the time you are done with this chapter, you will have a great overview of most of the ways to use iCloud, and will be ready to put this to use in your own apps!

This chapter begins with the project that you created in the previous chapter. If you don't have this project already, you can start from the one in the resources for this chapter, named **dox-02**.

Toggling iCloud Support

So far you have assumed that your users are willing to use iCloud and that it is available when the application runs.

However, it could be the case that a user has iCloud disabled. Or you might want to give the user a choice whether to save their data locally or on iCloud (even if iCloud is enabled). For example, a user might want to start using your application by just

saving notes locally. Later he might change his mind and turn iCloud on. How do you cope with this case? Let's see.

First, you have to update your user interface by adding a switch button that activates iCloud storage. You will place it in the toolbar at the bottom of the master view controller. The intended result is shown in the following screenshot:



To achieve this you have to refactor your code a bit.

Let's first add a boolean property to keep track when the iCloud functionality is enabled. Add the following properties to **SMMasterViewController.h**:

```
@property (strong, nonatomic) UISwitch *cloudSwitch;
@property (nonatomic) BOOL useiCloud;
```

Next in **SMMasterViewController.m** add a static string to use as a key for storing the switch button value (right after the `@implementation` statement):

```
static NSString * const useiCloudKey = @"useiCloud";
```

Next, add the following code to the beginning of `viewDidLoad` (after the call to `[super viewDidLoad]`) to initialize the `_useiCloud` Boolean and to add the switch to the toolbar:

```
_useiCloud = [[NSUserDefaults standardUserDefaults]
              boolForKey:useiCloudKey];

self.cloudSwitch = [[UISwitch alloc]
                   initWithFrame:CGRectMake(40, 4, 80, 27)];
self.cloudSwitch.on = _useiCloud;
```

```
[self.cloudSwitch addTarget:self
                      action:@selector(enableDisableiCloud:)
            forControlEvents:UIControlEventValueChanged];

UIBarButtonItem *flexSpace1 = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
                      target:nil
                      action:NULL];

UIBarButtonItem *iCloudSwitchItem = [[UIBarButtonItem alloc]
initWithCustomView:self.cloudSwitch];

UIBarButtonItem *flexSpace2 = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
                      target:nil
                      action:NULL];

self.toolbarItems = @[flexSpace1,iCloudSwitchItem, flexSpace2];
```

When the view is loaded you retrieve the value previously selected by the user and assign it to the boolean (you'll see how to store it later). To make the switch button centered you surround it with two flexible spaces (`flexSpace1` and `flexSpace2`).

Next write the method that will get called when the switch is toggled:

```
- (void) enableDisableiCloud: (id) sender {
    self.useiCloud = [sender isOn];
}
```

The navigation controller's toolbar is hidden by default, so implement `viewWillAppear` to make it visible:

```
- (void) viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [self.navigationController setToolbarHidden:NO];
}
```

The last step for your first milestone is to persist the value of the switch button. You can do this by overriding the setter of the boolean property, as follows:

```
- (void) setUseiCloud:(BOOL)val {
    if (_useiCloud != val) {
        _useiCloud = val;
        [[NSUserDefaults standardUserDefaults]
         setBool:_useiCloud
         forKey:useiCloudKey];
        [[NSUserDefaults standardUserDefaults] synchronize];
    }
}
```

```
}
```

Build and run, and you'll see the switch button now appears at the bottom of the app, its value is persisted even when you quit or restart the application.



You can play with the application to verify that the value of both the switch button and the `_useicloud` variable are the same values set before quitting or putting the application in background.

Now that the user interface is in place, let's move on to the iCloud logic.

Working Off-cloud

Describing the application from now on can be quite complex since code is pretty intertwined to cover both iCloud-on and iCloud-off cases. To make things simpler, you will start with the case where the user starts working on notes without synching to iCloud.

You are about to write two helper methods to make managing this easier:

- `localNotesURL`: this will return the URL of the directory where local notes are stored (the Documents directory).
- `ubiquitousNotesURL`: this will return the URL of the local iCloud directory (outside of the app directory), used by the daemon to synch with the remote location.

This is an extremely important distinction. When you switch on iCloud the notes will be moved from a local folder (e.g. Documents) to another iCloud-enabled local folder. On the contrary, when you switch off iCloud the notes will be moved from the iCloud-enabled folder to the local one.

It is important to stress that there is no way to change a property of a directory to make it iCloud enabled/disabled. So you need to have two separate URLs to cover both the off and the on case.

Add the implementations for these methods at the bottom of **SMMasterViewController.m**:

```
- (NSURL *) localNotesURL {
    return [[[NSFileManager defaultManager]
        URLsForDirectory:NSDocumentDirectory
        inDomains:NSUTFUserDomainMask] lastObject];
}

- (NSURL *) ubiquitousNotesURL {
    return [[[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil]
        URLByAppendingPathComponent:@"Documents"];
}
```

The first returns a URL pointing to the Documents directory inside your application sandbox. The second returns the ubiquity container that you are already familiar with.

So let's get back to your use case. The user starts the application, iCloud is off and the list of notes is empty. In the last chapter you already implemented a method to add a note. You just have to modify it a bit to cover the iCloud-off case. To do this, you change the base URL according to the value of the `_useicloud` boolean property.

By default you initialize it as local and you change it if iCloud is enabled. The rest of the method is the same of the previous project.

```
- (void)insertNewObject:(id)sender
{
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"yyyyMMdd_hhmmss"];

    NSString *fileName = [NSString stringWithFormat:@"Note_%@",
        [formatter stringFromDate:
            [NSDate date]]];
}

NSURL *baseURL = [self localNotesURL];

if (_useicloud) {
    baseURL = [self ubiquitousNotesURL]; // iCloud url
}
```

```

NSURL *noteFileURL = [baseURL
                      URLByAppendingPathComponent:fileName];
SMNote *doc = [[SMNote alloc] initWithFileURL:noteFileURL];

[doc saveToURL:[doc fileURL]
forSaveOperation:UIDocumentSaveForCreating
completionHandler:^(BOOL success) {

    if (success) {

        [self.notes addObject:doc];
        [self.tableView reloadData];

    }

}];

}

```

Now you can do a first test. Run the application, make sure iCloud storage is off and add a note. Then check your iCloud storage on your device (via Settings/iCloud/Storage & Backup/Manage Storage) and verify the note is not saved. Cool, first goal achieved!

However there's still a problem. If you quit and reopen the application you won't see the document you added (but you will see anything that might be in iCloud). Where'd your file go?

The local file is still there but in the `loadDocument` method you are searching for iCloud documents, whether iCloud is enabled or not by the switch. You need to rework the method to cover the iCloud-off case. Modify the method so it looks like the following:

```

- (void)loadDocument {

    NSURL *ubiq = [[NSFileManager defaultManager]
                   URLForUbiquityContainerIdentifier:nil];

    if (ubiq && _useiCloud) { // iCloud is on

        self.query = [[NSMetadataQuery alloc] init];
        [self.query setSearchScopes:[NSArray arrayWithObject:
                               NSMetadataQueryUbiquitousDocumentsScope]];

        NSPredicate *pred = [NSPredicate predicateWithFormat:
                           @"%K like 'Note_*'",
                           NSMetadataItemFSNameKey];
        [self.query setPredicate:pred];
    }
}

```

```
[ [NSNotificationCenter defaultCenter]
addObserver:self
    selector:@selector(queryDidFinishGathering:)
        name:NSMetadataQueryDidFinishGatheringNotification
    object:self.query];

[self.query startQuery];

} else { // iCloud switch is off or iCloud not available

[self.notes removeAllObjects];
NSArray *arr = [[NSFileManager defaultManager]
    contentsOfDirectoryAtURL:[self localNotesURL]
    includingPropertiesForKeys:nil
    options:0
    error:NULL];

for (NSURL *filePath in arr) {

    SMNote *doc = [[SMNote alloc]
        initWithFileURL:filePath];
    [self.notes addObject:doc];
    [self.tableView reloadData];

}

}
}
```

In the “iCloud off” case the retrieval is not asynchronous, so you can just cycle over the returned array, build instances of notes according to the array and reload the content of the table view.

Build and run the application, and this time you should see the note you created earlier in local storage!

At this point, you have an application that allows you to store documents locally or in iCloud. However, there's no way to move the local documents to iCloud (or vice versa). So let's tackle that next!

Turning on iCloud

In theory this last step is simple, since to move a document to/from the local iCloud directory is just a matter of calling a single method:

```
- (BOOL)setUbiquitous:(BOOL)flag
    itemAtURL:(NSURL *)url
destinationURL:(NSURL *)destinationURL
    error:(NSError **)error
```

The idea is that you want to call this method whenever the user toggles the switch, to move documents between local and iCloud storage.

However, there is a complication. If you check out the documentation for this method, it clearly says "Do not call this method from your application's main thread. This method performs a coordinated write operation on the file you specify, and calling this method from the main thread can trigger a deadlock with the file presenter." Ouch, sounds like a problem you want to avoid!

To resolve this problem, the documentation suggests performing this operation on a secondary thread. Here you have two options: using a Grand Central Dispatch (GCD) queue or an `NSOperationQueue`. You use GCD in many other places within this book, so to switch things up you'll use `NSOperationQueue` this time.

In essence you will run some code in the background to move all the notes from the local folder to the ubiquity container that the iCloud daemon keeps in sync with data on the cloud. You do this when the user taps the switch.

So modify the `setUseiCloud:` method to the following:

```
- (void) setUseiCloud:(BOOL)val {
    if (_useiCloud != val) {
        _useiCloud = val;

        [[NSUserDefaults standardUserDefaults] setBool:_useiCloud forKey:useiCloudKey];
        [[NSUserDefaults standardUserDefaults] synchronize];

        NSOperationQueue *iCloudQueue = [NSOperationQueue new];
        NSInvocationOperation *oper =
            [[NSInvocationOperation alloc]
                initWithTarget:self
                selector:@selector(setNotesUbiquity)
                object:nil];

        [iCloudQueue addOperation:oper];
    }
}
```

Here you instantiate a `NSOperationQueue` and you add to that an instance of `NSInvocationOperation`. The operation has the controller as target and a selector

(`setNotesUbiquity`) where the actual moving of files is performed. By adding this operation to the queue, it will run in the background.

Next implement `setNotesUbiquity` as the following:

```
- (void) setNotesUbiquity {
    NSURL *baseUrl = [self localNotesURL];

    if (_useiCloud)
        baseUrl = [self ubiquitousNotesURL];

    for (SMNote *note in self.notes) {
        NSURL *destUrl = [baseUrl URLByAppendingPathComponent:
                           [note.fileURL lastPathComponent]];
        NSLog(@"note.fileURL = %@", note.fileURL);
        NSLog(@"destUrl = %@", destUrl);

        [[NSFileManager defaultManager]
            setUbiquitous:_useiCloud
            itemAtURL:note.fileURL
            destinationURL:destUrl
            error:NULL];
    }

    [self performSelectorOnMainThread:@selector(ubiquityIsSet)
                           withObject:nil
                           waitUntilDone:YES];
}

}
```

As previously discussed, you have to assign a different folder according to the value of the `_useiCloud` boolean. Then you cycle over the list of notes and for each one you call the `NSFileManager` method described above.

You should remember to notify the main thread that the operation on the secondary thread is complete. For now, just implement the callback to log out the results as follows:

```
- (void) ubiquityIsSet {
    NSLog(@"notes are now ubiq? %i", _useiCloud);
}
```

In your apps, you might want to display a spinner to indicate ongoing activity, and then you could hide it in this callback.

In the implementation of `setNotesUbiquity` you have included two log statements to show both the current and the destination URL of each note, so you can see in the

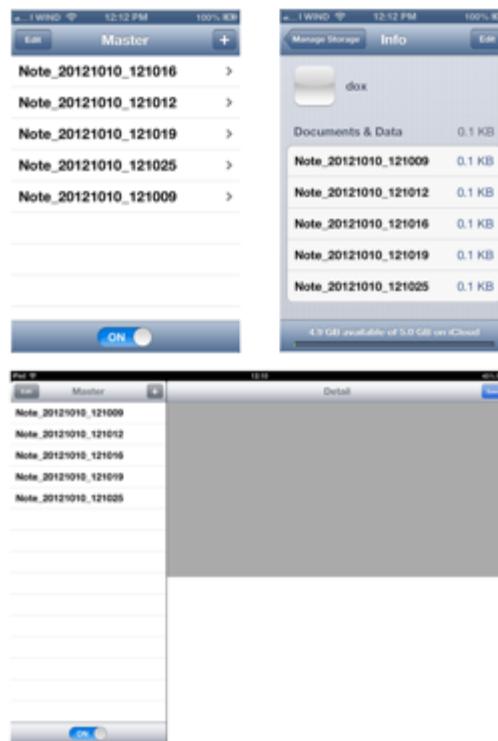
console that they are different. A local document not synched with iCloud has a URL like this (a subdirectory of the application's folder):

```
file:///localhost/var/mobile/Applications/75C93597-DD34-4F33-BE1D-  
DA092EBCF1E2/Documents/Note_20121020_121016
```

A document synched with iCloud has a different URL (outside of the application's directory):

```
file:///localhost/private/var/mobile/Library/Mobile%20Documents/D2C  
D3DR3GP~com~studiomagnolia~dox/Documents/Note_20121020_121016
```

Build and run the application on one device, create a few notes and turn on iCloud. Notice the user interface is not locked and you can continue to work. Then run the app on another device. Turn on iCloud, put the app in background and then reopen it. You'll see the notes you created locally are correctly synchronized!



But what happens if you turn iCloud off? Of course, notes are moved to the local directory. What happens to those on the iCloud account? They are gone!

Turning off iCloud within an application does not mean to unhook the synchronization, as if you were offline. It means that all the documents stored by your application on the iCloud account get deleted, though they still persist on the local directory.

The user might not expect this (because it would mean that other devices would no longer have access to the shared files). So it's a good idea to notify the user before making that choice, by means of an alert view.

Let's go for a quick refactoring. Edit **SMMasterViewController.h** to mark the interface as implementing the `UIAlertViewDelegate` protocol:

```
@interface SMMasterViewController : UITableViewController  
    <UIAlertViewDelegate>
```

Then switch to **SMMasterViewController.m** and add a new method to wrap the "migration" procedure, which will be performed on a secondary thread:

```
- (void) startMigration {  
  
    NSOperationQueue *iCloudQueue = [NSOperationQueue new];  
    NSInvocationOperation *oper =  
        [[NSInvocationOperation alloc]  
            initWithTitle:self  
            selector:@selector(setNotesUbiquity)  
            object:nil];  
    [iCloudQueue addOperation:oper];  
  
}
```

The next step is to refactor `setUseiCloud:` one more time to show an alert view if the user tries to turn off iCloud:

```
- (void) setUseiCloud:(BOOL)val {  
  
    if (_useiCloud != val) {  
  
        _useiCloud = val;  
        [[NSUserDefaults standardUserDefaults]  
            setBool:_useiCloud  
            forKey:useiCloudKey];  
        [[NSUserDefaults standardUserDefaults] synchronize];  
  
        if (! _useiCloud) {  
  
            UIAlertView *iCloudAlert =  
                [[UIAlertView alloc]  
                    initWithTitle:@"Attention"  
                    message:@"This will delete notes from your  
iCloud account. Are you sure?"  
                    delegate:self  
                    cancelButtonTitle:@"No"  
                    otherButtonTitles:@"OK"];  
            [iCloudAlert show];  
        }  
    }  
}
```

```
        otherButtonTitles:@"Yes", nil];  
  
        [iCloudAlert show];  
  
    } else {  
  
        [self startMigration];  
  
    }  
}
```

Finally you implement the callback from the alert view:

```
- (void)alertView:(UIAlertView *)alertView  
clickedButtonAtIndex:(NSInteger)buttonIndex {  
    if (buttonIndex == 0) {  
        [self.cloudSwitch setOn:YES animated:YES];  
        _useiCloud = YES;  
    } else {  
        [self startMigration];  
    }  
}
```

From now on, when the user wants to turn off iCloud synchronization, she will receive an alert as in the following screenshot:



Congratulations, you now know how to turn iCloud on and off within an application, allowing the user to choose whether to store locally or with iCloud (and change their mind at any point!)

Note: The project up to this point is in the resources for this chapter as **dox-03.**

Now is a good time to take a break because you're about to move on to a new topic. But don't go away, as there's a lot left to explore - such as multi-file documents, key-value pairs, conflict resolution, Core Data, and more!

Custom Documents

In the examples so far you have always opted for a one-to-one mapping, one note per file. Although this might seem natural to developers, end users might not be comfortable with that. For example, let's say a user wants to delete your application's data using the iCloud storage editor in Settings, but your application list is crowded with tons of weirdly named small files. That might be quite annoying to have to delete each one individually!

In this part you are going to create an alternate version of your app that stores all of the notes in a single file on iCloud. In the process of doing this, you'll also learn how you can create a document that consists of multiple files using `NSFileWrapper` (such as perhaps notes data plus image data) and how to register your app as the owner of a custom document type.

Rather than refactoring your existing app, I have prepared a starter project, called **dox-4-starter**. It is a very basic project that includes:

- A project configured with an iCloud-enabled application id and its related settings (as explained in the previous chapter).
- A check in `application:didFinishLaunchingWithOptions:` to see if iCloud is enabled
- Two storyboards for iPhone and iPad, using the same user interface hierarchy of previous projects: `UINavigationController`, `SMMasterViewController`, `SMDetailViewController`
- The "+" action to add a new note from the master view controller
- The loading of notes triggered when the application becomes active
- The `saveEdit:` action in the detail view controller at the top right
- A text field in detail view controller to show or edit the content of a note.
- `SMNote` as an empty class, extending `NSObject`.

This is all you need to build an app to focus on iCloud features. Have a look at the code, and update the Bundle identifier and entries in the Entitlements section to match your App ID. Moreover, make sure there are no notes on iCloud from previous runs, because you want to start from a clean situation.

Then run it and play with it a bit to get familiar with the structure of the application!

Modeling a Custom Document

Now it's time to focus on the way you model data. Unlike previous projects here you need two classes: one to manage the single note, and one to manage both iCloud interaction and the list of notes.

We will call the first class **SMNote**. It is a simple class just to store data. Since data will have to persist on disk (and then iCloud) the class has to implement the **NSCoding** protocol. By implementing this protocol you can encode and decode information into a data buffer. For each note you will keep track of the following properties:

- id
- content
- creation date
- update date

Open **SMNote.h** and replace the contents of the file with the following:

```
@interface SMNote : NSObject <NSCoding>

@property (copy, nonatomic) NSString *noteId;
@property (copy, nonatomic) NSString *noteContent;
@property (strong, nonatomic) NSDate *createdAt;
@property (strong, nonatomic) NSDate *updatedAt;

@end
```

Next switch to **SMNote.m** and replace it with the following implementation:

```
#import "SMNote.h"

@implementation SMNote

- (id) init {
    if (self = [super init]) {
        NSDateFormatter *formatter =
        [[NSDateFormatter alloc] init];
        [formatter setDateFormat:@"yyyyMMdd hhmmss"];
        _noteId = [NSString stringWithFormat:@"Note_%@",
                   [formatter stringFromDate:[NSDate date]]];
    }
    return self;
}

#pragma mark NSCoding methods
```

```

- (id)initWithCoder:(NSCoder *)aDecoder {
    if ((self = [super init])) {
        _noteId = [aDecoder decodeObjectForKey:@"noteId"];
        _noteContent = [aDecoder
                        decodeObjectForKey:@"noteContent"];
        _createdAt = [aDecoder decodeObjectForKey:@"createdAt"];
        _updatedAt = [aDecoder decodeObjectForKey:@"updatedAt"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:self.noteId
                  forKey:@"noteId"];
    [aCoder encodeObject:self.noteContent
                  forKey:@"noteContent"];
    [aCoder encodeObject:self.createdAt
                  forKey:@"createdAt"];
    [aCoder encodeObject:self.updatedAt
                  forKey:@"updatedAt"];
}
@end

```

This assigns an id automatically built according to the creation time. The two methods needed by the `NSCoding` allow the decoding and encoding of the object when loading/saving.

Next you'll create another class to keep track of all the notes created. Since this class will manage the interaction with iCloud, it will extend `UIDocument`.

Create a file with the **iOS\Cocoa Touch\Objective-C class** template, name the class `SMNotesDocument` and make it a subclass of `UIDocument`. Then replace **SMNotesDocument.h** with the following:

```

#import <UIKit/UIKit.h>
#import "SMNote.h"

@interface SMNotesDocument : UIDocument

@property (nonatomic, strong) NSMutableArray *entries;
@property (nonatomic, strong) NSFileWrapper *fileWrapper;

- (NSInteger) count;
- (void) addNote:(SMNote *) note;
- (SMNote *) entryAtIndex:(NSUInteger) index;

```

```
@end
```

Here you create an `NSFileWrapper` so you can build up a single document from potentially many files, and declare some methods to manage the array of notes.

Next switch to **SMNotesDocument.m** and replace it with the following:

```
#import "SMNotesDocument.h"

@implementation SMNotesDocument

- (id)initWithFileURL:(NSURL *)url {
    if ((self = [super initWithFileURL:url])) {
        _entries = [[NSMutableArray alloc] init];
        [[NSNotificationCenter defaultCenter]
            addObserver:self
            selector:@selector(noteChanged)
            name:@"com.studiomagnolia.noteChanged"
            object:nil];
    }
    return self;
}

@end
```

Here you override the `initWithFileURL:` method to initialize the array of entries, and you set up an observer to wait for a notification when a note has changed. The notification will trigger a selector, which in turn will save the document. Add the code for this next:

```
- (void) noteChanged {
    [self saveToURL:[self fileURL]
    forSaveOperation:UIDocumentSaveForOverwriting
    completionHandler:^(BOOL success) {
        if (success) {
            NSLog(@"note updated");
        }
    }];
}
```

Next you need to add a couple methods to wrap the notes array:

```
- (SMNote *)entryAtIndex:(NSUInteger)index{
    if (index < _entries.count) {
        return [_entries objectAtIndex:index];
    } else {
```

```
        return nil;
    }
}

- (NSInteger) count {
    return self.entries.count;
}
```

The `entryAtIndex:` method will be needed when you populate the table view. It simply returns the entry at a given index in the notes array. The `count` method is even simpler; it just returns the count of the array.

Next add the `addNote:` method, which you'll use to add a note instance to the array and save the document:

```
- (void) addNote:(SMNote *) note {
    [_entries addObject:note];
    [self saveToURL:[self fileURL]
    forSaveOperation:UIDocumentSaveForOverwriting
    completionHandler:^(BOOL success) {
        if (success) {
            NSLog(@"note added and doc updated");
            [self openWithCompletionHandler:^(BOOL success) {}];
        }
    }];
}
```

Now you are left with the two methods to manage reading and writing on iCloud. This is the crucial step for this application.

Unlike the previous chapter, where you stored just a string, here you have a more complex data model: an array of `SMNote` instances. To encode this set of objects you will use an `NSKeyedArchiver`, which allows converting objects that implement the `NSCoding` protocol into `NSData`, which can be then stored in a file. To use an `NSKeyedArchiver`, you perform the following steps:

1. Create a buffer of mutable data.
2. Initialize an archiver with the buffer.
3. Call `encodeObject:forKey:` on the archiver, passing in the objects to encode. It will convert the objects into the mutable data.

Here's what the code would look like to encode your list of note entries into an `NSMutableData`:

```
NSMutableData *data = [NSMutableData data];
NSKeyedArchiver *arch =
    [[NSKeyedArchiver alloc] initForWritingWithMutableData:data];
```

```
[arch encodeObject:_entries forKey:@"entries"];
[arch finishEncoding];
```

Once you have the `NSMutableData`, its time to pass the buffer to an `NSFileWrapper`. This class manages attributes related to a file. Moreover `NSFileWrappers` can be nested (so you can contain documents with multiple files inside).

For example, you might want to store notes and perhaps images in two separate wrappers, which are in turn wrapped by a 'root' `NSFileWrapper`. To do this you need to:

1. Create a mutable dictionary.
2. Initialize a wrapper with the buffer data of notes
3. Add the wrapper to the dictionary with a key.
4. Build another "root" file wrapper initialized with the dictionary.

Now you have enough information to build the method to generate your file wrapper, so add the following new method:

```
- (id)contentsForType:(NSString *)typeName
                 error:(NSError **)outError {

    NSMutableDictionary *wrappers =
    [NSMutableDictionary dictionary];

    NSMutableData *data = [NSMutableData data];
    NSKeyedArchiver *arch =
        [[NSKeyedArchiver alloc]
         initForWritingWithMutableData:data];
    [arch encodeObject:_entries forKey:@"entries"];
    [arch finishEncoding];

    NSFileWrapper *entriesWrapper =
    [[NSFileWrapper alloc] initRegularFileWithContents:data];

    [wrappers setObject:entriesWrapper forKey:@"notes.dat"];
    // here you could add another wrapper for other resources,
    // like images
    NSFileWrapper *res =
    [[NSFileWrapper alloc]
     initDirectoryWithFileWrappers:wrappers];

    return res;
}
```

In this app you only need the document to contain a single set of files, but this shows you how you could easily add multiple files if you need to.

Be sure to remember the keys used to encode objects ("entries" and "notes.dat"), since you'll need them when it is time to read and decode data.

Now it's time to write the `loadFromContents:ofType:error:` method, which will be called when you read data. It does the exact opposite of the previous method:

1. Unfolds the main wrapper in a dictionary.
2. Retrieves the wrapper of notes by using the same key ('notes.dat').
3. Builds a data buffer from the wrapper.
4. Initializes an `NSKeyedUnarchiver` with the buffer.
5. Decodes the buffer into the array of entries.

Implement the method as follows:

```
- (BOOL)loadFromContents:(id)contents
                    ofType:(NSString *)typeName
                      error:(NSError **)outError
{
    NSFileWrapper *wrapper = (NSFileWrapper *)contents;
    NSDictionary *children = [wrapper fileWrappers];

    NSFileWrapper *entriesWrap =
        [children objectForKey:@"notes.dat"];
    NSData *data = [entriesWrap regularFileContents];
    NSKeyedUnarchiver *arch = [[NSKeyedUnarchiver alloc]
                                initForReadingWithData:data];
    _entries = [arch decodeObjectForKey:@"entries"];

    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"com.studiomagnolia.notesLoaded"
        object:self];

    return YES;
}
```

Whenever notes are (re)loaded you also post a notification to trigger the reload of data in the user interface.

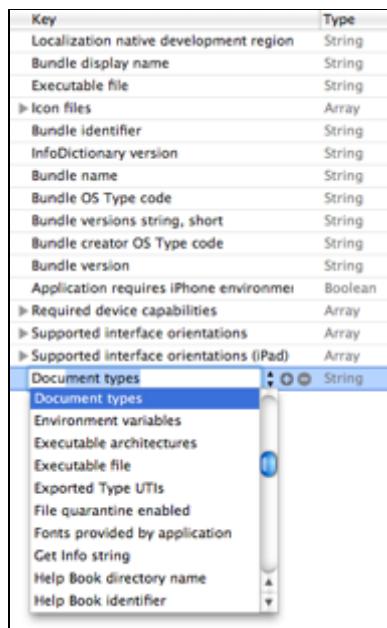
We are almost done with the iCloud part, but there is another crucial step to make: the registration of a custom document type.

Declaring a Custom Document

To build a custom document it is not enough to specify its structure and the way data is encoded and decoded. The iCloud requires the application to register its custom document type in the **Info.plist** file.

In this section you are going to create a new **UTI** (Uniform Type Identifier) for your app's custom data format. An UTI is a string that uniquely identifies a class of objects (in your case an array of notes), which you can refer to as a 'type', much like a jpg image.

To add a new UTI, select the **Supporting Files\dox-Info.plist** file in the project tree. Select one of the items shown and click **+**. This will display a menu from which you can add new metadata for the application. Select **Document types**, as shown below



If you click the down arrow to expand Document Types and the dictionary inside, you'll see it's created a dictionary with two elements inside: Document Type Name and Handler rank.

Document Type Name is the descriptive name for the document, so set it to **Dox Package**. Set the Handler rank to **Owner**.

Then select the dictionary and click the **+** button to add another entry. Set the key to **Document is a package or bundle**, and it will set itself up as a Boolean. Set the value to **YES**.

Add another entry to the dictionary, set the key to **Document Content Type UTIs**, and it will set itself up as an array. This is the list of unique IDs for the

document, so set the first entry to something unique like '**com.studiomagnolia.dox.package**'.

At this point your **dox-Info.plist** should look like the following:

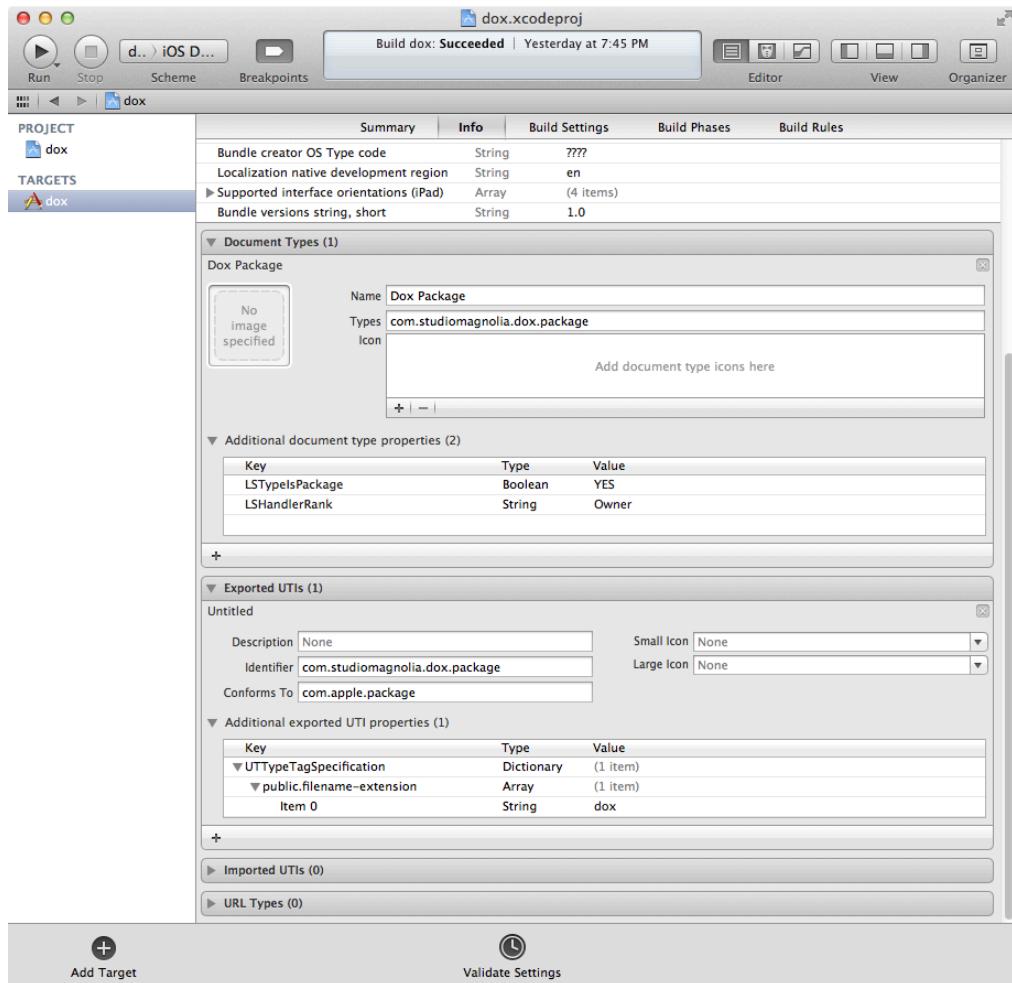
▼ Document types	Array	(1 item)
▼ Item 0 (Dox Package)	Dictionary	(4 items)
Document Type Name	String	Dox Package
Handler rank	String	Owner
▼ Document Content Type UTIs	Array	(1 item)
Item 0	String	com.studiomagnolia.dox.package
Document is a package or bur	Boolean	YES

Now you might think you are done, but you are not! It is not enough to declare a new UTI, you have also to export it. You have to add another property to the plist file, called 'Exported Type UTIs'. Essentially you have to recreate a structure as in the following picture:

▼ Exported Type UTIs	(1 item)
▼ Item 0 (com.studiomagnolia.dox.package)	(3 items)
▼ Conforms to UTIs	(1 item)
Item 0	com.apple.package
Identifier	com.studiomagnolia.dox.package
▼ Equivalent Types	(1 item)
▼ public.filename-extension	(1 item)
Item 0	dox

The identifier has to be the same provided above and 'dox' will be the extension of your file.

You should also check that all the information entered in the plist are replicated in the info section of your target application. Select the project in the tree, then the target, select the Info tab, and unfold both 'Document Types' and 'Exported UTIs'. The settings have to look as in the following screenshot:



Congrats! Now the iCloud setup is done and you are ready to connect the dots by hooking up the model with the user interface!

Showing Notes in the Table View

To display notes in the table view you will follow an approach similar to last chapter. Replace **SMMasterViewController.h** with the following:

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

@class SMDetailViewController;

@interface SMMasterViewController : UITableViewController

@property (strong, nonatomic) SMDetailViewController *detailViewController;
@property (strong, nonatomic) NSMetadataQuery *query;
```

```
@property (strong, nonatomic) SMNotesDocument *document;

- (void)loadDocument;

@end
```

The document property is a reference to the `SMNotesDocument` instance that manages the array of notes and iCloud functionality.

Next switch to **SMMasterViewController.m** and modify `viewDidLoad` to add a listener when notes are reloaded. You add this at the bottom of the method.

```
[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(notesLoaded:)
    name:@"com.studiomagnolia.notesLoaded"
    object:nil];
```

Next add the implementation of `insertNewObject:`, which is pretty trivial:

```
- (void)insertNewObject:(id)sender {
    SMNote *doc = [[SMNote alloc] init];
    doc.noteContent = @"Test";
    [self.document addNote:doc];
}
```

This simply creates a new `SMNote` with some default content and adds it to the `SMNotesDocument`.

Then add the selector triggered when notes are reloaded, which is simple as well:

```
- (void) notesLoaded:(NSNotification *) notification {
    self.document = notification.object;
    [self.tableView reloadData];
}
```

This notification passes the document as a parameter, so you store it in your instance variable and reload the table view to display the new data.

The `loadDocument` method is triggered when the application becomes active. Implement that as follows:

```
- (void)loadDocument {

    NSURL *ubiq = [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil];

    if (ubiq) {
```

```
NSMetadataQuery *query = [[NSMetadataQuery alloc] init];
_query = query;
[query setSearchScopes:[NSArray arrayWithObject:
NSMetadataQueryUbiquitousDocumentsScope]];
NSPredicate *pred = [NSPredicate predicateWithFormat:
@"%K == %@", NSMetadataItemFSNameKey, kFILENAME];
[query setPredicate:pred];

[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(queryDidFinishGathering:)
name:NSMetadataQueryDidFinishGatheringNotification
object:query];

[query startQuery];

} else {
    NSLog(@"No iCloud access");
}
}
```

You should be pretty familiar with this code. You check if iCloud is available and you run a query to look for a file in the cloud. Before you forget, declare `kFILENAME` at the top of the file as follows:

```
#define kFILENAME @"notes.dox"
```

Next add the callback for when the query has finished gathering data, which is pretty simple as well:

```
- (void)queryDidFinishGathering:(NSNotification *)notification {

    NSMetadataQuery *query = [notification object];
    [query disableUpdates];
    [query stopQuery];

    [self loadData:query];

    [[NSNotificationCenter defaultCenter]
removeObserver:self
name:NSMetadataQueryDidFinishGatheringNotification
object:query];

    _query = nil;
```

```
}
```

This simply stops the updates and calls `loadData:`.

The `loadData:` method behaves as in previous projects: if the document exists it gets opened, if it doesn't it is created.

```
- (void)loadData:(NSMetadataQuery *)query {

    if ([query resultCount] == 1) {
        NSMetadataItem *item = [query resultAtIndex:0];
        NSURL *url = [item
                      valueForAttribute:NSUTF8StringEncoding];
        SMNotesDocument *doc =
            [[SMNotesDocument alloc] initWithFileURL:url];
        self.document = doc;

        [doc openWithCompletionHandler:^(BOOL success) {
            if (success) {
                NSLog(@"doc opened from cloud");
                [self.tableView reloadData];
            } else {
                NSLog(@"failed to open");
            }
        }];
    } else { // No notes in iCloud
        NSURL *ubiqContainer = [[NSFileManager defaultManager]
                                 URLForUbiquityContainerIdentifier:nil];
        NSURL *ubiquitousPackage = [[ubiqContainer
                                     URLByAppendingPathComponent:@"Documents"
                                     URLByAppendingPathComponent:kFILENAME];
        SMNotesDocument *doc = [[SMNotesDocument alloc]
                               initWithFileURL:ubiquitousPackage];
        self.document = doc;

        [doc saveToURL:[doc fileURL]
        forSaveOperation:UIDocumentSaveForCreating
        completionHandler:^(BOOL success) {
            NSLog(@"new document saved to iCloud");
            [doc openWithCompletionHandler:^(BOOL success) {
                NSLog(@"new document was opened from iCloud");
            }];
        }];
    }
}
```

```
}
```

As the last step, select your **dox** Project entry, select the **dox** Target, and click the **Summary** tab. Scroll down to the **Entitlements** section, and click the **Enable iCloud** checkbox to enable iCloud (if it's not already). The default entries will be fine. You should be able to use the same provisioning profile you created in the last chapter, since your project name (hence bundle ID) should be the same.

Buid and run, and you should see a message in the console that says "new document saved to iCloud." The document has been correctly created, for this is the first time you run the application. The notes array is obviously empty.

Now you have to just to integrate the mechanism to render the view correctly according to the iCloud document. Add the following code to **SMMasterViewController.m**:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return self.document.entries.count;
}

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

    SMNote *n = [self.document entryAtIndex:indexPath.row];

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {

        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier];

        if ([[UIDevice currentDevice] userInterfaceIdiom] ==
            UIUserInterfaceIdiomPhone) {
```

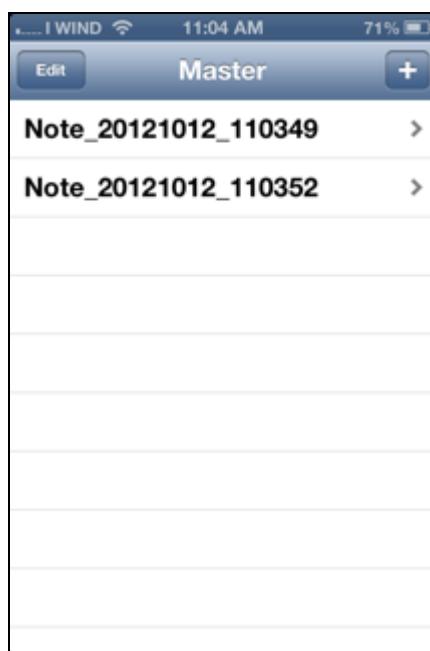
```
        cell.accessoryType =
            UITableViewCellStyleDisclosureIndicator;

    }

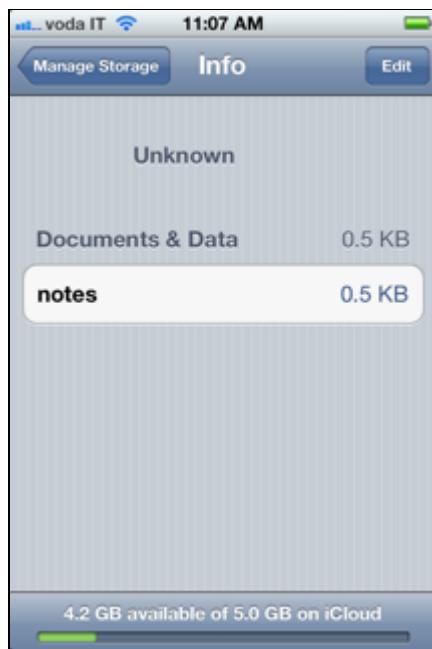
    cell.textLabel.text = n.noteId;
    return cell;

}
```

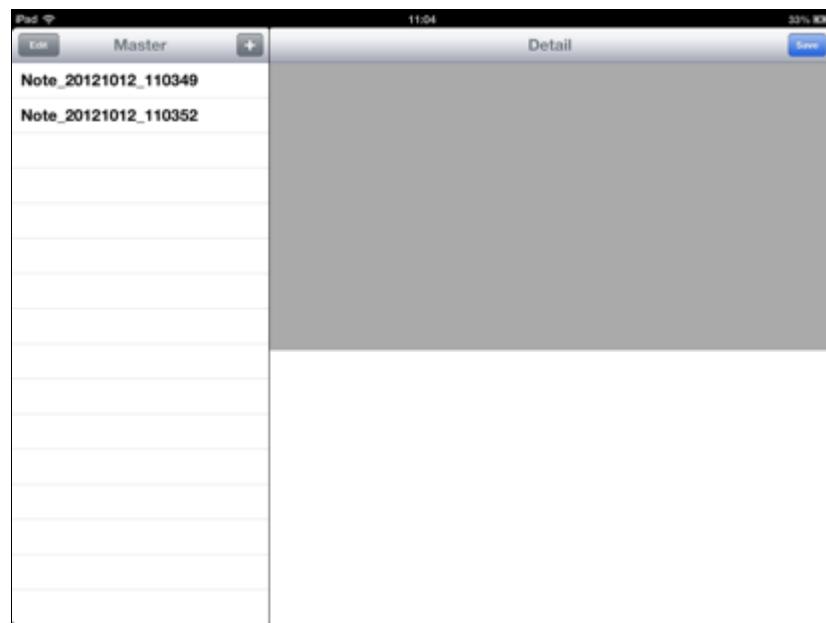
Run the application again and add a few notes:



As you can see in the figure the table is correctly populated. If you like you can also check out the iCloud panel on your device to verify the presence of a new package file. Although you have created multiple notes they are all wrapped in a single file.



You can also run the application on a second device to verify that notes are correctly loaded:



If you have any issues, you might want to change your `kFilename` constant to a different filename in case you have a corrupted file saved.

Now you are left with the last step: to show the content of a single note and save it when it is edited. Let's add the code to present the detail view when a note is tapped in the list. Open **SMMasterViewController.m** and add the following.

```

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad)
    {

        SMNote *selectedNote = [self.document
                               entryAtIndex:indexPath.row];
        self.detailViewController.detailItem = selectedNote;

    }
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue
          sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showDetail"]) {

        NSIndexPath *indexPath = [self.tableView
                                  indexPathForSelectedRow];
        SMNote *selectedNote = [self.document
                               entryAtIndex:indexPath.row];
        self.detailViewController.detailItem = selectedNote;
        [[segue destinationViewController]
         setDetailItem:selectedNote];

    }
}

```

Next you tweak a bit the detail view controller. Open **SMDetailViewController.h** and change as follows.

```

#import <UIKit/UIKit.h>
#import "SMNote.h"

@interface SMDetailViewController : UIViewController
<UISplitViewControllerDelegate, UITextViewDelegate>

@property (strong, nonatomic) SMNote *detailItem;
@property (weak, nonatomic) IBOutlet UITextView *noteTextView;
@property BOOL isChanged;

@end

```

Here you made the class implement the `UITextViewDelegate` and added a new property called `isChanged`.

Let's move onto the implementation file, **SMDetailViewController.m**. Add these two lines at the end of `viewDidLoad`:

```
self.noteTextView.delegate = self;
self.isChanged = NO;
```

This is just to initialize the property and set the class as delegate of the text field. The only method triggered by the text view that you are interested in is `textViewDidChange:`. Implement it as follows:

```
- (void)textViewDidChange:(UITextView *)textView
{
    self.isChanged = YES;
}
```

Then change the `configureView` method to display the note's content, as follows.

```
- (void)configureView
{
    if (self.detailItem) {
        self.noteTextView.text = self.detailItem.noteContent;
    }
}
```

And implement `saveEdits:` as follows:

```
- (void) saveEdits:(id)sender {

    if (self.isChanged) {
        self.detailItem.noteContent = self.noteTextView.text;
        [[NSNotificationCenter defaultCenter]
            postNotificationName:@"com.studiomagnolia.noteChanged"
            object:nil];
    }

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad) {

        self.detailItem = nil;
        self.noteTextView.text = @"";
    } else {

        [self.navigationController
```

```
    popViewControllerAnimated:YES];  
}  
}
```

Here when the user taps save button, you update the content of the note and post a "note changed" notification and pop or clean the detail view controller.

This notification will be caught by the `UIDocument` subclass which will trigger the save procedure.

Well done again! You have a whole new project where notes are not scattered in files but all wrapped in a single package, and have demonstrated how you can build up a single document consisting of multiple files.

Try it out on multiple devices to see how changes to note list and single notes are propagated correctly to the cloud.

Note: The project up to this point is in the resources for this chapter as **dox-04-final**.

That concludes this topic, so it's time for another break if you need one! Next up, you'll cover how you can easily store small amounts of key-value data in iCloud!

Storing Key-value Information in iCloud

If you have a very small amount of data you want to save in your app, like the current page number for an eBook reading app, or the last game level number you've unlocked, iCloud provides an extremely simple way to do so in the form of a key-value store.

You can think of the key-value store as an iCloud-enabled `NSUserDefaults`. It lets you easily store small pieces of data like `NSNumber`, `NSString`, `NSDate`, `NSData`, `NSArray`, or `NSDictionary`.

Before you think to refactor your application and store all the notes using this simple API, you should know that this part of iCloud is meant to share small amount of data, like preferences. The maximum amount of data an app can store in the key-value store was just 64KB in iOS 5 and is 1024KB for iOS 6.

One way that you could use the key-value in your app is to keep track of the current note you're editing. This would come in handy if you have a ton of notes in your application, and you're editing a note on the iPhone and you close the application. You might like to edit it on your iPad later, but you might not remember the note title or id.

If you kept track of the current note you're editing in the key-value store, the app could look this up when you open the app and start you off on the same note. This is exactly the feature that you are going to build next!

In this case you will store a simple and small piece of information: the id of the last edited note. In general the iCloud key-value store is perfect for this kind of usage.

The class that manages the iCloud key-value store is `NSUbiquitousKeyValueStore`. You can create a store, or a reference to the default one, set some objects and synchronize. Here's what the code might look like:

```
[ [NSUbiquitousKeyValueStore defaultStore]
    setString:@"YOUR_VALUE"
    forKey:@"YOUR_KEY"];
[ [NSUbiquitousKeyValueStore defaultStore] synchronize];
```

To retrieve a value you can use the message corresponding to the data type you stored, in your case `stringForKey`:

```
[ [NSUbiquitousKeyValueStore defaultStore]
    stringForKey:@"YOUR_KEY"];
```

As with `NSFileManager` the key-value store works asynchronously so you have to set up an observer to find out about changes. The notification to listen for has a pretty long name, `NSUbiquitousKeyValueStoreDidChangeExternallyNotification`.

```
[ [NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(YOURSELECTOR:)
    name: NSUbiquitousKeyValueStoreDidChangeExternallyNotification
    object:nil];
```

The pattern to work with key-value stores is pretty simple:

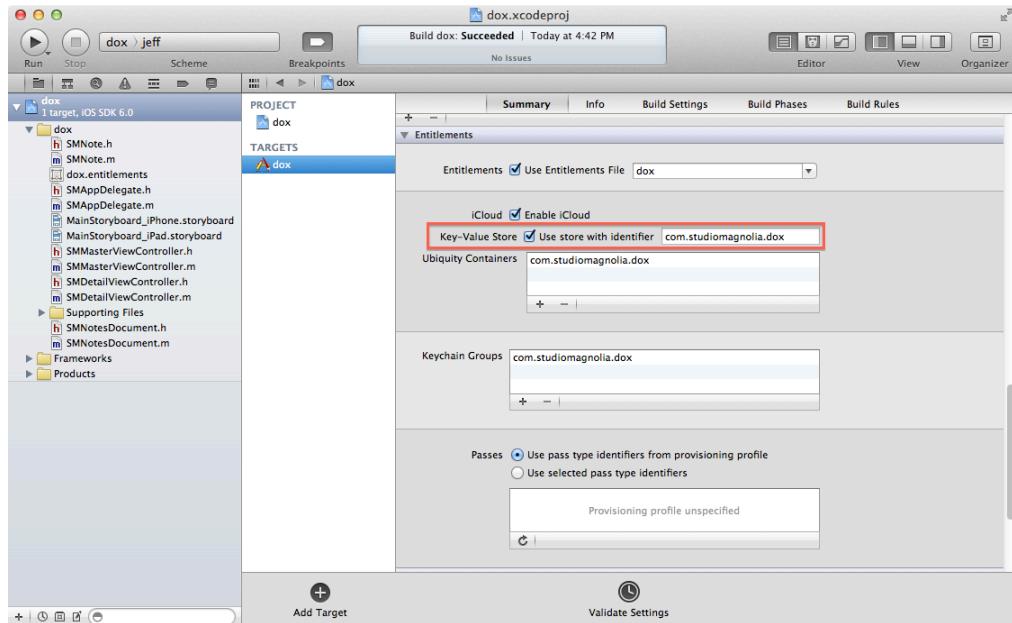
1. Register for updates from the iCloud key-value store
2. React accordingly in the callback

You will extend the project developed during the previous section. If you don't have it it's called **dox-04-final** in the resources for this chapter.

Let's update the app to work with the key-value store. Here are the changes you'll make:

1. Store a key-value entry when a single note is displayed and the user closes an application
2. Listen for key-value changes in `SMMasterViewController`
3. When notified, push a new view controller showing the note stored as the last edited

Before writing any code you should check that entitlements for iCloud key-value store are correctly set in the project as in the following figure (i.e. there is a iCloud key-value store entry):



Let's start by modifying **SMDetailViewController.m**. Add the following code to the bottom of `viewDidLoad`:

```
[ [NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(saveNoteAsCurrent)
    name:UIApplicationDidEnterBackgroundNotification
    object:nil];
```

Here you start listening to see if the application enters the background when the user is editing a note. Next implement the `saveNoteAsCurrent` callback:

```
- (void) saveNoteAsCurrent {
    [[NSUbiquitousKeyValueStore defaultStore]
        setObject:self.detailItem.noteId
        forKey:@"com.studiomagnolia.currentNote"];

    [[NSUbiquitousKeyValueStore defaultStore] synchronize];
}
```

This uses the iCloud key-value store to store the note id of the current note. It stores it under the key "com.studiomagnolia.currentNote", and you will use the very same key to look up the value later.

Now let's modify **SMMasterViewController.m**. Add the following at the bottom of `viewDidLoad`:

```
NSUbiquitousKeyValueStore* store =
    [NSUbiquitousKeyValueStore defaultStore];

[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateCurrentNodeIfNeeded:)
name: NSUbiquitousKeyValueStoreDidChangeExternallyNotification
object:store];

[store synchronize];
```

When you receive an update notification, you'll behave as if the user had tapped the cell corresponding to the note with that identifier. You will retrieve the note using the key and you push an instance of `SMDetailViewController`, as shown below:

```
- (void) updateCurrentNodeIfNeeded:(NSNotification *)
notification {
    NSLog(@"updateCurrent");

    NSString *currentNoteId =
    [[NSUbiquitousKeyValueStore defaultStore] stringForKey:
     @"com.studiomagnolia.currentNote"];
    SMNote *note = [self.document noteById:currentNoteId];
    if (note == nil) return;

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone) {

        int row = [self.document.entries indexOfObject:note];
        NSIndexPath * indexPath = [NSIndexPath
            indexPathForRow:row inSection:0];
        [self.tableView selectRowAtIndexPath:indexPath
            animated:YES
            scrollPosition:UITableViewScrollPositionBottom];
        [self performSegueWithIdentifier:@"showDetail"
            sender:nil];

    } else {
        self.detailViewController.detailItem = note;
    }
}
```

Also call this method inside `loadData`, right after calling `reloadData` on the table view:

```
[self updateCurrentNodeIfNeeded:nil];
```

Next you just need to add a little helper method, `noteById:`, to **SMNotesDocument.h**:

```
- (SMNote *) noteById:(NSString *)noteId;
```

And add the implementation in **SMNotesDocument.m**:

```
- (SMNote *) noteById:(NSString *)noteId {  
  
    SMNote *res = nil;  
  
    for (SMNote *n in _entries) {  
        if ([n.noteId isEqualToString:noteId]) {  
            res = n;  
        }  
    }  
  
    return res;  
}
```

Run this app on a device, open a note, and tap the home button to enter the background. Then run the app on a second device, and after the notes are loaded it will automatically open the note you were editing last.

Pretty cool eh? A nicer experience for the user, and you can see how easy `NSUbiquitousKeyValueStore` is to use!

Note: The project up to this point is in the resources for this chapter as **dox-05-kvs**.

How to Delete a File

You have seen how to add a local file to iCloud and how to remove a file from iCloud and just keep a local copy. But how can you just delete a file from iCloud (without necessarily having to keep a local copy)?

To delete a file completely, from the local directory and the iCloud, you can simply use the method `removeItemAtURL:error:`:

```
NSError *err;  
[[NSFileManager defaultManager] removeItemAtURL:  
    [self.document fileURL]
```

```
error:&err];
```

Note: There is no undo for this action so it's best to ask the user for confirmation beforehand.

Let's see how to integrate this in your application. You will start from a basic version of the project that you created earlier – the one that stores notes in separate files.

You can find this in the resources for this chapter as **dox-02**. Open it in Xcode, update your Bundle Identifier and Entitlement settings, and verify it works by installing on one device and creating a few notes. Then install on another device and check that the same notes appear as well.

To implement deletion from the table view you have to implement `tableView:commitEditingStyle:forRowAtIndexPath:`: Here you define what happens to the cell that the user chooses to delete. The selected note has to be deleted from three places: the table view, the array that populates the table and the file system.

Open **SMMasterViewController.m** and replace the implementation of the method as follows:

```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        SMNote *n = [self.notes objectAtIndex:indexPath.row];
        NSError *err;
        [[NSFileManager defaultManager] removeItemAtURL:
         [n fileURL]
         error:&err];
        [self.notes removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:
         [NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

Now you are ready to test the application. The ideal test is the following:

1. Run the app, add a few notes, and quit.

2. Reopen to check that notes are stored.
3. Delete a note, and quit again.
4. Reopen the app and verify that the deleted note is no longer there.

It all works fine, but what happens on other devices when you delete a note? Let's see in the following section.

Handling Deletion From Other Devices

Now try the following test script:

1. Add a few notes on one device.
2. Wait a little bit and start up the second device, verify the new notes appear.
3. Don't quit or put in background either app.
4. Delete a note on one device.

What happens on the other device? Nothing at all unfortunately!

The only way you can re-synchronize the devices is by putting the application in the background and making it active again to trigger the refresh of the list.

But shouldn't you handle that in real-time? Yes indeed - and that's the feature that you are going to add now!

One of the advantages of iCloud is that it can make applications aware of changes in remote resources on the fly. You just have to listen and react accordingly.

When there's a change in the cloud, a `UIDocumentStateChangedNotification` notification will be sent. That is the only notification available in the `UIDocument` class but it is enough to handle all the cases.

Besides notifications you can also keep track of the state of a document. Sometimes you might experience problems in reading or writing, e.g. the disk is full, the file has been edited, etc.

`UIDocument` has a key property, called `documentState`, which indicates how safe it is to allow a change in a document. This is a read-only property whose value is managed by the iCloud daemon. Possible values are the following:

- `UIDocumentStateNormal`: Everything is fine. Changes are persisted to disk and ready to be uploaded.
- `UIDocumentStateClosed`: When you have not yet opened the document.
- `UIDocumentStateInConflict`: There are conflicts (yes plural), in the document. For example when a document has been saved almost at the same time on two devices with different content.
- `UIDocumentStateSavingError`: There was an error in saving the document, e.g. no permission on the file, file already in use, etc.

- `UIDocumentStateEditingDisabled`: E.g. in the middle of a revert or whenever it is not safe to edit the document.

Now you are going to use the notification combined with these document states to handle the following use case: detecting when a note has been deleted and allowing the user to reinstate it.

Let's start by preparing `SMMasterViewController` to be ready to react to a deletion. The first step you want to achieve is to reload the list of notes in the table in case of change. Open up **SMMasterViewController.m** and add the following at the bottom of `viewDidLoad` to listen for a document change notification:

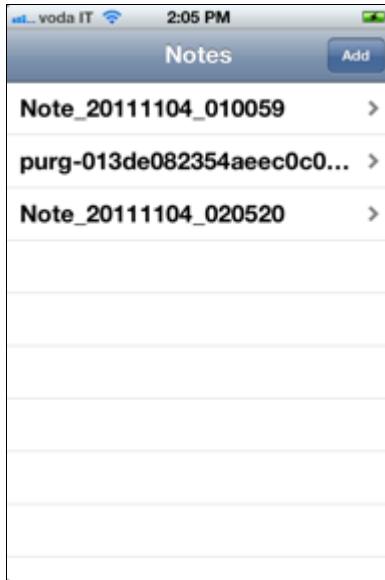
```
[ [NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(noteHasChanged:)
    name:UIDocumentStateChangedNotification
    object:nil];
```

It is important to notice that this notification is posted whenever there is a change, including when you open a file or when you add a new note.

So add this method to make the table view reload when the document state is `UIDocumentStateSavingError`. This is the state of a document when it has been deleted on the cloud but it is still present locally.

```
- (void)noteHasChanged:(id)sender {
    SMNote *d = [sender object];
    if ([d documentState] == UIDocumentStateSavingError) {
        [self.tableView reloadData];
    }
}
```

Now you can run the application on the two devices and verify the same list of documents appears. Now keep one device connected to the Mac and delete a document on the other. After a few seconds you should see the following:



The note has been deleted from the cloud and you have a sort of temporary version locally, without the previous id and file URL. If you tap the note you can see that also the content is still there. The console should also show an error message, like the following.

```
[Switching to process 8195 thread 0x2003]
[Switching to process 7171 thread 0x1c03]
2011-11-04 14:05:41.108 dox[2974:220f] Foundation called mkdir("/var/mobile/Library/Mobile Documents/.ubd/peer-98B230AE-135F-D571-D757-E25E258300F8-v23/ftr/(A Document Being Saved By dox)", it didn't return 0, and errno was set to 1.
```

This is due to the fact that the note is not a 'legal' document anymore. It is not on the cloud anymore and it is not even a 'classic' local file stored in the documents directory. It is just in a sort of limbo.

In such a situation, you can offer the user the opportunity to reinstate the note and its content. It will not be possible to reinstate a file with the same name as before, but you can create a new document with the content attached to the 'limbo note'.

To achieve this you have to refactor **SMDetailViewController.m** a bit. You have to change the label and action of the `UIBarButtonItem` on the top right according to the state of the document. Here is the new `viewDidLoad`.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [self configureView];
    self.noteTextView.backgroundColor =
        [UIColor lightGrayColor];

    UIBarButtonItem *saveButton = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemSave
        target:self
        action:@selector(saveEdits:)];
}
```

```
    self.navigationItem.rightBarButtonItem = saveButton;

    if (self.detailItem.documentState ==
        UIDocumentStateSavingError) {

        UIBarButtonItem *reinstateNoteButton =
        [[UIBarButtonItem alloc]
            initWithTitle:@"Reinstate"
            style:UIBarButtonItemStylePlain
            target:self
            action:@selector(reinstateNote)];

        self.navigationItem.rightBarButtonItem =
            reinstateNoteButton;
    }

}
```

If the document is in a saving error state the button on the right has the label "Reinstate" and the action `reinstateNote`. For the iPad to work correctly you also have to refactor the `configureView` method.

```
- (void)configureView
{
    if (self.detailItem) {
        self.noteTextView.text = self.detailItem.noteContent;

        if (self.detailItem.documentState ==
            UIDocumentStateSavingError) {

            UIBarButtonItem *reinstateNoteButton =
            [[UIBarButtonItem alloc]
                initWithTitle:@"Reinstate"
                style:UIBarButtonItemStylePlain
                target:self
                action:@selector(reinstateNote)];

            self.navigationItem.rightBarButtonItem =
                reinstateNoteButton;
        }
    }
}
```

If you run the application now and tap a 'limbo' note, the view should appear with the reinstate button as in the figure below:



Tapping the button will cause the app to crash since you haven't implemented the associated action yet, so let's do that next.

The method is pretty similar to the one used to create a new note. In fact, you have to instantiate a new note with a new URL and save it on disk. If the save operation is successful you can pop the view controller and post a notification to tell the table view that a note has been reinstated.

```
- (void) reinstateNote {  
  
    // Generate new filename for note based on date  
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];  
    [formatter setDateFormat:@"yyyyMMdd_hhmmss"];  
  
    NSString *fileName = [NSString stringWithFormat:@"Note_%@",  
                          [formatter stringFromDate:  
                           [NSDate date]]];  
  
    NSURL *ubiq = [[NSFileManager defaultManager]  
                    URLForUbiquityContainerIdentifier:nil];  
    NSURL *ubiquitousPackage =  
        [[ubiq URLByAppendingPathComponent:@"Documents"]  
         URLByAppendingPathComponent:fileName];  
  
    // Create new note and save it  
    SMNote *n = [[SMNote alloc]  
                  initWithFileURL:ubiquitousPackage];  
    n.noteContent = self.noteTextView.text;  
  
    [n saveToURL:[n fileURL]  
    forSaveOperation:UIDocumentSaveForCreating  
    completionHandler:^(BOOL success) {  
        if (success) {  
            [[NSNotificationCenter defaultCenter]  
             postNotificationName:  
             @"com.studiomagnolia.noteReinstated"]  
        }  
    }];  
}
```

```

        object:self];

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad) {
        self.detailItem = nil;
        self.noteTextView.text = @"";
    } else {
        [self.navigationController
            popViewControllerAnimated:YES];
    }
} else {
    NSLog(@"error in saving reinstated note");
}
];
}
}

```

We have now to catch the notification in **SMMasterViewController.m**. Add the following to the bottom of viewDidLoad:

```

[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(loadDocument)
    name:@"com.studiomagnolia.noteReinstated"
    object:nil];

```

This way, whenever a note is reinstated, the application loads a fresher list of notes from iCloud.

Since you have two view controllers, you have to manage the "note has been deleted" scenario also in **SMDetailViewController**. Add the following to the bottom of viewDidLoad in **SMDetailViewController.m**:

```

[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(noteHasChanged:)
    name:UIDocumentStateChangedNotification
    object:nil];

```

The selector is pretty similar to the previous one: you change the title and you show the "Reinstate" button.

```

- (void)noteHasChanged:(id)sender {

    if (self.detailItem.documentState ==
        UIDocumentStateSavingError) {
        self.title = @"Limbo note";
        UIBarButtonItem *reinstateNoteButton =

```

```
[ [UIBarButtonItem alloc]
    initWithTitle:@"Reinstate"
        style:UIBarButtonItemStylePlain
        target:self
        action:@selector(reinstateNote)];

self.navigationItem.rightBarButtonItem =
    reinstateNoteButton;

}

}
```

Build and run to test it. Now the 'Reinstate' button appears also when a note is opened and gets deleted on another device. Use the button to bring a deleted note back up to life!

Note: The project up to this point is in the resources for this chapter as **dox-06**.

Now that you know how to create and delete notes, let's see how to handle changes.

Handling Changes

So far you took care of situations where files are created or deleted. But you also need to handle the case where the content of a file changes. In particular, what happens if you are editing a note on two devices at about the same time?

In your current implementation, nothing happens because although you observe for document changes, you filter for only the `UIDocumentStateSavingError` case.

In the scenario of concurrent editing there is no save error, because the daemon on different devices pushes changes to the cloud at discrete intervals after the `updateChangeCount:` method is called. If you are editing a note and the daemon pulls new changes from the cloud, although there is no conflict at document level, there is a conflict at the user interface level, because you are editing a note whose content has changed. At this point you need a policy to handle the situation.

There is no golden policy perfect for every situation, but there is a good policy according to the aim of your application and the goal that the user is trying to achieve. Some examples of policies in conflict situations are:

- The most recent version wins;
- The current version on the device in usage wins

- The user is asked to pick a version
- The user is asked to manually merge changes

Before implementing a policy let's see where the best place is to apply it. In your current implementation (project **dox-06**) you are already listening for changes to the `documentState` property. Add the following code to the bottom of the `noteHasChanged:` method in **SMDetailViewController.m** as follows:

```
if (self.detailItem.documentState ==  
    UIDocumentStateEditingDisabled) {  
  
    self.navigationItem.leftBarButtonItem.enabled = NO;  
    NSLog(@"document state is  
          UIDocumentStateEditingDisabled");  
  
}  
  
if (self.detailItem.documentState == UIDocumentStateNormal) {  
  
    self.navigationItem.leftBarButtonItem.enabled = YES;  
    NSLog(@"old content is %@", self.noteTextView.text);  
    NSLog(@"new content is %@", self.detailItem.noteContent);  
  
}
```

Now run the app on two devices, and make sure the second is connected to Xcode so you can see the output on the console. Then run the following test:

1. On one device, connected to the debugger, tap a note as if you are going to edit it.
2. On the other device, open the same note, change it and close the single note view.

After a few minutes, the console should show something like this:

```
2011-11-04 15:55:28.964 dox[1351:707] document state is UIDocumentStateEditingDisabled  
2011-11-04 15:55:29.508 dox[1351:707] old content is This is a note  
2011-11-04 15:55:29.508 dox[1351:707] new content is This is an edited note
```

As you can see you receive two notifications. In the first case the state of the document is `UIDocumentStateEditingDisabled`. As explained above, while the document is in this state, it is not suggested to edit and save it. That is why you have temporarily disabled the 'back' button to prevent the user from closing the view, and thus saving the file. The document has been put in this state because new content is being received from the other device.

Right after that (depends on the length of the content to be saved), you receive a new notification, where the state of the document is back to normal.

We have also traced the new content of the current note, which is different with respect to the string populating your text view. This is the key point to implement your policy. Update the code in `noteHasChanged:` so that the latest version wins, by updating the `UIDocumentStateNormal` case to the following:

```
if (self.detailItem.documentState == UIDocumentStateNormal) {  
  
    self.navigationItem.leftBarButtonItem.enabled = YES;  
    NSLog(@"old content is %@", self.noteTextView.text);  
    NSLog(@"new content is %@", self.detailItem.noteContent);  
    self.noteTextView.text = self.detailItem.noteContent;  
  
}
```

Build and run to test it out - now when you edit a note on one device and have it open on a second, the second will automatically update to the latest content!

Note that if you want the old version to win you can just ignore the notification and don't update the view.

If you want the user to pick a version then you need a new view controller which shows both versions so the user can choose which one to continue with. Let's try this out to see how it works.

We will call this new class `SMVersionPicker`. It will have a view controller that displays two versions of a note, with two buttons to let the user pick, as in the following screenshot.



This view will be pushed when the user taps a 'Resolve' button, which will appear in case of conflict between local and iCloud content.

Update the `UIDocumentStateNormal` case in `noteHasChanged:` to change the top right button when needed. Here is the new version of the method in **SMDetailViewController.m**.

```
- (void)noteHasChanged:(id)sender {

    if (!self.detailItem)
        return;

    if (self.detailItem.documentState ==
            UIDocumentStateSavingError) {
        self.title = @"Limbo note";
        UIBarButtonItem *reinstateNoteButton =
            [[UIBarButtonItem alloc]
                initWithTitle:@"Reinstate"
                style:UIBarButtonItemStylePlain
                target:self
                action:@selector(reinstateNote)];

        self.navigationItem.rightBarButtonItem =
            reinstateNoteButton;
    }

    if (self.detailItem.documentState ==
            UIDocumentStateEditingDisabled) {

        self.navigationItem.leftBarButtonItem.enabled = NO;
        NSLog(@"document state is
            UIDocumentStateEditingDisabled");

    }

    if (self.detailItem.documentState == UIDocumentStateNormal)
    {

        self.navigationItem.leftBarButtonItem.enabled = YES;
        NSLog(@"old content is %@", self.noteTextView.text);
        NSLog(@"new content is %@",
            self.detailItem.noteContent);

        if (![self.noteTextView.text
            isEqualToString:self.detailItem.noteContent]) {

            UIBarButtonItem * resolveButton =

```

```

        [[UIBarButtonItem alloc]
            initWithTitle:@"Resolve"
                style:UIBarButtonItemStylePlain
                target:self
                action:@selector(resolveNote)];

        self.navigationItem.rightBarButtonItem =
            resolveButton;

    }

}

```

In the project **dox-07** in the resources for this chapter, I have made a view controller for you called `SMVersionPicker` that is ready to use, because creating the user interface itself is out of the scope of this chapter. Drag **SMVersionPicker.h**, **SMVersionPicker.m**, **SMVersionPicker_iPad.xib**, and **SMVersion_iPhone.xib** into your project.

It has xib files for both iPhone and iPad already configured, including two text views and two buttons to allow the user picking the version. To give you an idea of how it works, here is its header file:

```

#import <UIKit/UIKit.h>
#import "SMNote.h"

@interface SMVersionPicker : UIViewController

@property (strong, nonatomic) IBOutlet UITextView
    *oldContentTextView;
@property (strong, nonatomic) IBOutlet UITextView
    *newerContentTextView;
@property (strong, nonatomic) NSString *oldNoteContentVersion;
@property (strong, nonatomic) NSString *newerNoteContentVersion;
@property (strong, nonatomic) SMNote *currentNote;

- (IBAction)pickNewerVersion:(id)sender;
- (IBAction)pickOldVersion:(id)sender;

@end

```

And here are the relevant parts of its implementation:

```

- (void) viewWillAppear:(BOOL)animated {

```

```
[super viewWillAppear:animated];

self.oldContentTextView.text = self.oldNoteContentVersion;
self.newerContentTextView.text =
    self.newerNoteContentVersion;

}

- (IBAction)pickNewerVersion:(id)sender {

    self.currentNote.noteContent =
        self.newerContentTextView.text;

    [self.currentNote
        saveToURL:[self.currentNote fileURL]
    forSaveOperation:UIDocumentSaveForOverwriting
    completionHandler:^(BOOL success) {

        if (success) {

            [self.navigationController
                popViewControllerAnimated:YES];
        }
    }];
}

- (IBAction)pickOldVersion:(id)sender {

    self.currentNote.noteContent = self.oldContentTextView.text;

    [self.currentNote
        saveToURL:[self.currentNote fileURL]
    forSaveOperation:UIDocumentSaveForOverwriting
    completionHandler:^(BOOL success) {

        if (success) {

            [self.navigationController
                popViewControllerAnimated:YES];
        }
    }];
}
```

```
    }];
}
```

When the controller appears it sets both old and new contents in the text views. Once a button is tapped it updates the content of the note with the value of the corresponding text are and saves the note.

After adding the files into your project, all you have to do is add the code to display it in **SMDetailViewController.m**. First import the header at the top of the file:

```
#import "SMVersionPicker.h"
```

Then add the code for the `resolveNote` button tap callback to display the new view controller:

```
- (void) resolveNote {
    SMVersionPicker *picker = nil;
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        picker = [[SMVersionPicker alloc]
                   initWithNibName:@"SMVersionPicker_iPad"
                   bundle:nil];
    } else {
        picker = [[SMVersionPicker alloc]
                   initWithNibName:@"SMVersionPicker_iPhone"
                   bundle:nil];
    }

    picker.newerNoteContentVersion =
        self.detailItem.noteContent;
    picker.oldNoteContentVersion = self.noteTextView.text;
    picker.currentNote = self.detailItem;

    [self.navigationController pushViewController:picker
                                         animated:YES];
}
```

The final touch is to refresh the detail view when a version has been picked. To do this, implement `viewWillAppear:` as follows:

```
- (void) viewWillAppear:(BOOL)animated {

    self.noteTextView.text = self.detailItem.noteContent;

    if (self.detailItem.documentState == UIDocumentStateNormal)
    {
```

```

UIBarButtonItem *saveButton =
[[UIBarButtonItem alloc]
 initWithBarButtonSystemItem:UIBarButtonSystemItemSave
 target:self
 action:@selector(saveEdits:)];

self.navigationItem.rightBarButtonItem = saveButton;

}

}

```

Here you are, finally! Build and run, edit a document while you're viewing it on another, and you'll see your application is now able to detect changes to documents and to show the "Resolve" button when there are two versions of a note. By tapping the button the user is shown a view that helps choosing the version to keep on iCloud.

Note: The project up to this point is in the resources for this chapter as **dox-07**.

Handling Conflicts

Earlier in this chapter you saw the list of available states, and one of them was `UIDocumentStateInConflict`. This state is assigned to documents that are being updated almost at the same time from different devices.

At this point it is difficult to define the 'almost'. In many cases this scenario is usually reproducible by triggering the save actions on two different devices, both connected to the web, with a lag of 1-2 seconds.

Let's see how to deal with such a situation. First let's update your **dox-07** project so you can get an indication of when you have a conflict.

Inside **SMMasterViewController.m**, refactor `noteHasChanged:` as follows:

```

- (void)noteHasChanged:(id)sender {
    [self.tableView reloadData];
}

```

Then refactor `tableView:cellForRowAtIndexPath:` to color a cell red when it is in conflict, like this:

```

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

```

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

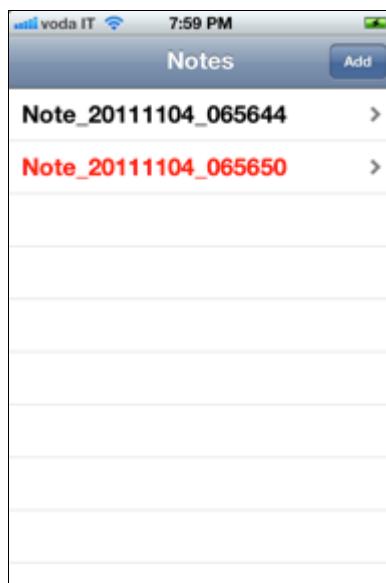
    SMNote * note = self.notes[indexPath.row];
    if ([note documentState] == UIDocumentStateInConflict) {
        cell.textLabel.textColor = [UIColor redColor];
    } else {
        cell.textLabel.textColor = [UIColor blackColor];
    }

    cell.textLabel.text = note.fileURL.lastPathComponent;
    return cell;
}
```

Then try out this scenario by running the following test:

1. Run the app on two devices.
2. Open the same note on both devices.
3. Edit the note so that the content is different on each device.
4. Tap the save button on both applications almost at the same time.

If you wait for a while (usually 10-15 seconds depending on the size of the file saved and the speed of the connection) both devices should receive a notification and the table should show a red note as in the following screenshot:



Unlike previously explored states, this kind of conflict is 'permanent'. That is, if you restart the application you will continue to see a red label. In previous examples, a

reload of the application would make the last version win over the rest, but in this case the conflict is detected by iCloud itself and requires user intervention to be resolved.

To resolve an iCloud-detected conflict, there is a handy set of APIs you can use. `NSFileVersion` is a very helpful class that represents 'snapshots' of a file at a given point. You can retrieve an array of `NSFileVersion` snapshots of the document that conflict with each other using the `unresolvedConflictVersionsOfItemAtURL:` method. You can access each documents data, modification time, and URL, and use this information resolve the conflicts automatically, or allow the user to decide.

To experiment with what we've got access to, refactor a bit the `tableView:cellForRowAtIndexPath:` method to display some properties of a version:

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

    SMNote * note = self.notes[indexPath.row];

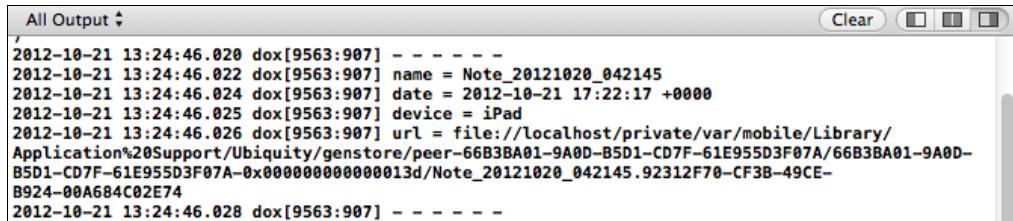
    if ([note documentState] == UIDocumentStateInConflict) {

        cell.textLabel.textColor = [UIColor redColor];

        NSArray *conflicts = [NSFileVersion
            unresolvedConflictVersionsOfItemAtURL:note.fileURL];
        for (NSFileVersion *version in conflicts) {
            NSLog(@"-----");
            NSLog(@"name = %@", version.localizedDescription);
            NSLog(@"date = %@", version.modificationDate);
            NSLog(@"device = %@", version.localizedDescriptionOfSavingComputer);
            NSLog(@"url = %@", version.URL);
            NSLog(@"-----");
        }
    } else {
        cell.textLabel.textColor = [UIColor blackColor];
    }

    cell.textLabel.text = note.fileURL.lastPathComponent;
    return cell;
}
```

Run your project again, and you should see something like the following in the console:



```
All Output ▾
2012-10-21 13:24:46.020 dox[9563:907] -----
2012-10-21 13:24:46.022 dox[9563:907] name = Note_20121020_042145
2012-10-21 13:24:46.024 dox[9563:907] date = 2012-10-21 17:22:17 +0000
2012-10-21 13:24:46.025 dox[9563:907] device = iPad
2012-10-21 13:24:46.026 dox[9563:907] url = file:///localhost/private/var/mobile/Library/Application%20Support/Ubiquity/genstore/peer-66B3BA01-9A0D-B5D1-CD7F-61E955D3F07A/66B3BA01-9A0D-B5D1-CD7F-61E955D3F07A-0x000000000000013d/Note_20121020_042145.92312F70-CF3B-49CE-B924-00A684C02E74
2012-10-21 13:24:46.028 dox[9563:907] -----
```

Notice that the URL of the document in conflict is a very long and unique address. This helps avoiding filename conflicts, much like in git or other source code management systems, where each commit has a project unique reference id.

One possible solution to resolve this conflict is to show the version picker when the user taps a red note. In this case you assume there will be just two competing versions of a document but you can easily adapt it to allow more complex cases.

To handle this case I have prepared a refactored version of the `SMVersionPicker`, which is included in the project named **dox-08** in the resources for this chapter. Remove the old version of `SMVersionPicker` from your project and drag the new files in.

Also, open **`SMDetailViewController.m`** and comment out `resolveNote:`, as you will not need that for this section.

This version of the class has slightly different names to match this new scenario, plus a new method called `cleanConflicts`. Here is the header:

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

@interface SMVersionPicker : UIViewController

@property (strong, nonatomic) IBOutlet UITextView *thisDeviceContentTextView;
@property (strong, nonatomic) IBOutlet UITextView *otherDeviceContentTextView;
@property (strong, nonatomic) NSString *thisDeviceContentVersion;
@property (strong, nonatomic) NSString *otherDeviceContentVersion;
@property (strong, nonatomic) SMNote *currentNote;

- (IBAction)pickOtherDeviceVersion:(id)sender;
- (IBAction)pickThisDeviceVersion:(id)sender;

- (void) cleanConflicts;
```

```
@end
```

The methods to pick the version have been refactored as follows.

```
- (IBAction)pickOtherDeviceVersion:(id)sender {

    self.currentNote.noteContent =
        self.otherDeviceContentTextView.text;
    [self cleanConflicts];

    [self.currentNote saveToURL:[self.currentNote fileURL]
        forSaveOperation:UIDocumentSaveForOverwriting
        completionHandler:^(BOOL success) {

        if (success) {

            [self.navigationController
                popViewControllerAnimated:YES];

            [[[NSNotificationCenter defaultCenter]
                postNotificationName:@"com.studiomagnolia.conflictResolved"
                object:self];
        }

    }];
}

- (IBAction)pickThisDeviceVersion:(id)sender {

    self.currentNote.noteContent =
        self.thisDeviceContentTextView.text;
    [self cleanConflicts];

    [self.currentNote saveToURL:[self.currentNote fileURL]
        forSaveOperation:UIDocumentSaveForOverwriting
        completionHandler:^(BOOL success) {

        if (success) {

            [self.navigationController
                popViewControllerAnimated:YES];

            [[[NSNotificationCenter defaultCenter]
```

```

    postNotificationName:@"com.studiomagnolia.conflictResolved"
        object:self];
}

};

}

```

Besides storing the selected value you also call the `cleanConflicts` method, you save the URL as usual, and you post a notification before popping the view controller. The `cleanConflicts` method is defined like this:

```

- (void) cleanConflicts {
    NSArray *conflicts =[NSFileVersion
        unresolvedConflictVersionsOfItemAtURL:
            [self.currentNote fileURL]];

    for (NSFileVersion *c in conflicts) {
        c.resolved = YES;
    }

    NSError *error = nil;
    BOOL ok = [NSFileVersion
        removeOtherVersionsOfItemAtURL:
            [self.currentNote fileURL]
            error:&error];

    if (!ok) {
        NSLog(@"Can't remove other versions: %@", error);
    }
}

```

As stated earlier a conflict is 'permanent' until it is marked as resolved. To resolve a conflict you have to perform two steps:

1. Mark each unresolved conflict as resolved
2. Remove the other versions of the file.

The code above performs both of these tasks. Once you have met these conditions the conflict is fully resolved and a reload of the note list will show the normal black labeled titles!

In the next step you have to present the version picker if the document is in conflict. To do so, import the file at the top of **SMMasterViewController.m**:

```
#import "SMVersionPicker.h"
```

Then modify `tableView:didSelectRowAtIndexPath:` as follows:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    SMNote *selectedNote = self.notes[indexPath.row];
    if (selectedNote.documentState == UIDocumentStateInConflict)
    {
        SMVersionPicker *picker = nil;

        if (UI_USER_INTERFACE_IDIOM() ==
            UIUserInterfaceIdiomPad) {
            picker = [[SMVersionPicker alloc]
                      initWithNibName:@"SMVersionPicker_iPad"
                      bundle:nil];
        } else {
            picker = [[SMVersionPicker alloc]
                      initWithNibName:@"SMVersionPicker_iPhone"
                      bundle:nil];
        }

        picker.currentNote = selectedNote;

        NSArray *conflicts =
        [NSFileVersion unresolvedConflictVersionsOfItemAtURL:
         selectedNote.fileURL];

        for (NSFileVersion *version in conflicts) {

            SMNote *otherDeviceNote = [[SMNote alloc]
                                       initWithFileURL:version.URL];

            [otherDeviceNote openWithCompletionHandler:^(BOOL
success) {

                if (success) {
                    picker.thisDeviceContentVersion =
                        selectedNote.noteContent;
                    picker.otherDeviceContentVersion =
                        otherDeviceNote.noteContent;
                    [self.navigationController
                     pushViewController:picker
                     animated:YES];
                }
            }];
        }
    }
}
```

```
        }

    if (selectedNote.documentState == UIDocumentStateNormal) {
        if ([[UIDevice currentDevice] userInterfaceIdiom] ==
            UIUserInterfaceIdiomPad)
    {
        self.detailViewController.detailItem = selectedNote;
    } else {
        SMDetailViewController *detailViewController =
        [self.storyboard
            instantiateViewControllerWithIdentifier:
            @"SMDetailViewControllerID"];
        detailViewController.detailItem = selectedNote;
        [self.navigationController
            pushViewController:detailViewController
            animated:YES];
    }
}
}
```

Here you have hijacked a bit the flow of the storyboard. The easiest way to fix this is to import the new storyboards from the **dox-08** project (or you could tweak yours to 'unlink' the master and detail view controller and to assign to the detail view the storyboard id "SMDetailViewControllerID").

The last step is to listen for the "com.studiomagnolia.conflictResolved" notification in SMMasterViewController to correctly update the list of notes and revert to black those previously red. Add the following to end of viewDidLoad:

```
[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(conflictResolved)
    name:@"com.studiomagnolia.conflictResolved"
    object:nil];
```

Then implement the associated selector, which simply reloads the table view data:

```
- (void) conflictResolved {
    [self.tableView reloadData];
}
```

Let's try it out! Run the application on two devices (d1 and d2) and perform the following steps:

1. Create a conflicted file if you don't have one already (following the instructions earlier)
2. On d1 tap the red note and pick one version
3. Notice on d1 the reloaded list of notes with no conflict
4. Wait to see the update also on d2

Your knowledge of iCloud is growing more and more. you are now able to create, edit, delete documents and even resolve version conflicts!

Note: The project up to this point is in the resources for this chapter as **dox-08**.

There's still a lot left to explore with iCloud - still to cover is working with undo and redo, exporting data for download, and working with Core Data.

Using the Undo Manager

In all the examples so far, to save a document you have used either `saveToURL:forSaveOperation:completionHandler:` or a simple `updateChangeCount:`. Both of these methods are basically telling iCloud "I am done editing this file, so you can send updates to the remote location."

In this section, you'll learn about a third way to trigger this mechanism: using the undo manager.

`NSUndoManager` is a class available since iOS 3 (!) and it helps you keep track of undo (and redo) changes to properties. It is quite simple to use, and allows you to register a set of operations related to an object together with the method to be invoked if the user chooses to undo.

So if you want to use the undo manager, whenever you modify an object, you should call this method to tell the undo manager how to undo the modification:

```
- (void)registerUndoWithTarget:(id)target  
                      selector:(SEL)selector  
                        object:(id)anObject;
```

Let's go over these parameters one by one:

1. **target**: The object to which the selector belongs.
2. **selector**: The method called to revert the modification.
3. **object**: Here you can pass an object to be passed to the selector. Often you'll pass the "old state" here so the undo method can switch back to the old state.

The manager keeps also track whenever you move back in the history of states. For example if you undo an action the manager does not delete current state, but keeps it so you can reinstate it by means of a redo.

It is important to note that all the actions collected in the undo stack are related to the main run loop, so if you quit and restart the application you lose memory of the stack.

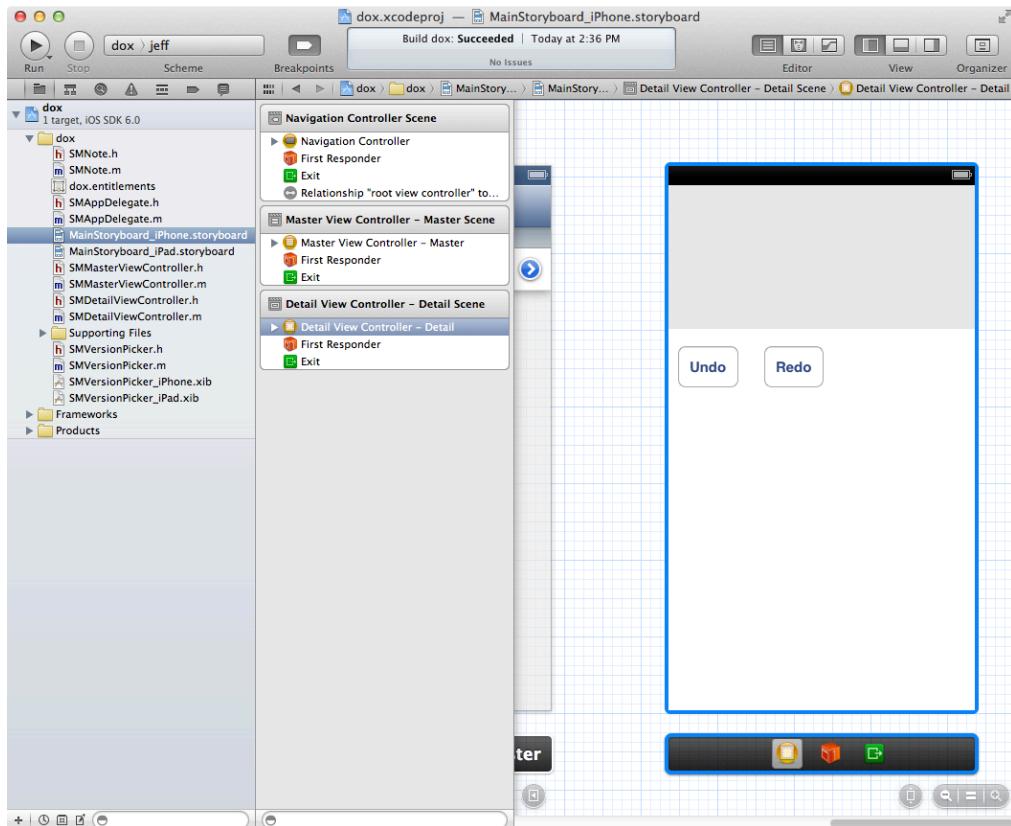
Whenever you use an undo manager, there is no need to call `updateChangeCount:`, because is automatically called for you when you send an undo or redo message. Moreover, Apple engineers have been kind enough to integrate an undo manager in `UIDocument`, so each instance of this class (and subclasses) has a property called `undoManager` which allows developers to build their stack of changes for each iCloud-enabled file.

Let's see how you can integrate an undo manager in your application. You can start from project **dox-08**, which you have finished in the previous section. It is included in the resources for the chapter.

Open up **SMDetailViewController.h** and add two new properties and methods, as shown below:

```
@property (strong, nonatomic) IBOutlet UIButton *undoButton;
@property (strong, nonatomic) IBOutlet UIButton *redoButton;
```

Then open the storyboard and add two `UIButtons` for undo and redo:



Note you should put the buttons high enough so that the keyboard doesn't cover them up when it appears. Remember to connect the buttons to their outlets. Finally, repeat this entire process for the iPad storyboard.

Now that your user interface is hooked up with the view controller you need to refactor its logic. Open **SMDetailViewController.m** and add the following at the bottom of `viewDidLoad`:

```
[self.undoButton addTarget:self
                     action:@selector(performUndo:)
           forControlEvents:UIControlEventTouchUpInside];

[self.redoButton addTarget:self
                     action:@selector(performRedo:)
           forControlEvents:UIControlEventTouchUpInside];
```

Then implement the methods that get called when these buttons are tapped as follows:

```
- (void)performUndo:(id)sender {
    if ([self.detailItem.undoManager canUndo]) {
        [self.detailItem.undoManager undo];
    }
}
```

```
- (void)performRedo:(id)sender {
    if ([self.detailItem.undoManager canredo]) {
        [self.detailItem.undoManager redo];
    }
}
```

Then change the `saveEdits:` method associated to the 'save' button like this:

```
- (void) saveEdits:(id)sender {
    [self saveNoteWithContent:self.noteTextView.text];
}
```

Finally implement the method `saveNoteWithContent:` as follows:

```
- (void) saveNoteWithContent:(NSString *)newContent {

    NSString *currentText = self.detailItem.noteContent;

    if (newContent != currentText) {

        [self.detailItem.undoManager
            registerUndoWithTarget:self

                selector:@selector(saveNoteWithContent:)
                object:currentText];

        self.detailItem.noteContent = newContent;
        self.noteTextView.text = self.detailItem.noteContent;

    }

    self.undoButton.enabled =
        [self.detailItem.undoManager canUndo];
    self.redoButton.enabled =
        [self.detailItem.undoManager canredo];

}
```

This method updates the content of the note and the text view, but before that it registers itself with the undo manager built in the `SMNote` class (inherited from `UIDocument`). After that it updates the states of buttons according to the stack of changes.

This way, whenever and undo or redo is triggered that action will be registered as a change and the user will be free to move back and forth new or old versions of the note.

Compile and run on two different devices, and run through the following test case:

- Open a note on device 1, edit the text and save.
- Then you can tap the undo and redo buttons to see the text changing.

This is a new tool that you can exploit to build user-friendly applications, which enable the user to review changes to the content of notes. And as you can see – it's pretty easy to add, eh?

Exporting Data From iCloud

You can export a URL for your data you save in iCloud, so that users or non-iCloud applications can access the data just by downloading it at the given URL.

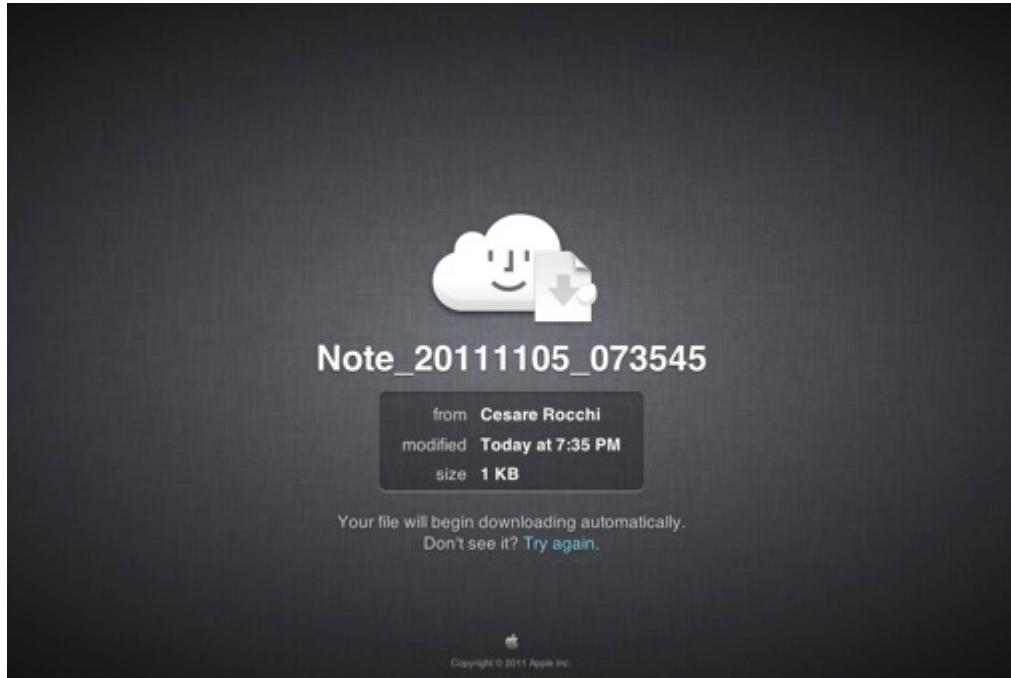
To do this, you just call the following method on `NSFileManager`:

```
- (NSURL *)  
URLForPublishingUbiquitousItemAtURL:(NSURL *)url  
expirationDate:(NSDate **)outDate  
error:(NSError **)error;
```

This method returns a URL like the following:

```
https://www.icloud.com/documents/dl/?p=2&t=BAJJZKAKzzXMg7o4tC8BODE  
IOzkOd6UN2q0A
```

This is a temporary URL and is not indexed by search engines. Whoever has the URL can download the file unless it is expired. This is what it looks like when you open the URL:



Let's see how you can integrate this functionality in your application.

In this section you will add an export button when a note is opened. Tapping the button will send an email containing the URL of the exported note.

In this section, you will start from the project **dox-07** completed previously. Grab it from the resources for this chapter, update the bundle identifier and entitlements, and run it to check that everything is ok.

Then open **SMDetailViewController.h** and modify the file to the following:

```
#import <UIKit/UIKit.h>
#import "SMNote.h"
#import <MessageUI/MessageUI.h>

@interface SMDetailViewController : UIViewController
<UISplitViewControllerDelegate,
MFMailComposeViewControllerDelegate>

@property (strong, nonatomic) SMNote *detailItem;
@property (weak, nonatomic) IBOutlet UITextView *noteTextView;

- (NSURL *) generateExportURL;

@end
```

This imports the `MessageUI` framework, implements the compose delegate, and adds the method `generateExportURL`.

Since you're using the `MessageUI` framework you have to import it into your project as well. So select the project, then the target and finally the **Build Phases** tab. Expand the list of linked libraries, click on the **+** sign, select **MessageUI.framework**, and click **Add**.

Next, open up **SMDetailViewController.m** and change `viewWillAppear:` like this:

```
- (void) viewWillAppear:(BOOL)animated {

    self.noteTextView.text = self.detailItem.noteContent;

    UIBarButtonItem *exportButtonItem =
        [[UIBarButtonItem alloc]
            initWithTitle:@"Export"
            style:UIBarButtonItemStylePlain
            target:self
            action:@selector(sendNoteURL)];

    self.navigationItem.rightBarButtonItem = exportButtonItem;

}
```

Then implement the callback as follows:

```
- (void) sendNoteURL {

    NSURL *url = [self generateExportURL];

    MFMailComposeViewController *mailComposer;
    mailComposer = [[MFMailComposeViewController alloc] init];
    mailComposer.mailComposeDelegate = self;
    [mailComposer
        setModalPresentationStyle:UIModalPresentationFormSheet];
    [mailComposer setSubject:@"Download my note"];
    [mailComposer setMessageBody:[NSString stringWithFormat:
        @"The note can be downloaded at the following url:\n%@%
        \n\n It will expire in one hour.", url]
        isHTML:NO];

    [self presentViewController:mailComposer
        animated:YES
        completion:nil];

}
```

This creates a view controller similar to the one used by the mail application in iOS. You set the subject and the body of the email.

Next, add the following method to generate the URL:

```
- (NSURL *) generateExportURL {

    NSTimeInterval oneHourInterval = 3600.0;
    NSDate *expirationInOneHourSinceNow =
    [NSDate dateWithTimeInterval:oneHourInterval
                           sinceDate:[NSDate date]];
    NSError *err;

    NSURL *url = [[NSFileManager defaultManager]
                  URLForPublishingUbiquitousItemAtURL:
                  [self.detailItem fileURL]
                  expirationDate:&expirationInOneHourSinceNow
                  error:&err];
    if (err)
        return nil;
    else
        return url;

}
```

Here you create an interval of one hour to calculate the expiration date of the download link. Then you pass it as a parameter to the `URLForPublishingUbiquitousItemAtURL:expirationDate:error:` method, which instructs iCloud to make the file downloadable for a given time. If you are not interested in an expiration date you can pass nil as parameter.

The final touch is to implement the delegate method to dismiss the mail compose view when done:

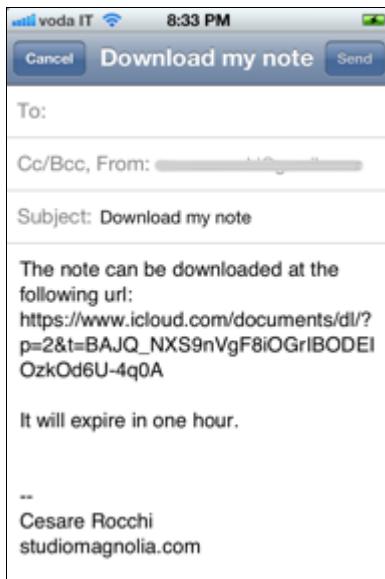
```
- (void)mailComposeController:
(MFMailComposeViewController*)controller
    didFinishWithResult:(MFMailComposeResult)result
                  error:(NSError*)error {

    if (result == MFMailComposeResultSent) {

        [self dismissViewControllerAnimated:YES
                                  completion:nil];

    }
}
```

Build and run the app, and when you tap the Export button you should see something like this:



Try sending the email to yourself, and verify you can access the note by clicking the link. Pretty cool, eh?

Note: The project up to this point is in the resources for this chapter as **dox-09**.

Congrats, you have now covered almost everything you might want to do with loading and saving data with the `UIDocument` class. Next up you're going to switch gears, and dive into using iCloud with Core Data!

What is Core Data?

Besides files, either single or packaged, iCloud allows storing information in a relational form. That means you can use iCloud in your applications as a backend database! You can store objects that are related to others, push changes on one device and receive them on another.

All this enabled by Core Data, which has been extended in iOS5 to support iCloud. Core Data is a technology introduced in iOS 3 to manage relational data. In such a model, objects can be described in terms of attributes and can be related to other objects by means of relations.

For example, a commonly known domain used to represent relational data is a company. There are different types of objects like employees, departments, offices and each is related to each other in some way. An employee has attributes like name, surname, phone number; he belongs to a department and has an office. An office, in turn, can host more than one employee and so on.

All this structured information can be stored in different ways: xml files, binary form or even SQLite. Core Data abstracts the way you usually work with such kind of model. Instead of writing sql queries to interact with your objects you will write actual Objective-C code. For example to create a new employee and assign him to a department you will use the following way (pseudo-code):

```
Department *dep ... // a department
Employee *newEmployee = [[Employee alloc] init];
newEmployee.name = "Cesare"
newEmployee.surname = "Rocchi"
newEmployee.department = dep;
```

You might be tempted to say that Core Data is an ORM (Object Relational Mapping) system. It is not, for data are not related to a database schema. Core Data is an "object graph management framework", where data are kept in memory and, when needed, persisted on disk. An object graph is a set of interrelated instances, which describes your domain. If Core Data were an ORM it would just store information in SQLite form. Instead, it allows dumping a graph in xml and binary format as well.

Apart from this, Core Data includes many features to manage relational information like:

- validation of attributes (values for properties have to be correctly typed)
- migration of schemas (when you add new objects or relations)
- support to the tracking of changes (when you update objects or relations)
- fetching, filtering and sorting (to retrieve the data you need)
- user interface integration (to notify the view of changes to the data model)
- merging (to resolve conflicts in data, especially in iCloud)

In writing a Core Data enabled application there are three key elements to deal with: model, context and coordinator. Now you will see in details their respective roles.

Model

You might be already familiar with the notion of **model**. For example, the applications built so far have been 'modeled' in terms of notes, which were defined by attributes like identifier, title, content, creation date and so on.

XCode allows defining a model graphically, much like a xib. The model defines the objects of your domain in terms of properties and relationships. It's like a schema for databases.

Objects are often referred to as **entities**, which usually correspond to classes. In your example, employees, offices and departments are entities of the company domain. Depending on the way entities are persisted on disk, entities are represented as subclasses of NSManagedObject or UIManagedDocument.

Entities are described in terms of **attributes**, which correspond to instance variables. For example, name and surname of an employee are attributes describing that entity. The fillers of attributes can be of many types like strings, numbers, dates, etc.

Relationships are the way to describe how two entities are connected.

Relationships have a cardinality which can be:

- one-to-one (one employee has one office and each office can host just one employee)
- one-to-many (one employee has one office and one office can host more than one employee)

Finally, there is concept similar to relationships called **fetched properties**. An example of fetched property in plain English is "all the employees of department a whose name starts with 'F'". Although this example relates two entities (employees and departments) it is not a relationship defined in the schema but a sort of 'temporary' relation defined in terms of rules or constraints applied to the domain.

Context

A **context**, short for managed object context, is a sort of mediator between managed objects and the functionalities of Core Data. Whenever there is change in a domain (e.g. an employee changes department) such a variation is not stored directly in the database but it is first written on "scratch paper", the context.

So whenever you write some code to change information in your domain, those variations are not immediately persisted to disk. Indeed, to make them persistent you have to explicitly tell the context: "please store all the changes I have made so far". At this point the context performs a lot of tasks such as:

- validating values (for example a filler for the name property has to be a string)
- managing undo and redo

To some extent a context is a gateway: no changes to the databases are committed if they are not first annotated into a context. This is why a context plays a central role in Core Data architecture and you will use it a lot.

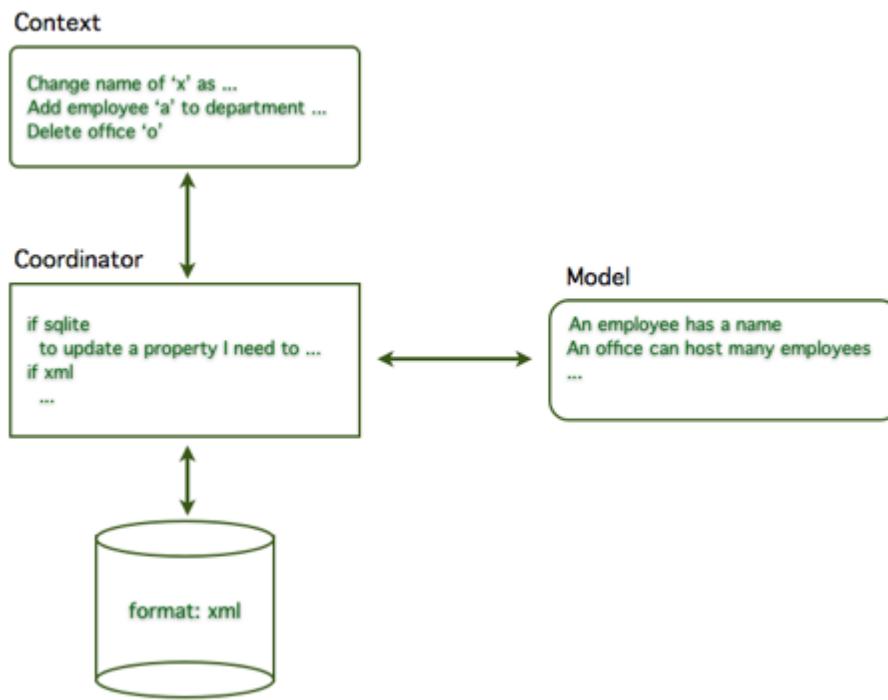
Coordinator

The **coordinator**, short for persistent store coordinator, is the mediator between a store and a context. The changes temporarily included in a context are persisted on disk by means of a coordinator, which serializes all the changes and updates the information on the disk.

While the context deals with changes at an abstract level, with data kept in memory, the coordinator performs the dirty work to store those changes according to the store type. So a coordinator has "competence" in SQLite, XML and binary

format. It is a façade that acts as a bridge between the context and the specific behaviors of the persistent store type.

I should mention that a coordinator can manage multiple contexts and stores at a time. The following image shows the interplay between model, context and coordinator.



Before digging into iCloud features you will build a simple example to show how information can be persisted on disk by means of Core Data.

Porting the Dox Application to Core Data

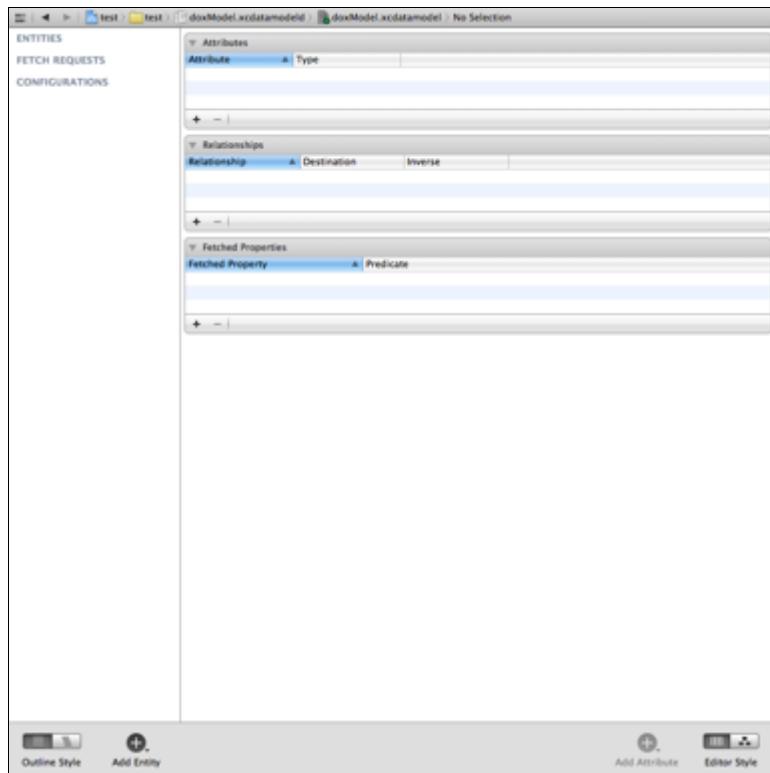
Since Core Data is ideal for relational information you are going to extend your model a bit. Now, each note will have one or more tags attached. This way you will have two entities in your domain, note and tag, related by a many-to-many relationship (each note can have multiple tags, and each tag can be assigned to multiple notes). The user interface will be pretty similar, but under the hood there are many changes to do.

If you are new to Core Data, you might want to run through the next few sections so you get some experience with Core Data and see how it fits together. But if you are already experienced with Core Data, you might want to skip to the "Porting a Core Data Application with iCloud" section later on in this chapter.

Creating A Model For Notes And Tags

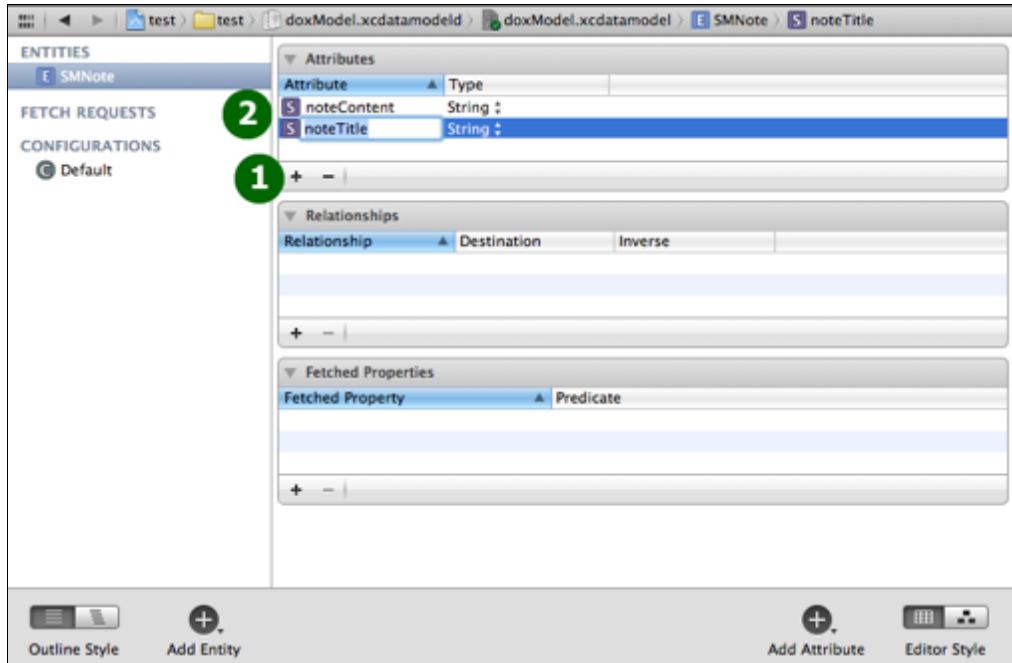
In this section you will start with the **dox-10-starter-project** in the resources for this chapter, so create a copy and edit your bundle ID, entitlements, and code signing profile as usual. It is a barebones master/detail project, with the storyboards configured correctly but nothing more. Run it on your device to check that everything works fine.

Next, create a new file with the **iOS\Core Data\Data Model** template, and name it **doxModel.xcdatamodeld**. Select the file, and you'll see the following view:



This is where you can define the entities you'll use in your project and their relationships. At the bottom there is a button **Add Entity**. Click it and call the new entity **SMNote**.

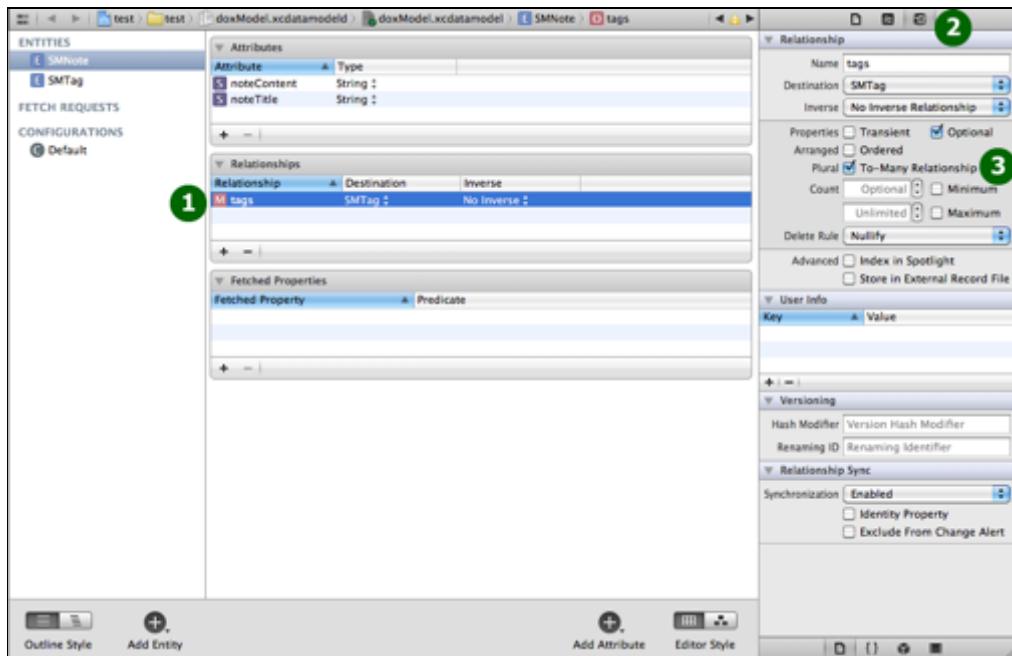
Next add two properties to the class. Click on the '+' sign in the attributes section and add two entries named `noteContent` and `noteTitle`, and set them as strings.



Next add another entity, called **SMTAG**, with just one attribute, tagContent as a string. This is the name of the tag that will be displayed on the user interface.

Now it is time to relate these new entities that you have created. As you saw above there is a many-to-many relationship between them. First select **SMNote** and click on the **+** in the **Relationships** section. Name the relation **tags** and set **SMTAG** as the destination.

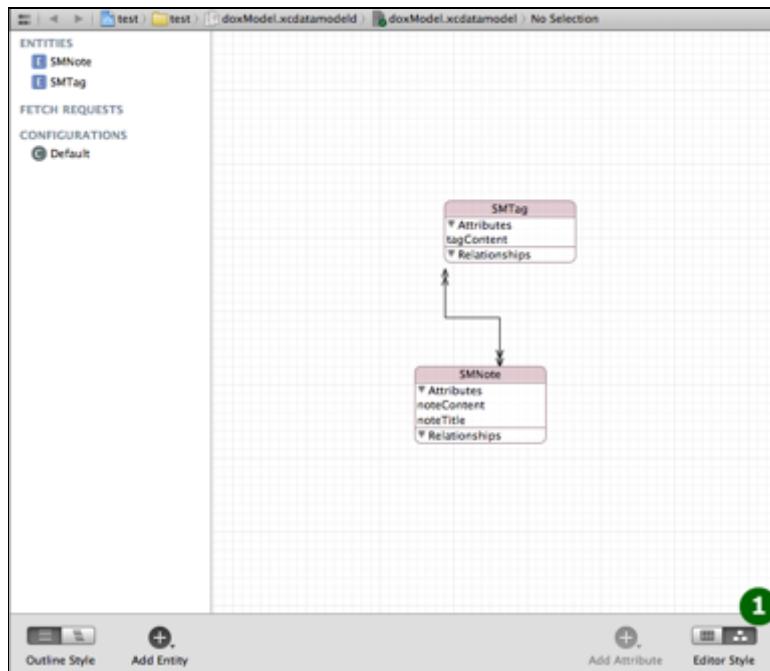
Then you should specify that the relation is of type **To-Many**. With the relation selected you open the panel on the right and you select the corresponding option.



This is just one part of relation, from a note to a tag. You should also model its counterpart.

Select the **SMTAG** entity and add a new relationship, named `notes`, with the destination **SMNote**, whose inverse relation is `tags` (the one defined previously). Also mark this as a **To-Many** relation. The inverse relationship will help Core Data keep the consistency in the model.

Now if you switch the **Editor Style** to **Graph** (click the button in the bottom right of the Core Data editor), you'll see a diagram of entities as the following:



This view can be a nice way to visualize the relationships between the entities. All the classes have the attributes and the relations specified previously.

Now you can generate classes for the entities you created in the Core Data editor. To do this, drag a rectangle on the diagram to select all the boxes and then from the main menu select **Editor\Create NSManagedObject Subclass**, and click **Create**. XCode will generate header and implementation files for both of your entities: **SMTAG** and **SMNote**.

You will notice that each class extends `NSManagedObject`, which is an abstract class that helps to interact with Core Data storage. Also relationships are modeled. The `SMNote` class will contain the following helper methods, which are needed to add/remove single or multiple tags.

```
- (void)addTagsObject:(SMTAG *)value;
- (void)removeTagsObject:(SMTAG *)value;
- (void)addTags:(NSSet *)values;
- (void)removeTags:(NSSet *)values;
```

For example, when you will need to assign a tag to a note the corresponding code will be pretty simple.

```
SMNote *note ... // a note instance
SMTAG *tag ... // a tag instance
[note addTagsObject:tag];
// save context
```

The data model for your application is ready. Now it is time to load the model in the application and prepare it to manage Core Data functionalities.

Integrating Core Data

As you learned earlier there are three elements that interplay in the Core Data-based application: the model, the context and the coordinator. But before you can add these elements to the project you have to import the Core Data framework.

To do this, select the root element of the project, then the target and open the **Link Binary with Libraries** section in the **Build Phases** tab. Click the + sign to add a new framework. You can use the search bar to look for the **CoreData.framework**. Select it and click to add it.

Now you can include the Core Data classes without any complaint from the compiler. It is better to place these elements in the application delegate so whenever the application starts up or gets reactivated you can perform the actions needed to refresh data. The three classes that you need are: `NSManagedObjectModel`, `NSPersistentStoreCoordinator` and `NSManagedObjectContext`.

Go ahead and update **SMAAppDelegate.h** as follows:

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

@interface SMAAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) NSManagedObjectContext
    *managedObjectContext;
@property (strong, nonatomic) NSManagedObjectModel
    *managedObjectModel;
@property (strong, nonatomic)
    NSPersistentStoreCoordinator *persistentStoreCoordinator;

- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;

@end
```

In addition to adding the Core Data classes you need, here you also added two helper methods to retrieve the documents directory and to save changes.

Now let's start by defining the model. Switch to **SAppDelegate.m** and add the following method:

```
- (NSManagedObjectModel *)managedObjectModel
{
    if (_managedObjectModel != nil)
    {
        return _managedObjectModel;
    }

    NSURL *modelURL = [[NSBundle mainBundle]
                         URLForResource:@"doxModel"
                         withExtension:@"momd"];

    _managedObjectModel = [[NSManagedObjectModel alloc]
                           initWithContentsOfURL:modelURL];

    return _managedObjectModel;
}
```

The task of this method is to load the schema that you have defined graphically to provide all the rules and constraints needed to verify the consistency. The model needs a URL that you build from the main bundle file of the application.

You might wonder why you are loading a file with extension "momd". The model defined in doxModel.xcdatamodeld is not copied as-is in the application folder. Instead, each file corresponding to an entity gets "compiled" into a file whose extension is .mom (which stands for managed object model). In a second step all the mom files are wrapped into a .momd file, which is the one you have to load from the bundle of the application.

Now you can use the model to create a coordinator:

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
    if (_persistentStoreCoordinator != nil)
    {
        return _persistentStoreCoordinator;
    }

    NSURL *storeURL = [[self applicationDocumentsDirectory]
                        URLByAppendingPathComponent:@"dox.sqlite"];

    NSError *error = nil;
```

```

_persistentStoreCoordinator =
    [[NSPersistentStoreCoordinator alloc]

        initWithManagedObjectModel:[self managedObjectModel]];

if (![_persistentStoreCoordinator
    addPersistentStoreWithType:NSSQLiteStoreType
        configuration:nil
            URL:storeURL
            options:nil
            error:&error])
{
    NSLog(@"Core Data error %@", error,
        [error userInfo]);
    abort();
}

return _persistentStoreCoordinator;
}

```

In your case you are going to use a SQLite store type so you need to provide a name for the file where the database will be stored, in your case 'dox.sqlite'. The coordinator is initialized with the model created above.

Once you have an instance of the coordinator, you can add the persistent store by specifying the type and the store URL. You could also provide a configuration and some options. To keep things simple, for the moment you will pass nil as a parameter. You will come back to options when you will configure Core Data for iCloud.

Next add the helper method `applicationDocumentsDirectory` as follows:

```

- (NSURL *)applicationDocumentsDirectory
{
    return [[[NSFileManager defaultManager]
        URLsForDirectory:NSDocumentDirectory
            inDomains:NSUTFUserDomainMask]
        lastObject];
}

```

And implement the method to return the context, which is initialized and associated to the coordinator:

```

- (NSManagedObjectContext *)managedObjectContext
{
}

```

```

    if (_managedObjectContext != nil)
    {
        return _managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator =
        [self persistentStoreCoordinator];

    if (coordinator != nil)
    {
        _managedObjectContext =
            [[NSManagedObjectContext alloc] init];
        [_managedObjectContext
            setPersistentStoreCoordinator:coordinator];
    }

    return _managedObjectContext;
}

```

The last thing to implement in the application delegate is the `saveContext` method:

```

- (void)saveContext {
    NSError *error = nil;

    if ([self.managedObjectContext hasChanges] &&
        ![self.managedObjectContext save:&error])
    {
        NSLog(@"Core Data error %@", error,
              [error userInfo]);
        abort();
    }
}

```

This method calls the `save` method of the context if there are changes. It can be useful to place a call to such a method when the application is put in background or quit.

```

- (void)applicationDidEnterBackground:
    (UIApplication *)application
{
    [self saveContext];
}

- (void)applicationWillTerminate:(UIApplication *)application

```

```
{  
    [self saveContext];  
}
```

Now the application is ready to interact with a database persisted on disk and based on the model you have defined previously. Let's see how you can retrieve and store notes in your new data structure.

Adding and Retrieving Notes

As mentioned above, the interaction with the data store is managed by the context. So you can use the context to get or modify the list of notes.

The first task is to show the list of notes retrieved from the store into the table view. This is done by using a `NSFetchedResultsController`, which allows defining a query which is managed by the context to fetch data.

Replace `SMMasterViewController.h` with the following:

```
#import <UIKit/UIKit.h>  
#import "SMNote.h"  
#import "SMTag.h"  
  
@class SMDetailViewController;  
  
@interface SMMasterViewController :  
    UITableViewController<NSFetchedResultsControllerDelegate>  
  
    @property (strong, nonatomic) SMDetailViewController  
        *detailViewController;  
    @property (strong, nonatomic) NSFetchedResultsController  
        *fetchedResultsController;  
    @property (strong, nonatomic) NSManagedObjectContext  
        *managedObjectContext;  
  
    @end
```

We have added a result controller and a managed object context, along with a protocol to manage callbacks from the result controller.

There are a few steps to create a fetched results controller:

1. Create a fetch request.
2. Create an entity description for the entity to be fetched (in your case, `SMNote`).
3. Assign the entity description to the request.
4. Create a sort descriptor to specify how the results should be sorted.
5. Assign the sort descriptor to the request.

6. Create a fetched results controller with the fetch request and the context.
7. Perform a fetch request to retrieve the data!

Here's the code corresponding to these steps - go ahead and add it to **SMMasterViewController.m**:

```
- (NSFetchedResultsController *)fetchedResultsController
{
    if (_fetchedResultsController != nil) {
        return _fetchedResultsController;
    }

    // 1) Create a fetch request.
    NSFetchRequest *fetchRequest =
    [[NSFetchRequest alloc] init];

    // 2) Create an entity description for the entity to be
    // fetched.
    NSEntityDescription *entity = [NSEntityDescription
        entityForName:@"SMNote"
        inManagedObjectContext:
        self.managedObjectContext];

    // 3) Assign the entity description to the request.
    [fetchRequest setEntity:entity];

    // 4) Create a sort descriptor to specify how the results
    // should be sorted.
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
        initWithKey:@"noteTitle"
        ascending:NO];
    NSArray *sortDescriptors =
    [NSArray arrayWithObjects:sortDescriptor, nil];

    // 5) Assign the sort descriptor to the request.
    [fetchRequest setSortDescriptors:sortDescriptors];

    // 6) Create a fetched results controller with the fetch
    // request and the context.
    NSFetchedResultsController *aFetchedResultsController =
    [[NSFetchedResultsController alloc]
        initWithFetchRequest:fetchRequest
        managedObjectContext:self.managedObjectContext
        sectionNameKeyPath:nil
        cacheName:@"Master"];
}
```

```
aFetchedResultsController.delegate = self;
self.fetchedResultsController = aFetchedResultsController;

// 7) Perform a fetch request to retrieve the data!
NSError *error = nil;
if (![self.fetchedResultsController performFetch:&error]) {

    NSLog(@"Unresolved error %@, %@", error, [error
userInfo]);

}

return _fetchedResultsController;

}
```

To keep things simple, in creating the controller you did not provide any section path (see the Core Data documentation for more details) and you specified a name for the cache to quicken the retrieval time.

Finally you called `performFetch:` to load data when the instance variable is created. You might have noticed that you have specified 'self' as the delegate of the controller.

Next you are going to implement the fetched result controller delegate's `controllerDidChangeContent:` method, which is called when the result controller has processed some change to data. Implement it to reload data in the table view:

```
- (void)controllerDidChangeContent:
    (NSFetchedResultsController *)controller {

    NSLog(@"something has changed");
    [self.tableView reloadData];

}
```

Next, replace `insertNewObject:` with the following:

```
- (void)insertNewObject:(id)sender
{
    SMNote *newNote =
    [NSEntityDescription
        insertNewObjectForEntityForName:@"SMNote"
        inManagedObjectContext:self.managedObjectContext];

    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"yyyyMMdd_hhmmss"];
```

```
NSString *noteTitle = [NSString stringWithFormat:@"Note_%@",
    [formatter stringFromDate:[NSDate
        date]]];
newNote.noteTitle = noteTitle;
newNote.noteContent = @"New note content";

NSError *error = nil;
if (![self.managedObjectContext save:&error]) {
    NSLog(@"Core Data error %@", error,
        [error userInfo]);
}
}
```

Unlike previous examples, notes populating the table view are not stored in an array inside the class, but rather the Core Data the store itself. So when you add a new note you instantiate it in the context, you set its properties and you save the context.

Here the note is not created via the usual alloc/init, but instead with `NSEntityDescription's insertNewObjectForEntityForName:inManagedObjectContext:` method (this is what you have to do with Core Data).

Once you have an instance you have access to its properties as usual. The `save:` method of the context, once done, will trigger `controllerDidChangeContent:` which will reload data in the table. Finally, you have just to implement the table view delegates to use the data returned by the fetched results controller:

```
- (NSInteger)numberOfSectionsInTableView:
    (UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    id <NSFetchedResultsSectionInfo> sectionInfo =
        [[self.fetchedResultsController sections]
            objectAtIndex:section];
    return [sectionInfo numberOfObjects];
}

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

```

    NSManagedObject *managedObject =
        [self.fetchedResultsController
            objectAtIndexAtIndexPath:indexPath];

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"Cell"
            forIndexPath:indexPath];

    cell.textLabel.text = [managedObject
        valueForKey:@"noteTitle"];

    return cell;
}

```

This way data persisted on disk will be displayed in the table view as in previous examples. As a final step, open **SAppDelegate.m** and application:didFinishLaunchingWithOptions: to this:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad) {
        UISplitViewController *splitViewController =
            (UISplitViewController*)self.window.rootViewController;
        UINavigationController *navigationController =
            [splitViewController.viewControllers lastObject];
        splitViewController.delegate =
            (id)navigationController.topViewController;

        UINavigationController *masterNavigationController =
            splitViewController.viewControllers[0];
        SMMasterViewController *controller =
            (SMMasterViewController *)
            masterNavigationController.topViewController;
        controller.managedObjectContext =
            self.managedObjectContext;
    } else {
        UINavigationController *navigationController =
            (UINavigationController *)self.window.rootViewController;
        SMMasterViewController *controller =
            (SMMasterViewController *)
            navigationController.topViewController;
        controller.managedObjectContext =
            self.managedObjectContext;
    }
}

```

```
}

id currentToken =
    [[NSFileManager defaultManager] ubiquityIdentityToken];
if (currentToken) {
    NSLog(@"iCloud access on with id %@", currentToken);
} else {
    NSLog(@"No iCloud access");
}
return YES;
}
```

Phew, finally done! Run the application and add a few notes:



As you can see in the screenshot, the table view is correctly populated after each new addition. If you quit and restart the application, the data is properly reloaded from the Core Data SQLite database. Nice!

If you want to show a single note you have to push an instance of **SMDetailViewController**. So open **SMMasterViewController.m** and add the implementation for `tableView:didSelectRowAtIndexPath:` and `prepareForSegue:sender:` methods as follows:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad)
    {
        SMNote *n = (SMNote *)[self.fetchedResultsController
```

```

        objectAtIndexPath:indexPath];
self.detailViewController.detailItem = n;

}

}

- (void)prepareForSegue:(UIStoryboardSegue *)segue
    sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"showDetail"]) {

        NSIndexPath *indexPath = [self.tableView
                                  indexPathForSelectedRow];
        SMNote *n = (SMNote *)[self.fetchedResultsController
                               objectAtIndex:indexPath];
        self.detailViewController.detailItem = n;
        [[segue destinationViewController] setDetailItem:n];

    }
}

```

Now let's move on to the visualization of a single note and its tags.

Single Note and Tags

Now it is time to rework the user interface of `SMDetailViewController` to show the tags in a label. Replace **SMDetailViewController.h** with the following:

```

#import <UIKit/UIKit.h>
#import "SMNote.h"
#import "SMTag.h"

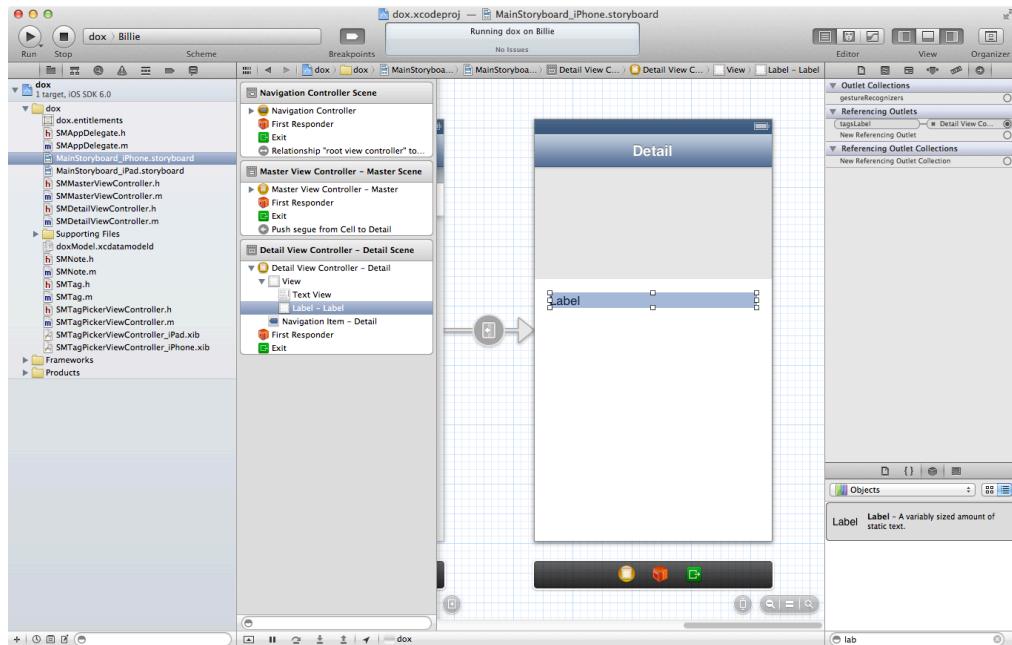
@interface SMDetailViewController : UIViewController
<UISplitViewControllerDelegate>

@property (strong, nonatomic) SMNote *detailItem;
@property (weak, nonatomic) IBOutlet UITextView *noteTextView;
@property (weak, nonatomic) IBOutlet UILabel *tagsLabel;

@end

```

You have to squeeze the text view a bit to accommodate a new label to show tags related to a note. Open the storyboards, resize the text view, and add a new label at the bottom. Then connect the label to `tagsLabel` outlet declared in the header file, as shown in the following screenshot:



The next step is to show the note content and the tags when a note is selected from the list. In the case of the iPhone you do it in `viewWillAppear:`, to be added in **SMDetailViewController.m**:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    NSArray *tagsArray = [self.detailItem.tags allObjects];
    NSMutableArray *tagNames =
    [NSMutableArray arrayWithCapacity:tagsArray.count];

    for (SMTag *t in tagsArray) {
        [tagNames addObject:t.tagContent];
    }

    if (tagNames.count == 0) {
        self.tagsLabel.text = @"Tap to add tags";
    } else {
        NSString *s = [tagNames componentsJoinedByString:@", "];
        self.tagsLabel.text = s;
    }
    [self.noteTextView becomeFirstResponder];
}
```

With the iPad you do it in `configureView`.

```
- (void)configureView
{
```

```

if (self.detailItem) {

    self.noteTextView.text = self.detailItem.noteContent;
    self.title = self.detailItem.noteTitle;

    NSArray *tagsArray = [self.detailItem.tags allObjects];
    NSMutableArray *tagNames =
        [NSMutableArray arrayWithCapacity:tagsArray.count];

    for (SMTAG *t in tagsArray) {
        [tagNames addObject:t.tagContent];
    }

    NSString *s = [tagNames componentsJoinedByString:@", "];
    self.tagsLabel.text = s;
    [self.noteTextView becomeFirstResponder];
}
}

```

In both cases this view controller does not need a fetch result controller, because it deals with only one note. Tags, although stored in the persistent store, are referenced as a property of a note. If you check the code generated for the class `SMNote` you will notice the following property.

```
@property (nonatomic, retain) NSSet *tags;
```

This is exactly the way the has-tags relation is modeled, by means of a set. This set is populated with the instances of tags related to a note. To populate the label you have transformed the set in an array and concatenated its elements with commas.

A note gets saved when the user taps the 'save' button. Unlike previous example, here you call the `save:` method of the context to store changes. Add the implementation of `saveEdits:` as following:

```

- (void) saveEdits:(id)sender {
    self.detailItem.noteContent = self.noteTextView.text;

    NSError *error = nil;
    if (![self.detailItem.managedObjectContext save:&error]) {
        NSLog(@"Core data error %@", error,
              [error userInfo]);
        abort();
    }
}

```

A new note comes with no tags, so you have to create a way to assign one or more tags to a note. In the next step you will build a tag picker controller. Since you are editing the single note controller let's add the code to show the picker at the end of viewDidLoad. The action is triggered when the user taps the tags label. You associate this action once the view is loaded.

```
self.tagsLabel.userInteractionEnabled = YES;
UITapGestureRecognizer *tapGesture =
    [[UITapGestureRecognizer alloc]
        initWithTarget:self
        action:@selector(tagsTapped)];
[self.tagsLabel addGestureRecognizer:tapGesture];
```

The action tagsTapped creates an instance of a new controller (which you are going to create in a bit), assign it the current note and push it onto the navigation stack. As usual you have to make the difference between the iPhone and the iPad version.

```
- (void) tagsTapped {
    SMTagPickerController *tagPicker = nil;
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone) {
        tagPicker =
            [[SMTagPickerController alloc]
                initWithNibName:@"SMTagPickerController_iPhone"
                bundle:nil];
    } else {
        tagPicker =
            [[SMTagPickerController alloc]
                initWithNibName:@"SMTagPickerController_iPad"
                bundle:nil];
    }
    tagPicker.currentNote = self.detailItem;
    [self.navigationController pushViewController:tagPicker
        animated:YES];
}
```

Of course for this to work you have to import the header you'll write in a minute at the top of the file:

```
#import "SMTagPickerController.h"
```

Let's now define the behavior of the tag picker.

Building The Tag Picker

This component is meant to show the list of available tags so that the user can pick one or more and associate them to a note.

Create a new file with the **iOS\Cocoa Touch\Objective-C** class template. Name the class **SMTagPickerController**, make it a subclass of **UITableViewController**, and make sure that With XIB for user interface is checked.

Rename the newly created xib to **SMTagPickerController_iPhone.xib**, and create a xib for the iPad as well named **SMTagPickerController_iPad.xib**, with a table view like the one of the iPhone.

Next replace **SMTagPickerController.h** with the following:

```
#import <UIKit/UIKit.h>
#import "SMNote.h"
#import "SMTag.h"

@interface SMTagPickerController : UITableViewController

@property (nonatomic, strong) SMNote *currentNote;
@property (nonatomic, strong) NSMutableSet *pickedTags;
@property (strong, nonatomic) NSFetchedResultsController
    *fetchedResultsController;

@end
```

The `currentNote` will keep a reference to the note shown in the previous view controller, the set will store selected tags and the result controller will contain all the tags available in your domain.

Switch to **SMTagPickerController.m** to implement the fetch controller. This will perform a different query, focused on the `smtag` entity. The pattern is exactly the same presented above: create a request, set some properties on it, and initialize a fetched results controller with the request and the context.

```
- (NSFetchedResultsController *)fetchedResultsController
{
    if (_fetchedResultsController != nil) {
        return _fetchedResultsController;
    }

    NSFetchedResultsController *fetchRequest =
        [[NSFetchedResultsController alloc] init];
```

```
NSEntityDescription *entity =
    [NSEntityDescription
        entityForName:@"SMTag"
inManagedObjectContext:self.currentNote.managedObjectContext];

[fetchRequest setEntity:entity];

NSSortDescriptor *sortDescriptor =
    [[NSSortDescriptor alloc]
        initWithKey:@"tagContent"
        ascending:NO];
NSArray *sortDescriptors =
    [NSArray arrayWithObjects:sortDescriptor, nil];
[fetchRequest setSortDescriptors:sortDescriptors];

NSFetchedResultsController *aFetchedResultsController =
    [[NSFetchedResultsController alloc]
        initWithFetchRequest:fetchRequest
        managedObjectContext:self.currentNote.managedObjectContext
        sectionNameKeyPath:nil
        cacheName:@"Master"];

self.fetchedResultsController = aFetchedResultsController;

NSError *error = nil;
if (![self.fetchedResultsController performFetch:&error]) {

    NSLog(@"Core data error %@", %@", error, [error userInfo]);
    abort();
}

return _fetchedResultsController;
}
```

In this specific case you do not need a reference to the context created in the application delegate, because each class generated from the schema contains a reference to that exact context. So in this case you use `self.currentNote.managedObjectContext`.

To keep things simple, you want to avoid creating another view controller to enter new tags. So you will populate your application with a static list of tags if none are present. You will perform this task when the tag picker has loaded.

First you perform a query and if that is empty you create a few tags and save them in the context. After saving them you reload tags from the persistent store.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.pickedTags = [[NSMutableSet alloc] init];
    self.title = @"Tag your note";

    // Retrieve tags
    NSError *error;
    if (![self.fetchedResultsController performFetch:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    // in case there are no tags,
    // presumably it's the first time you run the application
    if (self.fetchedResultsController.fetchedObjects.count == 0)
    {
        for (int i = 1; i < 6; i++) {

            SMTag *t = [NSEntityDescription
                        insertNewObjectForEntityForName:@"SMTag"
                        inManagedObjectContext:
                        self.currentNote.managedObjectContext];
            t.tagContent =
            [NSString stringWithFormat:@"tag%i", i];

        }
    }

    // Save to new tags
    NSError *error = nil;
    if (![self.currentNote.managedObjectContext
          save:&error]) {
        NSLog(@"Unresolved error %@, %@", error,
              [error userInfo]);
        abort();
    } else {
        // Retrieve tags again
        NSLog(@"new tags added");
        [self.fetchedResultsController performFetch:&error];
    }
}
```

```

    // Each tag attached to the note should be included
    // in the pickedTags array
    for (SMTag *tag in self.currentNote.tags) {
        [self.pickedTags addObject:tag];
    }
}

```

This code is executed each time the tag picker is loaded but new tags are added just once, the first time the application is run. You will assign the tag selected when the user is about to close the picker, in the `viewWillDisappear:` method.

```

- (void) viewWillDisappear:(BOOL)animated {

    [super viewWillDisappear:animated];
    self.currentNote.tags = self.pickedTags;

    NSError *error = nil;
    if (![self.currentNote.managedObjectContext save:&error]) {
        NSLog(@"Core data error %@", @"",
              error, [error userInfo]);
        abort();
    }
}

```

Now you have the entire infrastructure needed to create the tag picker. You are left with connecting the user interface with the result controller. As previously the table is populated by the result of the fetch controller.

```

- (NSInteger)numberOfSectionsInTableView:
    (UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    id <NSFetchedResultsSectionInfo> sectionInfo =
        [[self.fetchedResultsController sections]
            objectAtIndex:section];
    return [sectionInfo numberOfRowsInSection];
}

```

Each cell will show the string stored in the `tagContent` property of each tag. The cell will also display a checkmark if the tag is selected (associated with the note).

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell =
    [tableView
     dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier];
    }

    cell.accessoryType = UITableViewCellAccessoryNone;
    SMTag *tag = (SMTag *)[self.fetchedResultsController
                           objectAtIndex:indexPath];
    if ([self.pickedTags containsObject:tag]) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }
    cell.textLabel.text = tag.tagContent;
    return cell;
}
```

Finally when you tap a cell you will add or remove the tag from the array of picked tags.

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    SMTag *tag = (SMTag *)[self.fetchedResultsController
                           objectAtIndex:indexPath];
    UITableViewCell * cell = [self.tableView
                               cellForRowAtIndexPath:indexPath];
    [cell setSelected:NO animated:YES];

    if ([self.pickedTags containsObject:tag]) {

        [self.pickedTags removeObject:tag];
        cell.accessoryType = UITableViewCellAccessoryNone;

    } else {

        [self.pickedTags addObject:tag];
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }
}
```

```
}
```

w00t - finally time to build and run the application! Go ahead and try adding a few notes, editing the text, and assigning one or more tags. Note that to save a tag selection you have to dismiss the tag picker view, and to save note edits you have to put the application in background or quit it.

Here are a few screenshots of the application at work:

Congrats, you have built a good old Core Data application!

Note: The project up to this point is in the resources for this chapter as **dox-10**.

Now it is time to see how to port it to iCloud, so that all the modifications made to the persistent store are propagated to the cloud and other devices.

Porting a Core Data Application to iCloud

In iOS 5 Core Data was extended to support iCloud. One key addition is related to the event that happens when you push new changes to the store. Each change is saved into a transaction, which in turn is pushed to iCloud.

Much like `UIDocument`, only changes are propagated to the cloud, making the mechanism much more efficient than pushing a whole new file at each change, especially if the database is large.

In a Core Data scenario, changes are propagated to iCloud when you call the `save` method of a context. The steps needed to get this working are:

1. Provide a location to store transaction operations in iCloud
2. Observe new notifications coming from iCloud to correctly update the local persistent store.
3. Update the user interface according to new changes in the persistent store.

We will adapt the project created in the previous section (called **dox-10** if you jumped here from earlier) to enable iCloud synchronization.

There are two fundamental changes to make: one is related to the configuration of the persistent store and the other deals with the context. Both elements have to be made iCloud-aware.

Another important aspect is that, unlike in the previous project, you are in an asynchronous situation, since data from the cloud might take a while to be

downloaded. That's why you have to use a secondary thread to instantiate the store coordinator.

Let's proceed step by step. In the previous example, in the `SMAppDelegate`, you provided no options to the

`addPersistentStoreWithType:configuration:URL:options:error:` method. In this case you have to create an array of options, in which you specify the features to be enabled in the persistent store. Three are the options you are interested in:

- `NSPersistentStoreUbiquitousContentNameKey`: A unique name that identifies the store in the ubiquity container
- `NSPersistentStoreUbiquitousContentURLKey`: The path to store log transactions to the persistent store
- `NSMigratePersistentStoresAutomaticallyOption`: A boolean value to allow automatic migrations in the store if needed

The key is arbitrary so you can provide the name you like, such as `@"com.studiomagnolia.coredata.notes"`.

The URL for transaction logs is built by creating a subdirectory in the ubiquity container, as follows (you don't need to add this quite yet, you'll add the complete method soon):

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *transactionLogsURL =
    [fileManager URLForUbiquityContainerIdentifier:nil];
NSString* coreDataCloudContent = [[transactionLogsURL path]
    stringByAppendingPathComponent:@"dox_data"];
transactionLogsURL = [NSURL
    fileURLWithPath:coreDataCloudContent];
```

So the resulting array of options can be defined like this:

```
NSDictionary* options =
[NSDictionary dictionaryWithObjectsAndKeys:
 @"com.studiomagnolia.coredata.notes",
 NSPersistentStoreUbiquitousContentNameKey,
 transactionLogsURL,
 NSPersistentStoreUbiquitousContentURLKey,
 [NSNumber numberWithBool:YES],
 NSMigratePersistentStoresAutomaticallyOption,
 nil];
```

The final step is to add the store to the coordinator as in the previous project but there is a catch. You are in a threaded situation so it is preferable to prevent other threads executions by adding a lock mechanism as follows.

```
// psc is an instance of persistent store coordinator
```

```

NSError *error = nil;
[psc lock];
if (![psc addPersistentStoreWithType:NSSQLiteStoreType
                           configuration:nil
                                 URL:storeUrl
                               options:options
                                 error:&error]) {

    NSLog(@"Core data error %@", error, [error userInfo]);
    abort();
}
[psc unlock];

```

Once the store has been added you can post a notification to refresh the user interface.

```

dispatch_async(dispatch_get_main_queue(), ^{
    NSLog(@"persistent store added correctly");
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"com.studiomagnolia.refetchNotes"
          object:self
        userInfo:nil];
});

```

Here is the complete code for the new `persistentStoreCoordinator` method, with comments. Replace the current implementation in **SAppDelegate.m** with the following:

```

- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {

    if (_persistentStoreCoordinator != nil) {
        return _persistentStoreCoordinator;
    }

    _persistentStoreCoordinator =
    [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel: [self managedObjectModel]];

    NSPersistentStoreCoordinator* psc =
    _persistentStoreCoordinator;
    NSURL *storeUrl = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"dox.sqlite"];

    // done asynchronously since it may take a while

```

```
// to download preexisting iCloud content
dispatch_async(dispatch_get_global_queue(
    DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // building the path to store transaction logs
    NSFileManager *fileManager = [NSFileManager
        defaultManager];
    NSURL *transactionLogsURL = [fileManager
        URLForUbiquityContainerIdentifier:nil];
    NSString* coreDataCloudContent = [[transactionLogsURL
        path]
        stringByAppendingPathComponent:@"dox_data"];
    transactionLogsURL = [NSURL
        fileURLWithPath:coreDataCloudContent];

    // Building the options array for the coordinator
    NSDictionary* options =
    [NSDictionary dictionaryWithObjectsAndKeys:
        @"com.studiomagnolia.coredata.notes",
        NSPersistentStoreUbiquitousContentNameKey,
        transactionLogsURL,
        NSPersistentStoreUbiquitousContentURLKey,
        [NSNumber numberWithBool:YES],
        NSMigratePersistentStoresAutomaticallyOption,
        nil];

    NSError *error = nil;
    [psc lock];
    if (![psc addPersistentStoreWithType:NSSQLiteStoreType
        configuration:nil
        URL:storeUrl
        options:options
        error:&error]) {
        NSLog(@"Core data error %@", %@", error, [error userInfo]);
    }
    [psc unlock];

    // post a notification to tell the main thread
    // to refresh the user interface
    dispatch_async(dispatch_get_main_queue(), ^{
        NSLog(@"persistent store added correctly");
        [[NSNotificationCenter defaultCenter]
            postNotificationName:
            @"com.studiomagnolia.refetchNotes"];
```

```

        object:self
        userInfo:nil];
    });

}

return _persistentStoreCoordinator;
}

```

The next step is to revise the implementation of the context. In an iCloud-enabled scenario the context has to be initialized according to a concurrency type, that is the way the context is bound to threads. Since views and controllers are already bound with the main thread it is appropriate to adopt the same for the context, by choosing a `NSMainQueueConcurrencyType` as follows:

```

NSManagedObjectContext* moc =
[[NSManagedObjectContext alloc]
initWithConcurrencyType: NSMainQueueConcurrencyType];

```

This means that all the code executed by the context will be performed on the main thread. When you send messages to a context which implements a queue like this you have to use either the `performBlock:` or the `performBlockAndWait:` method. This is due to the new queue-based nature of the context. The first method is synchronous, while the second is asynchronous. For example to set the coordinator you can use:

```

[moc performBlockAndWait:^{
    [moc setPersistentStoreCoordinator: coordinator];
    // other configurations
}];

```

As you learned above, one of the keys to an iCloud-enabled Core Data application is to listen for notifications about changes to the persistent store. This step has to be performed when you define the context. In this case the notification has a very long name: `NSPersistentStoreDidImportUbiquitousContentChangesNotification`. You can set up an observer like this:

```

[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(mergeChangesFrom_iCloud:)
name:
NSPersistentStoreDidImportUbiquitousContentChangesNotification
object:coordinator];

```

Summing up the final implementation of `managedObjectContext` is the following:

```

- (NSManagedObjectContext *)managedObjectContext
{
    if (_managedObjectContext != nil) {
        return _managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator =
    [self persistentStoreCoordinator];

    if (coordinator != nil) {
        // choose a concurrency type for the context
        NSManagedObjectContext* moc =
        [[NSManagedObjectContext alloc]
         initWithConcurrencyType:NSMainQueueConcurrencyType];

        [moc performBlockAndWait:^{
            // configure context properties
            [moc setPersistentStoreCoordinator: coordinator];
            [[NSNotificationCenter defaultCenter]
             addObserver:self
             selector:@selector(mergeChangesFrom_iCloud:)
             name:

            NSPersistentStoreDidImportUbiquitousContentChangesNotification
            object:coordinator];
        }];
        _managedObjectContext = moc;
    }
    return _managedObjectContext;
}

```

The selector associated to the notification is defined as follows. To go back to the main thread you use the `performBlock:` API.

```

- (void)mergeChangesFrom_iCloud:(NSNotification *)notification {

    NSManagedObjectContext* moc = [self managedObjectContext];
    [moc performBlock:^{
        [self mergeiCloudChanges:notification
                           forContext:moc];
    }];
}

```

You should remember to add this method signature to the header file.

```
- (void)merge iCloudChanges:(NSNotification*)note
    forContext:(NSManagedObjectContext*)moc;
```

The actual method that performs the merging is defined as follows.

```
- (void)merge iCloudChanges:(NSNotification*)note
    forContext:(NSManagedObjectContext*)moc {

    [moc mergeChangesFromContextDidSaveNotification:note];
    //Refresh view with no fetch controller if any

}
```

The method `mergeChangesFromContextDidSaveNotification:` does some sort of magic by updating the objects that have been notified as changed, inserted or deleted. If you need to have a higher control on the merging, as you will learn below.

Once this code is executed all the views that embed a fetch controller do not need any notification, because each instance of a fetch controller listens for changes by means of the context. In case there are views with no instance of `NSFetchedResultsController`, here you should post a notification and hook up those view with the same notification name.

You might remember you have setup a notification when you add the persistent store to the coordinator. It is important to catch it and refresh the user interface.

In **SMMasterViewController.m**, once the view has loaded, you listen for such a notification. Add this at the bottom of `viewDidLoad`:

```
[ [NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(reloadNotes)
    name:@"com.studiomagnolia.refetchNotes"
    object:nil];
```

The method triggered by the notification performs a retrieval and reloads data in the table view as follows:

```
- (void) reloadNotes {
    NSLog(@"refetching notes");
    NSError *error = nil;
    if (![[self fetchedResultsController] performFetch:&error])
    {
        NSLog(@"Core data error %@", error, [error userInfo]);
    } else {
        NSLog(@"reloadNotes - results are %i",
              self.fetchedResultsController.fetchedObjects.count);
```

```
[self.tableView reloadData];  
}  
}
```

The rest of the code is untouched, since all the views are already populated by Core Data contents.

Believe it or not, you are done! As you can see, it's really easy to make your Core Data applications synch with iCloud.

Build and run and test out the application. You should follow the usual protocol: install on two devices and see changes propagated from one to another.

The code for this section is in the repository under the name "dox-11".

Merging Conflicts in Core Data Apps

In the last application you have seen that merging can be done by means of a simple call to the `mergeChangesFromContextDidSaveNotification:` method. For sake of completeness I should mention that, if you like to dig deeper, there are many more aspects that you might want to consider.

In this section I won't guide you through the process of building an application, for conflict resolution is very specific to the purpose of the application. You will provide an explanation of the capabilities of iCloud when it comes to merging conflicting data.

For example, a context can have different merge policies. As long as the user changes different fields on the same note, the application raises no issues and merges changes. For example, if you edit a note's content on one device and the same note's tags on another, after a while changes will be propagated on both devices with no issues. In this case Core Data is smart enough to "make the sum" of edits. Problems arise when you edit the same property of the same note.

The first way to approach this issue is by adopting a merge policy in the context. You did not mention before that the default policy for a context is `NSErrorMergePolicy`, meaning that an error is thrown when you try to save a context that includes conflicts. Alternative policies are the following:

`NSMergeByPropertyStoreTrumpMergePolicy`: If there is a conflict between the persistent store and the in-memory version of an object, external changes win over the in-memory ones.

`NSMergeByPropertyObjectTrumpMergePolicy`: Similar to the above, but in-memory changes win.

`NSOverwriteMergePolicy`: Changes are overwritten in the persistent store.

`NSRollbackMergePolicy`: In-memory changes to conflicting objects are discarded.

In all these policies the conflict resolution is made for each record, in your case for each note. If you are happy with one of these you can just change the declaration of the context and leave the rest unchanged.

Otherwise, you can actually unfold what is being in conflict and try to resolve it by adopting a custom policy. All this happens in the `mergeiCloudChanges:forContext:` method. For example, when you add a note, if you print out the notification object

```
- (void)mergeiCloudChanges:(NSNotification*)notification
    forContext:(NSManagedObjectContext*)moc {

    NSLog(@"%@", notification);

}
```

we should see in the console a message like the following:

```
NSConcreteNotification 0x37fec0 {name =
    com.apple.coredata.ubiquity.importer.didfinishimport;
object = <NSPersistentStoreCoordinator: 0x353f30>;
userInfo = {
    deleted = "{(\n)}";
    inserted = "{(\n      0x36d050 <x-coredata://9F154E9E-658A-
4653-8981-
F3619AE5FE18/SMNote/p5>\n)}";
    updated = "{(\n)}";
}}
```

Here the key is that the `userInfo` array contains keyed lists of changes categorized in deletions, insertions and updates. You receive this notification each time you update, delete or create a new note. The `userInfo` is a dictionary directly attached to the notification, so if you want to unfold its content you can use the key corresponding to the change you are keeping track of, as in the following code.

```
- (void)mergeiCloudChanges:(NSNotification*)note
    forContext:(NSManagedObjectContext*)moc {

    NSDictionary *noteInfo = [note userInfo];
    NSMutableDictionary *localUserInfo =
        [NSMutableDictionary dictionaryWithDictionary:noteInfo];
    NSSet* allInvalidations = [noteInfo
        objectForKey:NSInvalidatedAllObjectsKey];

    NSLog(@"insertions = %@", [noteInfo objectForKey:NSInsertedObjectsKey]);
    NSLog(@"deletions = %@", [noteInfo objectForKey:NSDeletedObjectsKey]);
```

```
        NSLog(@"updates = %@",  
      [noteInfo objectForKey: NSUpdatedObjectsKey]);  
}
```

At this point you have all the elements to detect which objects are in conflict. The output result of your policy has to be another dictionary that you pass to the `mergeChangesFromContextDidSaveNotification:` method. So a schema of a custom policy for conflict resolution might be like this.

```
- (void)merge iCloudChanges:(NSNotification*)note  
forContext:(NSManagedObjectContext*)moc {  
  
    NSMutableDictionary *mergingPolicyResult =  
    [NSMutableDictionary dictionaryWithCapacity: 1];  
  
    // do something with insertions  
    // do something with deletions  
    // do something with updates  
  
    NSNotification *saveNotification = [NSNotification  
    notificationWithName: NSManagedObjectContextDidSaveNotification  
    object: self  
    userInfo: mergingPolicyResult];  
  
    [moc mergeChangesFromContextDidSaveNotification:  
     saveNotification];  
    [moc processPendingChanges];  
}
```

Once you have established the content of `mergingPolicyResult` you build a notification object wrapping that dictionary and a `NSManagedObjectContextDidSaveNotification` notification type. To force the context to update the object graph you call `processPendingChanges`. The "do something" part is pretty trivial. Each `objectForKey:` call returns a set of objects, that you can include or not in the `mergingPolicyResult`.

```
// do something with updates  
NSDictionary *noteInfo = [note userInfo];  
  
NSSet *updatedObjects = [noteInfo  
                        objectForKey: NSUpdatedObjectsKey];  
NSMutableSet *objectsToBeAccepted = [NSMutableSet set];
```

```

for (NSManagedObjectID *updatedObjectID in updatedObjects) {

    [objectsToBeAccepted addObject:
        [moc objectWithID:updatedObjectID]];
}

[mergingPolicyResult setObject:objectsToBeAccepted
    forKey:NSUpdatedObjectsKey];

```

This way you have the opportunity to filter out insertions and deletions. The same mechanism can be applied to deletions and modifications.

During development and debugging you might end up with some warning or error due to lo loading to Core Data changes in iCloud. To avoid that you might want to refresh sometimes the URL of transaction logs.

Prevent Synching With iCloud

As a final note, there are scenarios in which your application stores data that have to be kept permanent on disk but are not needed to be synched with iCloud. In this case there is an attribute, available since iOS 5.0.1, which allows you to mark a file to be skipped for backup. Here is the code if you need it:

```

#include <sys/xattr.h>

- (BOOL)addSkipBackupAttributeToItemAtURL:(NSURL *)URL
{
    const char* filePath = [[URL path]
                           fileSystemRepresentation];
    const char* attrName = "com.apple.MobileBackup";
    u_int8_t attrValue = 1;

    int result = setxattr(filePath, attrName, &attrValue,
                          sizeof(attrValue), 0, 0);
    return result == 0;
}

```

If you are targeting iOS5.1 or later there is a shorter way to do it.

```

NSError *error = nil;

BOOL success = [URL setResourceValue:
    [NSNumber numberWithBool: YES]

```

```
    forKey: NSURLIsExcludedFromBackupKey  
    error: &error];  
  
if(!success){  
  
    NSLog(@"Error excluding %@ from backup %@",  
          [URL lastPathComponent], error);  
  
}
```

Where To Go From Here?

Wow, this was a long journey - but now you have a lot of experience working with iCloud that you can leverage in your own apps.

iCloud is probably the biggest feature introduced in iOS 5. Including iCloud support in your new or existing applications is your call. Nonetheless it is undeniable that your application (and your income!) will likely profit from such a cool new tool.

As a general rule, you should add iCloud support only if you foresee benefits for the end users of your application. In case you are building an 'ecosystem', e.g. an iOS and MacOS application, it is very likely that you can leverage all the power of iCloud to enrich the experience of your apps.

We hope to see an iCloud-enabled app from you soon! :]

Chapter 8: Beginning OpenGL ES 2.0 with GLKit

By Ray Wenderlich

OpenGL ES is the lowest level graphics API on the iPhone, that you can use to draw things to the screen quickly and efficiently, leveraging the power of the graphics card. It most commonly used in games, and many libraries you may know and love such as Cocos2D, Corona, or Unity3D are built on top of OpenGL ES.

Prior to iOS 5, there were a lot of common frustrations when making OpenGL ES apps or games:

- **Tons of boilerplate.** Creating the simplest possible OpenGL ES app that rendered a triangle to the screen took a ton of code – things like setting up render buffers, compiling shaders, and more. This made it extremely difficult to get started, since there was so much prerequisite code and knowledge required.
- **Hard to load textures.** Although you'd think it would be simple to load a texture to use in an OpenGL ES app, it wasn't – you had to write a ton of code yourself to get this working. The worst part was this code was quite complicated and easy to get wrong.
- **No math libraries.** Writing apps or games that use OpenGL ES use a lot of math, and you commonly need functions to work with vectors and matrices. But since there was no vector and matrix math libraries included with the library, each project would add their own implementation, adding extra work to the project and making project less reusable.
- **No easy transition to OpenGL ES 2.0.** In the old days, people wrote games for the iPhone using OpenGL ES 1.0, but the latest and most powerful graphics API on the iPhone is OpenGL ES 2.0. It's best to use OpenGL ES 2.0 these days when possible, but if you had an app already written in OpenGL ES 1.0 it wasn't very easy to transition to OpenGL ES 2.0.

But since iOS 5, Apple has introduced a new set of APIs known as GLKit that makes developing with OpenGL ES much easier than it used to be!

GLKit contains four main sections:

1. **GLKView/GLKViewController:** These classes abstract out much of the boilerplate code it used to take to set up a basic OpenGL ES project.

2. **GLKTextureLoader**: This class makes it much easier to load images as textures to be used in OpenGL. Rather than having to write a complicated method dealing with tons of different image formats, loading a texture is now a single method call!
3. **GLKMath**: Now instead of each project having to include its own math library, GLKMath contains the most common math routines for you!
4. **GLKEffects**: These classes implement common shading behaviors used in OpenGL ES 1.0, to make transitioning to OpenGL ES 2.0 easier. They're also a handy way to get some basic lighting and texturing working.

The goal of this tutorial is to get you quickly up-to-speed with the basics of using OpenGL ES 2.0 with GLKit, from the ground up, assuming that you have no experience with OpenGL ES 2.0 or GLKit whatsoever. You will build a simple app from scratch that draws a simple cube to the screen and makes it rotate around.

In the process, you'll learn the basics of using each of these new APIs. It should be a good introduction to GLKit, whether you've already used OpenGL ES in the past, or if you're a complete beginner!

Note: This chapter is split into two parts. The first part teaches you about how `GLKView` and `GLKViewController` by building a simple GLKit project from scratch – basically a very simple version of the OpenGL game template that comes with Xcode.

If you don't care much about how the template works and want to jump straight to writing OpenGL ES code, feel free to skip to the "Creating vertex data for a simple square" section later in this chapter – there's a starter project there where you can pick up where we left off.

What is OpenGL ES?

Before you get started, let's talk a bit more about what OpenGL ES is for those of you who are unfamiliar with it.

As I mentioned earlier, OpenGL ES is the lowest level API available on iOS to let you interact with the graphics card to draw to the screen. It is most typically used for games, since games often require heavy graphics processing and effects.

If you are familiar with other game frameworks such as Cocos2D, Corona, or Unity3D, these are built on top of OpenGL ES. Programmers often prefer working with one of these higher level game frameworks instead of using OpenGL ES directly, because they are easier to learn and are often faster to write code with.

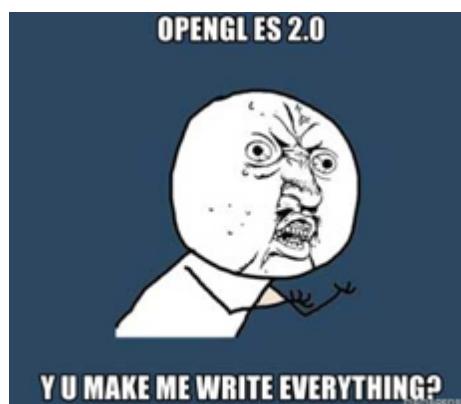
However, OpenGL ES is still good to know for several reasons:

- **It has the most power.** OpenGL ES gives you full control and power, while the other game frameworks often have limitations. If you have a game where you want to push the boundaries of what can be done, OpenGL ES may be the way to go.
- **It's a great learning experience.** Learning OpenGL ES is a great learning experience – it will teach you a lot about how graphics cards work, how game engines work, a lot about math, and more.
- **You'll be able to better understand and extend game frameworks.** Even if you decide to use a game framework, understanding OpenGL ES will help you better understand of what the game frameworks do for you. This can help you increase the performance of your games, and extend the game frameworks a low level – such as creating some special effects for Cocos2D games.
- **It's much easier than it used to be!** Thanks to the new GLKit framework in iOS that you'll learn about in this book, making OpenGL ES apps and games is much easier than it used to be. Especially with this book to guide you through, there's no reason not to get started learning OpenGL ES! ☺

The “ES” at the end of “OpenGL ES” stands for “Embedded Systems”, and is there to distinguish the API from the plain-old OpenGL (without the “ES”) used on desktops. Basically, OpenGL ES is a stripped down/simplified version of the plain-old OpenGL API designed for devices with less powerful graphics processing capabilities - such as your iPhone! If you’re familiar with plain-old OpenGL, you will notice a few differences here and there – but overall it’s fairly similar.

Finally, note that OpenGL ES is a C-based API. If you do not know C (and only know Objective-C), this chapter might be a struggle for you, as some of the code might look unfamiliar. I will not be covering the C syntax here, so if you are new to C I recommend getting a C reference book to help get you used to the syntax.

OpenGL ES 1.0 vs OpenGL ES 2.0

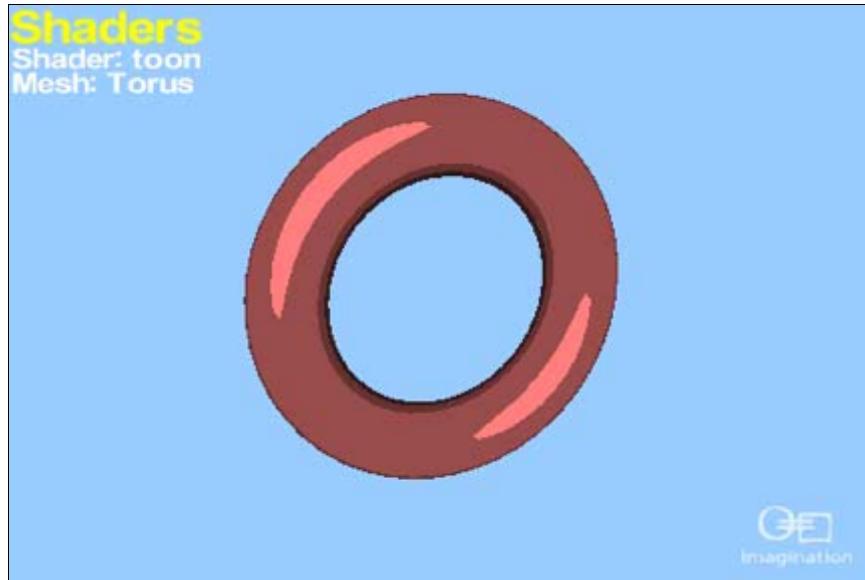


Note that this book will be focusing on OpenGL ES 2.0 – not OpenGL ES 1.0.

If you are new to OpenGL ES programming, here is the difference between OpenGL ES 1.0 and OpenGL ES 2.0:

- OpenGL ES 1.0 uses a fixed pipeline, which is a fancy way of saying you use built-in functions to set lights, vertices, colors, and more.
- OpenGL ES 2.0 uses a programmable pipeline, which is a fancy way of saying all those built-in functions go away, and you have to write everything yourself.

"OMG!" you may think , "well why would I ever want to learn OpenGL ES 2.0 then, if it's just extra work?!" Although it does add some extra work, with OpenGL ES 2.0 you can make some really cool effects that wouldn't be possible in OpenGL ES 1.0, such as this toon shader by [Imagination Technologies](#):



Or these amazing lighting and shadow effects by [Fabien Sanglard](#):



Pretty cool, eh?

OpenGL ES 2.0 is only available on the iPhone 3GS+, iPod Touch 3G+, and all iPads. But the overwhelming majority of your customers are using these devices these days, so OpenGL ES 2.0 is well worth using!

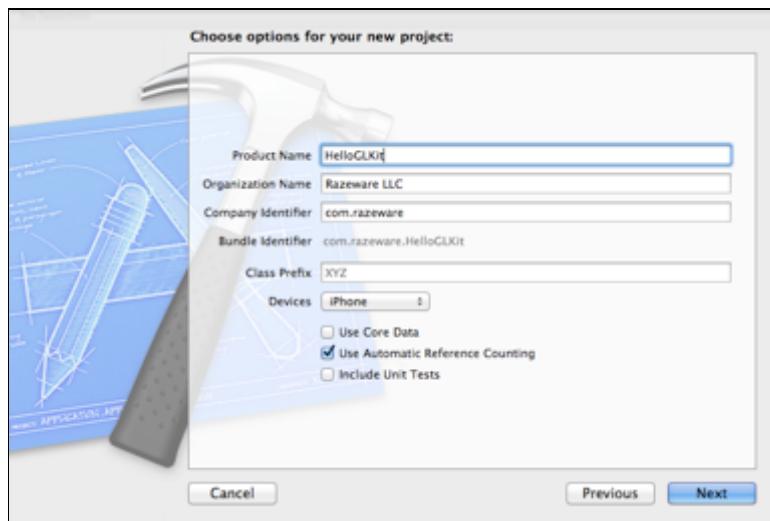
OpenGL ES 2.0 does have a bit of a higher learning curve than OpenGL ES 1.0, but now with GLKit the learning curve is much easier, because the GLKEffect and GLKMath APIs allow you to easily do a lot of the stuff that was built into OpenGL ES 1.0.

I'd say if you're new to OpenGL ES programming, it's probably best to jump straight into OpenGL ES 2.0 rather than trying to learn OpenGL ES 1.0 and then upgrading, especially now that GLKit is available. And this book will show you the basics and help you get started! ☺

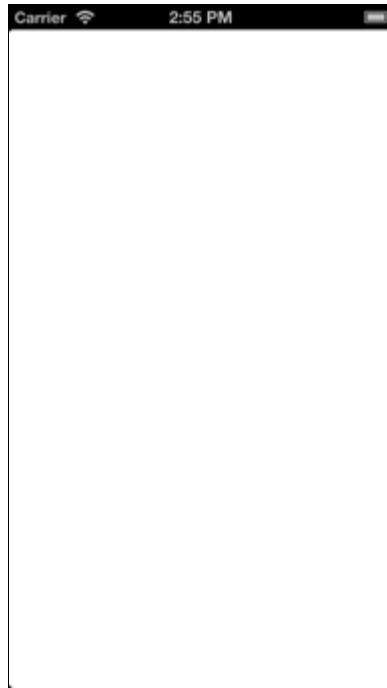
Getting started

Create a new project in Xcode with the **iOS\Application\Empty Application** template. You're choosing the Empty Application template (not the OpenGL Game template!) so you can put everything together from scratch and get a better understanding of how it works.

Set the Product Name to **HelloGLKit**, make sure Devices is set to **iPhone**, and make sure **Use Automatic Reference Counting** is selected. Click **Next**, choose a location for your project, and click **Create**.



Build and run your app, and you should see a blank Window:



The project contains almost no code at this point, but let's take a look to see how it all fits together. If you went through the Storyboard tutorial earlier in this book, this should be a good review.

Open **main.m**, and you'll see the first function that is called when the app starts up, unsurprisingly called `main`:

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

The last parameter to this method tells `UIApplication` the class to create an instance of and use as its delegate – in this case, a class called `AppDelegate`.

So let's check that out. Open **AppDelegate.m** and take a look at the method that gets called when the app starts up:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
                  [[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];
```

```
[self.window makeKeyAndVisible];
return YES;
}
```

This programmatically creates the main window for the app and makes it visible. And that's it! This is about as "from scratch" as you can get.

Introducing GLKView

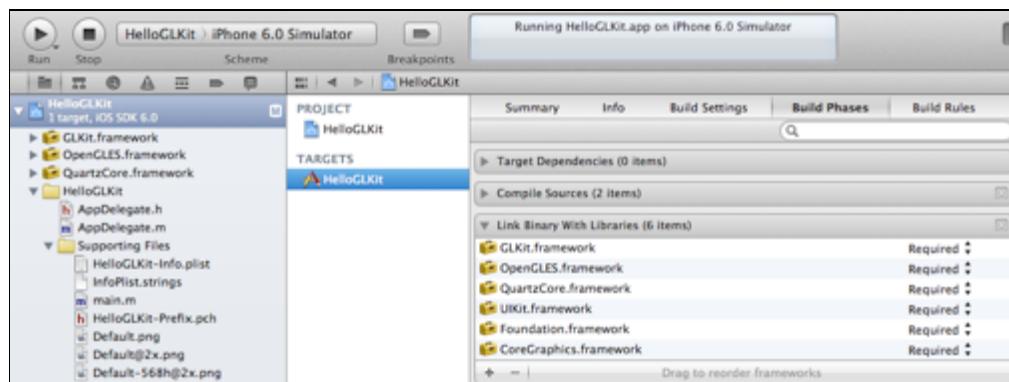
To get started with OpenGL ES 2.0, the first thing you need to do is add a new subview to the window that does its drawing with OpenGL. If you've programmed with OpenGL ES 2.0 before, you know that there used to be a ton of boilerplate code to get this working – things like creating a render buffer and frame buffer, etc.

But now it's nice and easy with a new GLKit class called `GLKView`! Whenever you want to use OpenGL ES rendering inside a view, you simply add a `GLKView` (which is a normal subclass of `UIView`) and configure a few properties on it.

You can then set a class as the `GLKView`'s delegate, and it will call a method on that class when it needs to be drawn. And that's where you can put in your OpenGL ES commands!

Let's see how this works. First things first – you need to add a few frameworks to your project to use GLKit. Select your **HelloGLKit** project in the Project Navigator, select the **HelloGLKit** target, select the **Build Phases** tab, expand the **Link Binary with Libraries** section, and click the **+** button. From the drop-down list, select the following frameworks and click **Add**:

- QuartzCore.framework
- OpenGLES.framework
- GLKit.framework



Switch to **AppDelegate.h**, and at the top of the file import the header file for GLKit as follows:

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

Next switch to **AppDelegate.m** and modify the `application:didFinishLaunchingWithOptions:` method to add a new `GLKView` as a subview of the main window:

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    self.window = [[UIWindow alloc] initWithFrame:  
        [[UIScreen mainScreen] bounds]];  
  
    EAGLContext * context = [[EAGLContext alloc]  
        initWithAPI:kEAGLRenderingAPIOpenGLES2]; // 1  
    GLKView * view = [[GLKView alloc] initWithFrame:  
        [[UIScreen mainScreen] bounds] context:context]; // 2  
    view.delegate = self; // 3  
    [self.window addSubview:view]; // 4  
  
    self.window.backgroundColor = [UIColor whiteColor];  
    [self.window makeKeyAndVisible];  
    return YES;  
}
```

The lines marked with comments are the new lines – so let's go over them one by one.

1. **Create an OpenGL context.** To do anything with OpenGL ES, you need to create an [EAGLContext](#).

An `EAGLContext` manages all of the information iOS needs to draw with OpenGL. It's similar to how you need a `CGContextRef` to do anything with Core Graphics.

When you create a context, you specify which version of the OpenGL ES API you want to use. Here, you specify that you want to use OpenGL ES 2.0. If this version of the API is not available (such as if the app was run on an iPhone 3G), the app would terminate.

2. **Create a `GLKView`.** This creates a new instance of [GLKView](#), passes in the context that was created, and makes it the full size of the window.
3. **Set the `GLKView's delegate`.** This sets the current class (`AppDelegate`) as the `GLKView's delegate`. This means whenever the view needs to be redrawn, it will call a method named `glkView:drawInRect:` on whatever classs you specify here. You will implement this shortly to contain some basic OpenGL ES commands to paint the screen green.
4. **Add the `GLKView` as a subview.** This lie adds the `GLKView` as a subview of the main window so you can see it.

Since you marked the `AppDelegate` as the `GLKView`'s delegate, you need to mark it as implementing the `GLKViewDelegate` protocol. So switch to `AppDelegate.m` and add a class extension as follows:

```
@interface AppDelegate () <GLKViewDelegate>
@end
```

If you're wondering why you added protocol declaration to the class extension rather than the header file, check out Chapter 2, "Programming in Modern Objective-C" in *iOS 6 by Tutorials*.

One step left! Add the following method to the file:

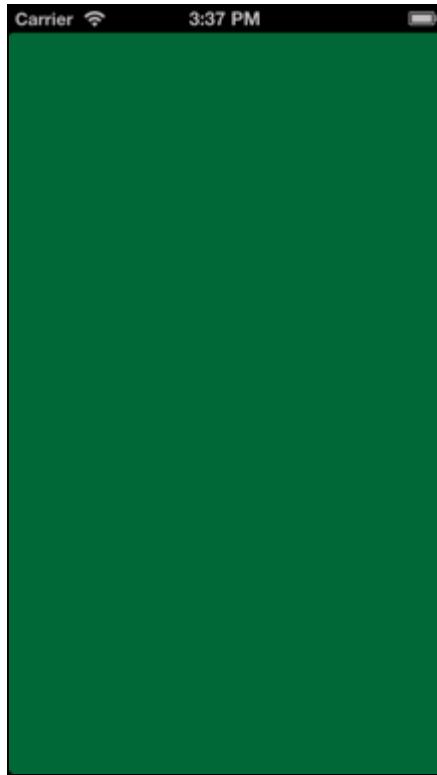
```
#pragma mark - GLKViewDelegate

- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
    glClearColor(0.0, 0.41, 0.22, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}
```

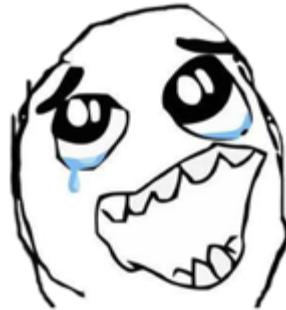
The first line calls `glClearColor`, which is an OpenGL ES function that lets you specify the RGB and alpha (transparency) values to use when clearing the screen. You set it to a particular shade of green here – does it remind you of anything? ☺

The second line calls `glClear`, another OpenGL ES function that actually clears the screen. This function allows you to clear different things which you'll learn about when you get more advanced, but for now you just need to know that you pass in `GL_COLOR_BUFFER_BIT` here, which means it should clear the current render/color buffer that displays on the screen.

That's it! Build and run, and with just 6 lines of code you have OpenGL rendering to the screen!



Those of you who are completely new to OpenGL ES might not be very impressed, but those of you used to the old painful way of doing things probably look like this guy right about now:



GLKView properties and methods

So far you've only set two properties on GLKView (`delegate` directly, and `context` indirectly), but I wanted to mention the other properties and methods on GLKView that might be useful to you in the future.

Note: This is an optional section and does not contain any code – it is mostly for the curious and for future reference. If you are a beginner or want to

continue coding the sample project, I recommend that you skip to the next section and refer back here when necessary.

Let's go through the properties and methods one group at a time:

context and delegate properties

```
@property (nonatomic, assign) IBOutlet id <GLKViewDelegate>  
delegate;  
@property (nonatomic, retain) EAGLContext *context;
```

You already learned about these in the previous section, so I won't repeat here.

drawableColorFormat property

```
@property (nonatomic) GLKViewDrawableColorFormat  
drawableColorFormat;
```

Your OpenGL ES context has a buffer it uses to store the colors that will be displayed on the screen. You can use this property to set the color format for each pixel in the buffer.

The default value is `GLKViewDrawableColorFormatRGBA8888`, which means 8 bits are used for each pixel in the buffer (so 4 bytes per pixel). This is nice because it gives you the widest possible range of colors to work with, which often makes your app look nicer.

But if your app can get away with a lower range of colors, you might want to switch this to `GLKViewDrawableColorFormatRGB565`, which makes your app consume less resources (memory and processing time).

drawableDepthFormat property

```
@property (nonatomic) GLKViewDrawableDepthFormat  
drawableDepthFormat;
```

Your OpenGL ES context can also optionally have another buffer associated with it called the depth buffer. This helps make sure that objects closer to the viewer show up in front of objects farther away.

The way it works by default is OpenGL ES keeps a buffer kind of like a render/color buffer, but instead of storing color values at each pixel, it stores the closest object to the viewer at each pixel. When it goes to draw a pixel, it checks the depth buffer to see if it's already drawn something closer to the viewer, and if so discards it. Otherwise, it adds it to the depth buffer and the color buffer.

You can set this property to choose the format of the depth buffer. The default value is `GLKViewDrawableDepthFormatNone`, which means that no depth buffer is enabled at all.

But if you want this feature (which you usually do to increase performance for 3D games), you should choose `GLKViewDrawableDepthFormat16` or `GLKViewDrawableDepthFormat24`.

The tradeoff between these two options is with `GLKDrawableDepthFormat16` your app will use less resources, but you might have rendering issues when objects are very close to each other.

drawableStencilFormat property

```
@property (nonatomic) GLKViewDrawableStencilFormat  
drawableStencilFormat;
```

Another optional buffer your OpenGL ES context can have is the stencil buffer. This helps you restrict drawing to a particular portion of the screen. It's often useful for things like shadows – for example you might use the stencil buffer to make sure the shadows are cast on the floor.

The default value for this property is `GLKViewDrawableStencilFormatNone`, which means there is no stencil buffer. However, you can enable it by setting it to the only alternative – `GLKViewDrawableStencilFormat8`.

drawableMultisample property

```
@property (nonatomic) GLKViewDrawableMultisample  
drawableMultisample;
```

The last optional buffer you can set up through a `GLKView` property is the multisampling buffer. If you ever try drawing lines with OpenGL and notice “jagged lines”, multisampling can help with this issue.

Basically what it does is instead of calling the fragment shader one time per pixel, it divides up the pixel into smaller units and calls the fragment shader multiple times at smaller levels of detail. It then merges the colors returned, which often results in a much smoother look around the edges of geometry.

Be careful about setting this value because it requires more processing time and memory for your app. The default value is `GLKViewDrawableMultisampleNone`, but you can enable it by setting it to the only alternative – `GLKViewDrawableMultisample4X`.

drawableWidth and **drawableHeight** properties

```
@property (nonatomic, readonly) NSInteger drawableWidth;  
@property (nonatomic, readonly) NSInteger drawableHeight;
```

These are read-only properties that indicate the integer height and width of your various buffers. These are based on the bounds and `contentSize` of the view – the buffers are automatically resized when these change.

snapshot method

```
- (UIImage *)snapshot;
```

This is a handy way to get a `UIImage` of what the `GLKView` is currently showing.

bindDrawable method

```
- (void)bindDrawable;
```

OpenGL ES has yet another buffer called a frame buffer, which is basically a collection of all the other buffers we talked about (color buffer, depth buffer, stencil buffer, etc).

Before your `glkView:drawInRect` is called, GLKit will bind to the frame buffer it set up for you behind the scenes. But if your game needs to change to a different frame buffer to perform some other kind of rendering (for example, if you're rendering to another texture), you can use the `bindDrawable` method to tell GLKit to re-bind back to the frame buffer it set up for you.

deleteDrawable method

```
- (void)deleteDrawable;
```

`GLKView` and OpenGL ES take a substantial amount of memory for all of these buffers. If your `GLKView` isn't visible, you might find it useful to deallocate this memory temporarily until it becomes visible again. If you want to do this, just use this method!

Next time the view is drawn, `GLKView` will automatically re-allocate the memory behind the scenes. Quite handy, eh?

enableSetNeedsDisplay property and display method

```
@property (nonatomic) BOOL enableSetNeedsDisplay;
- (void)display;
```

I don't want to spoil the surprise – I'll explain these in the next section! ☺

Updating the `GLKView`

Let's try to update what shows up in the `GLKView` periodically, like you would in a game. How about you try to make the screen pulse from red to black, kind of like a "Red Alert" effect!

Go to the top of `AppDelegate.m` and modify the `@implementation` line to add two private variable as follows:

```
@implementation AppDelegate {
    float _curRed;
```

```
    BOOL _increasing;
}
```

And initialize these in `application:didFinishLaunchingWithOptions:` before the return statement:

```
_increasing = YES;
_curRed = 0.0;
```

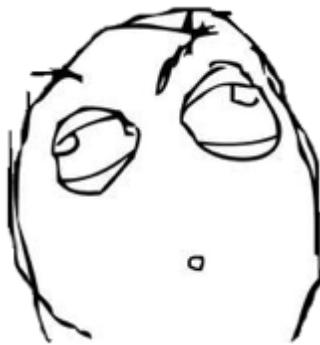
Then go to the `glkView:drawInRect:` method and update it to the following:

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
    if (_increasing) {
        _curRed += 0.01;
    } else {
        _curRed -= 0.01;
    }
    if (_curRed >= 1.0) {
        _curRed = 1.0;
        _increasing = NO;
    }
    if (_curRed <= 0.0) {
        _curRed = 0.0;
        _increasing = YES;
    }
    glClearColor(_curRed, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}
```

Every time `glkView:drawInRect:` is called, it updates the `_curRed` value a little bit based on whether it's increasing or decreasing.

Note: This code isn't perfect, because it doesn't take into effect how long it takes between calls to `glkView:drawInRect:`. This means that the animation might be faster or slower based on how quickly `glkView:drawInRect:` is called. You'll learn about a way to fix this later on in the tutorial.

Build and run and... wait a minute, nothing's happens, it's just a black screen!



This happens because by default, `GLKView` only updates itself on an as-needed basis – i.e. when views are first shown when the view's size changes, and the like. However, most of the time in game programming you need to redraw the screen multiple times per second!

So right now, `GLKView` will only redraw itself when you manually call its `setNeedsDisplay:` method – and even then, it's not synchronized with anything (such as when the screen refreshes), it's just whenever the next draw cycle happens to occur.

You can disable this behavior setting `enableSetNeedsDisplay` to `false`. Then, you can control when the redrawing occurs by calling the `display:` method at the exact point you want the screen to be refreshed.

Ideally, you would like to synchronize the time you render with OpenGL to the rate at which the screen refreshes.

Luckily, Apple provides an easy way for you to do this with the `CADisplayLink` class! This class is really easy to use, so let's just dive in and try it out. First add this import to the top of `AppDelegate.m`:

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

Then add these lines to `application:didFinishLaunchingWithOptions:` before the return statement:

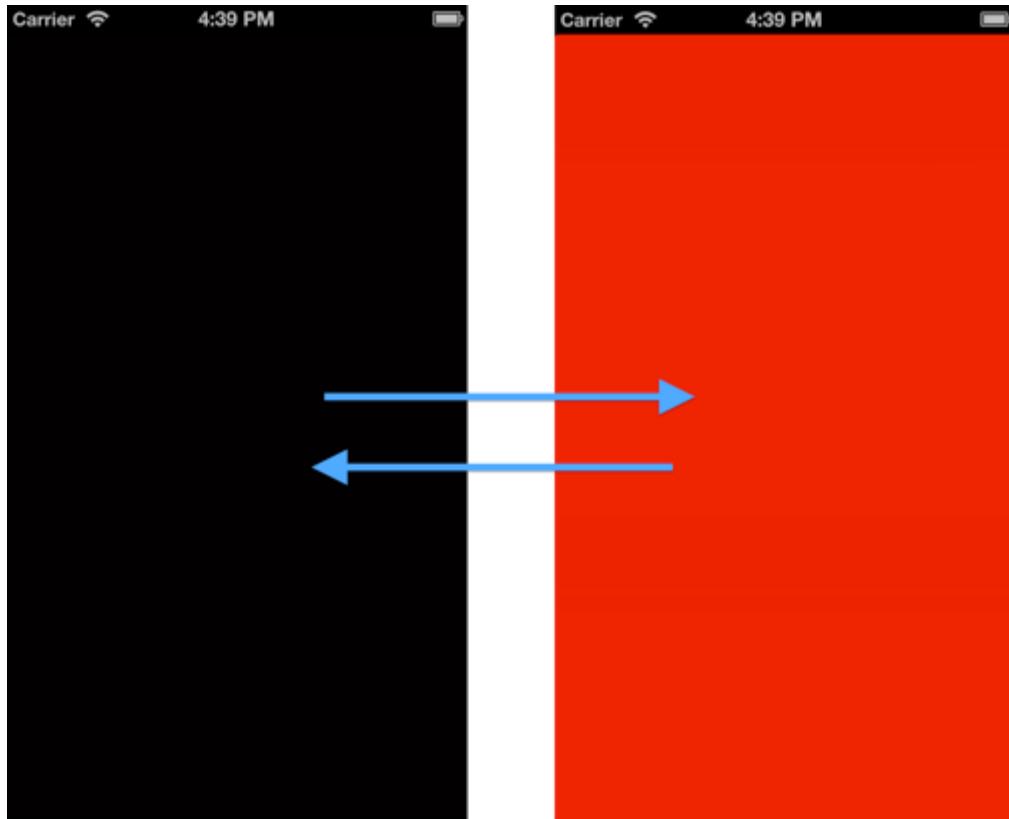
```
view.enableSetNeedsDisplay = NO;
CADisplayLink* displayLink = [CADisplayLink
    displayLinkWithTarget:self selector:@selector(render:)];
[displayLink addToRunLoop:[NSRunLoop currentRunLoop]
    forMode:NSTimerDeliveryMode];
```

This creates a `CADisplayLink` and configures it to call a method called `render:` every time the screen refreshes. Implement the `render:` method as follows:

```
- (void)render:(CADisplayLink*)displayLink {
    GLKView * view = [self.window.subviews objectAtIndex:0];
```

```
[view display];  
}
```

Build and run, and you should now see a cool pulsating “red alert” effect!



Introducing GLKViewController

You know that code you just wrote in the last section to call `display:` on the `GLKView` every frame? Well you can just forget about it, because there's a much easier way to do so by using `GLKViewController`! ☺

The reason I showed you how to do it with a plain `GLKView` first was so you understand the point behind using `GLKViewController` – it saves you from having to write that code, plus adds some extra neat features that you would have had to code yourself.

So let's try out `GLKViewController`. Modify your `application:didFinishLaunchingWithOptions:` to look like this:

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    self.window = [[UIWindow alloc] initWithFrame:
```

```
[[UIScreen mainScreen] bounds]];
```

```
EAGLContext * context = [[EAGLContext alloc]
    initWithAPI:kEAGLRenderingAPIOpenGLGLES2];
GLKView * view = [[GLKView alloc] initWithFrame:
    [[UIScreen mainScreen] bounds] context:context];
view.delegate = self;
// [self.window addSubview:view];
```

```
_increasing = YES;
_curRed = 0.0;
```

```
//view.enableSetNeedsDisplay = NO;
//CADisplayLink* displayLink = [CADisplayLink
displayLinkWithTarget:self selector:@selector(render:)];
// [displayLink addToRunLoop:[NSRunLoop currentRunLoop]
forMode:NSTimerRunLoopMode];
```

```
GLKViewController * viewController =
    [[GLKViewController alloc]
        initWithNibName:nil bundle:nil]; // 1
viewController.view = view; // 2
viewController.delegate = self; // 3
viewController.preferredFramesPerSecond = 60; // 4
self.window.rootViewController = viewController; // 5
```

```
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

Feel free to delete the commented lines – I just commented them out so you could easily see what is no longer needed. There are also four new lines (marked with comments):

1. **Create a `GLKViewController`.** This creates a new instance of `GLKViewController` programmatically. In this case, it has no XIB associated.
2. **Set the `GLKViewController's view`.** The root view of a `GLKViewController` should be a `GLKView`, so you set it to the one you just created.
3. **Set the `GLKViewController's delegate`.** Here you set the current class (`AppDelegate`) as the delegate of `GLKViewController`. This means that the `GLKViewController` will call a method on the `AppDelegate` each frame where you can add game logic, and another method when the game pauses (a nice built-in feature of `GLKViewController` which you'll learn about later).

4. Set the preferred FPS. `GLKViewController` will call your draw method a certain number of times per second. This number gives a hint to the `GLKViewController` how often you'd like to be called. Of course, if your game takes a longer time to render frames, the actual number may be lower than this.

The default value is 30 FPS. Apple's guidelines are to set this to whatever your app can reliably support so that the frame rate is consistent and doesn't seem to stutter. This app is very simple and can easily run at 60 FPS, so we set it to that.

Also, as an FYI, if you want to see the actual number of times the OS will attempt to call your update/draw methods, check the read-only property `framesPerSecond`.

5. Set the rootViewController. You want the `GLKViewController` to be the first thing that shows up, so here you add it as the `rootViewController` of the window. Note that you no longer need to add the view as a subview of the window manually, because you've set it as the root view of the `GLKViewController`.

At this point, you no longer need the code to run the render loop and tell the `GLKView` to refresh each frame – `GLKViewController` automatically does that for you in the background! So go ahead and delete the `render:` method now.

Also, remember you set the `GLKViewController`'s delegate to the current class (`AppDelegate`), so you should mark the class as implementing the `GLKViewControllerDelegate`. Modify the class extension at the top of **AppDelegate.m** to the following:

```
@interface AppDelegate () <GLKViewControllerDelegate,
GLKViewControllerDelegate>
@end
```

As a final step, update `glkView:drawInRect:` and add the new `glkViewControllerUpdate:` method as follows:

```
#pragma mark - GLKViewDelegate

- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
    glClearColor(_curRed, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}

#pragma mark - GLKViewControllerDelegate

- (void)glkViewControllerUpdate:(GLKViewController *)controller {
    if (_increasing) {
        _curRed += 1.0 * controller.timeSinceLastUpdate;
    } else {
        _curRed -= 1.0 * controller.timeSinceLastUpdate;
    }
    if (_curRed >= 1.0) {
```

```
    _curRed = 1.0;
    _increasing = NO;
}
if (_curRed <= 0.0) {
    _curRed = 0.0;
    _increasing = YES;
}
}
```

Note that you moved the code that changes the current color from the old `glkView:drawInRect:` method (intended for code to draw the current state only) to the new `glkViewControllerUpdate:` method (intended to update game/app logic).

Also notice you changed the amount the red color increments from a hardcoded value to a calculated value, based on the amount of time since the last update. This is nice because it guarantees the animation will always proceed at the same speed, regardless of the frame rate.

Storing the time since the last update is another of those convenient things that `GLKViewController` does for you! You didn't have to write any special code to keep track of that – it was done automatically. There are some other time-based properties you might find useful as well that you'll learn about later in this chapter.

GLKViewController and storyboards

So far, you've created a `GLKViewController` and a `GLKView` manually because it was a simple way to introduce you to how they work. But you probably wouldn't want to do things this way in a real app – it's much better to leverage the power of Storyboards, so you can include this view controller anywhere you want in your app's flow!

For example, you might have a game where you make the main menu with standard UIKit controls, and the game itself is an OpenGL ES game in a `GLKViewController/GLKView`.

So let's do a little refactoring to work with this more typical workflow. Let's start by creating a subclass of `GLKViewController` that you can use to contain your app's logic.

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template, set the class to **HelloGLKitViewController**, and make it a subclass of **GLKViewController**. Make sure both Targeted for iPad and With XIB for user interface are both unselected, and finish creating the file.

Open up **HelloGLKitViewController.m**, and add a few private instance variables:

```
@implementation HelloGLKitViewController {
```

```
    float _curRed;
    BOOL _increasing;
    EAGLContext * _context;
}
```

Then implement `viewDidLoad`, `dealloc`, and `didReceiveMemoryWarning` as the following:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    _context = [[EAGLContext alloc]
        initWithAPI:kEAGLRenderingAPIOpenGLGLES2];
    if (!_context) {
        NSLog(@"Failed to create ES context");
    }
    GLKView *view = (GLKView *)self.view;
    view.context = _context;
}

- (void)cleanup {
    if ([EAGLContext currentContext] == _context) {
        [EAGLContext setCurrentContext:nil];
    }
    _context = nil;
}

- (void)dealloc {
    [self cleanup];
}

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

In `viewDidLoad`, you create an OpenGL ES 2.0 context (the same as you did last time in `AppDelegate`) and squirrel it away. Your root view is a `GLKView` (you know this because you will set it up this way in the Storyboard editor), so you cast it as one. You then set its context to the OpenGL ES context you just created.

Note that you don't have to set the view controller as the view's delegate – `GLKViewController` does this automatically behind the scenes.

When the view controller is deallocated, you have to handle the case where OpenGL ES's current context is the context you've created for this view controller. You do this in the `cleanup` method - if they are the same, you need to set OpenGL ES's current context to nil.

You should also perform cleanup upon a memory warning and the view isn't currently visible, since it's important to free as much memory as possible upon a memory warning.

Note: In the old days, iOS would automatically unload your view controller's view for you upon a memory warning and call `viewDidUnload` when this occurs. This no longer happens in iOS 6 – check out Chapter 20, "What's New with Cocoa Touch" in *iOS 6 by Tutorials* for more information.

Next, add the implementations for the `glkView:drawInRect:` and `update` callbacks, similar to before:

```
#pragma mark - GLKViewDelegate

- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
    glClearColor(_curRed, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}

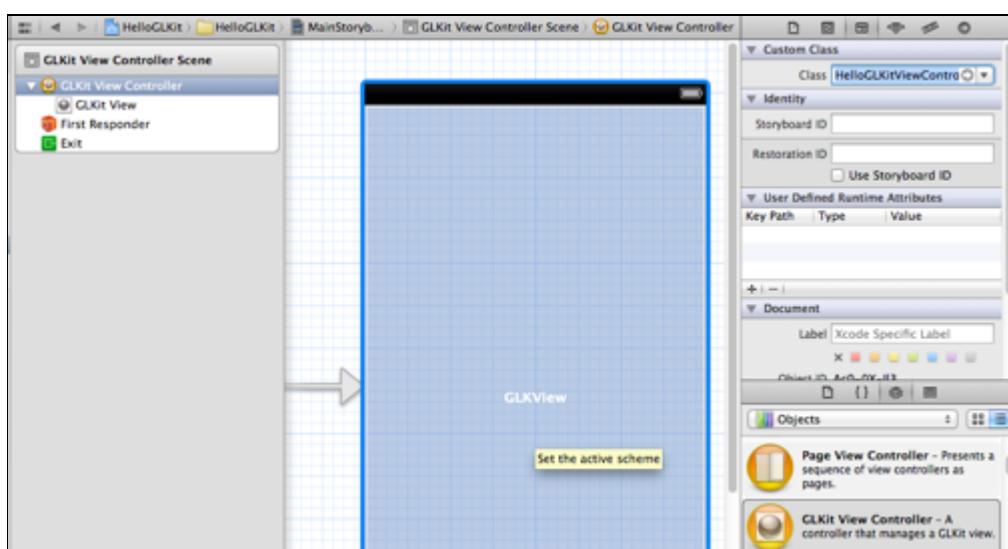
#pragma mark - GLKViewControllerDelegate

- (void)update {
    if (_increasing) {
        _curRed += 1.0 * self.timeSinceLastUpdate;
    } else {
        _curRed -= 1.0 * self.timeSinceLastUpdate;
    }
    if (_curRed >= 1.0) {
        _curRed = 1.0;
        _increasing = NO;
    }
    if (_curRed <= 0.0) {
        _curRed = 0.0;
        _increasing = YES;
    }
}
```

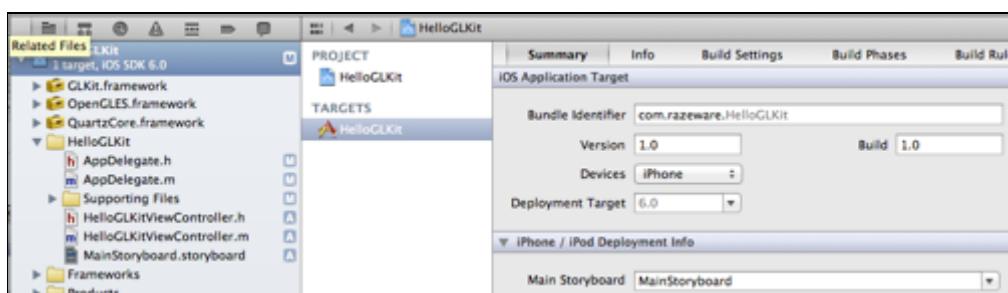
Note that the update method is called plain `update` (instead of `glkViewControllerUpdate`), because now that you're subclassing `GLKViewController` you can just override one of its methods instead of having to set a delegate. Also, the `timeSinceLastUpdate` property is accessed via `self`, not a passed in view controller.

Now that your `GLKViewController` subclass is in place, let's create the Storyboard. Create a new file with the **iOS\User Interface\Storyboard** template, choose **iPhone** for the device family, and save it as **MainStoryboard.storyboard**.

Open **MainStoryboard.storyboard**, and from the Objects panel drag a **GLKit View Controller** into the grid area. Select the **GLKit View Controller** in the left sidebar, and in the Identity Inspector set the class to **HelloGLKitViewController**:



To make this Storyboard run on startup, select **HelloGLKit** in the Project Navigator and select the **Summary** tab. Under the **iPhone / iPod Deployment Info** section, set the **Main Storyboard** to **MainStoryboard**.



That pretty much completes everything you need, but you still have some old code in `AppDelegate` that needs to be cleaned up. Luckily this is incredibly simple, just delete everything in **AppDelegate.m** and replace it with the following:

```
#import "AppDelegate.h"
```

```
@implementation AppDelegate  
  
- (BOOL)application:(UIApplication *)application  
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    return YES;  
}  
  
@end
```

That's it! Build and run your app, and you'll see the "red alert" effect working as usual.

At this point, you're getting pretty close to the setup you get when you choose the OpenGL Game template with the Storyboard option set. The difference is that template has a bunch of other code in there, which you can delete if you don't need it to get back to this "fresh and simple" starting point.

Feel free to use the OpenGL game template in the future to save a little time – but now you know how `GLKView` and `GLKViewController` works from the ground up!

GLKViewController and pausing

Before we move on to covering rendering some geometry with OpenGL ES, let's discuss one last feature of `GLKViewController` – pausing.

To see how it works, add the following to `HelloGLKitViewController.m`:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {  
    self.paused = !self.paused;  
}
```

Build and run the app, and now whenever you tap the animation stops or resumes! Behind the scenes, `GLKViewController` stops calling your update method and your draw method. This is a really handy way to implement a pause button in your game.

In addition to that, `GLKViewController` has a `pauseOnResignActive` property that is set to `YES` by default. This means when the user hits the home button or receives an interruption such as a phone call, your app will automatically be paused! Similarly, it has a `resumeOnDidBecomeActive` property that is also set to `YES` by default, which means that when the user comes back to your app, it will automatically unpause. Handy, that!

You've learned about almost every property on `GLKViewController` by now, except for the extra time info properties I mentioned earlier:

- **timeSinceLastDraw** gives you the elapsed time since the last call to the draw method. Note this might be different than timeSinceLastUpdate, since your update method takes time!
- **timeSinceFirstResume** gives you the elapsed time since the first time GLKViewController resumed sending updates. This often means the time since your app launched, if your GLKViewController is the first thing that shows up.
- **timeSinceLastResume** gives you the elapsed time since the last time GLKViewController resumed sending updates. This often means the time since your game was unpause.

Let's add some code to try these out. Add the following code to the top of touchesBegan:withEvent::

```
NSLog(@"%@", @"timeSinceLastUpdate: %f", self.timeSinceLastUpdate);
NSLog(@"%@", @"timeSinceLastDraw: %f", self.timeSinceLastDraw);
NSLog(@"%@", @"timeSinceFirstResume: %f", self.timeSinceFirstResume);
NSLog(@"%@", @"timeSinceLastResume: %f", self.timeSinceLastResume);
```

Build and run, and check the console as you play around with the app a bit so you can get familiar with how these work. Then it's time for more fun stuff – rendering a square to the screen!

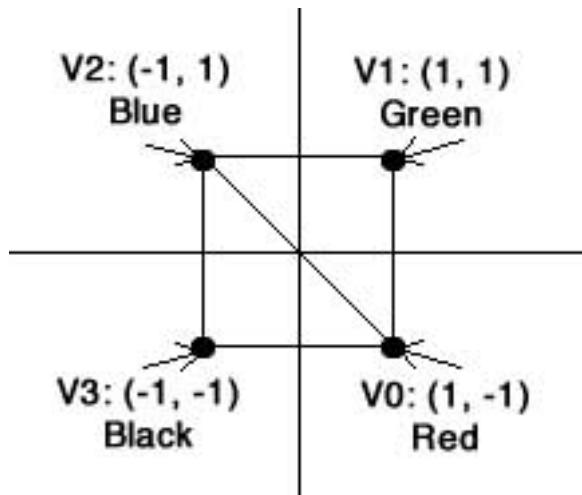
Creating vertex data for a simple square

Note: If you skipped to this section from the beginning of this tutorial, you can pick up with the **HelloGLKitPart1** starter project, which contains all of the code developed up to this point. It is basically a stripped-down version of the Hello GLKit project template that comes with Xcode.

Now that you have a good understanding of how `GLKView` and `GLKViewController` work, you'll do something more interesting than a flashing screen – you'll render a simple square to the screen!

To do this, you need to know the points for the four corners of the square. It's common to call a point in OpenGL ES space a vertex, and a set of points vertices.

The square will be set up like the following:



When you render geometry with OpenGL ES, it can't render squares – it can only render triangles. However, you can create a square with two triangles, as you can see in the picture above: one triangle with vertices $(0, 1, 2)$ and one triangle with vertices $(2, 3, 0)$.

One of the nice things about OpenGL ES 2.0 is you can keep your vertex data organized in whatever manner you like, rather than having predefined structures like in OpenGL ES 1.0. Open up **HelloGLKitViewController.m** and create a plain old C-structure and a few arrays to keep track of the information, as shown below:

```
typedef struct {
    float Position[3];
    float Color[4];
} Vertex;

const Vertex Vertices[] = {
    {{1, -1, 0}, {1, 0, 0, 1}},
    {{1, 1, 0}, {0, 1, 0, 1}},
    {{-1, 1, 0}, {0, 0, 1, 1}},
    {{-1, -1, 0}, {0, 0, 0, 1}}
};

const GLubyte Indices[] = {
    0, 1, 2,
    2, 3, 0
};
```

The above code creates:

- a structure to keep track of your per-vertex information (currently just color and position). For those of you who are unfamiliar with C syntax, this defines a structure (think of this as a class with no methods) with an array of 3 floats for the position of the vertex, and an array of 4 floats for the color of the vertex.

- an array with all the info for each vertex. For those of you who are unfamiliar with C syntax, this defines and initializes an array of 4 Vertex structures. The first element in the array has a position of {1, -1, 0} (x=1, y=-1, z=0) and a color of {1, 0, 0, 1} (red=1, green=0, blue=0, alpha=1). The four entries in this array represent the four corners in the square, as shown in the diagram earlier.
- an array that gives a list of triangles to create, by specifying the indices of the 3 vertices that make up each triangle.

You now have all the information you need, you just need to pass it to OpenGL!

Creating vertex buffer objects

The best way to send data to OpenGL is through something called vertex buffer objects.

Basically these are OpenGL objects that store buffers of vertex data for you. You use a few function calls to send your data over to OpenGL-land.

There are two types of vertex buffer objects: one to keep track of the per-vertex data (like you have in the Vertices array), and one to keep track of the indices that make up triangles (like you have in the Indices array).

So first add the following private instance variables to your class:

```
GLuint _vertexBuffer;  
GLuint _indexBuffer;
```

Then add a method to create these:

```
- (void)setupGL {  
    [EAGLContext setCurrentContext:_context];  
  
    glGenBuffers(1, &_vertexBuffer);  
    glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);  
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices),  
                 Vertices, GL_STATIC_DRAW);  
  
    glGenBuffers(1, &_indexBuffer);  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBuffer);  
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices),  
                 Indices, GL_STATIC_DRAW);  
}
```

The first thing this does is set the current OpenGL ES context to the current context. This is important in case some other code has changed the global context.

It then calls `glGenBuffers` to create a new vertex buffer object. Remember that you need a vertex buffer to store all of the vertices for the square so OpenGL can draw the square for you.

Note: If you are unfamiliar with C, this syntax might look different to you. To explain the syntax, this is calling a C function called `glGenBuffers` with the first parameter set to 1, and the second parameter set to a pointer to the vertex buffer variable. The ampersand converts a regular variable to a pointer to that variable.

In summary, this line of code is saying “create a new buffer, and store the result in the vertex buffer variable.”

Next it calls `glBindBuffer` to make `vertexBuffer` the active buffer to use for future commands using the `GL_ARRAY_BUFFER` parameter. You need to do this because you’re about to tell OpenGL to do something with the vertex buffer you created.

Finally, it calls `glBufferData` to send the data over to OpenGL-land. Although you have defined your vertices earlier in your own array, this memory is on the CPU, not in the graphics card. So you need to call this function to move the data to the graphics card.

Also before you forget, call this method at the bottom of `viewDidLoad`:

```
[self setupGL];
```

And add this to the beginning of your `cleanup` method:

```
[EAGLContext setCurrentContext:_context];
glDeleteBuffers(1, &_vertexBuffer);
glDeleteBuffers(1, &_indexBuffer);
```

This makes sure that you free the memory for the buffers you created when necessary.

You’re almost done – you just need to add the code to render the geometry to the screen with `GLKBaseEffect`!

Introducing `GLKBaseEffect`

In OpenGL ES 2.0, to render any geometry to the scene, you have to create two tiny little programs called shaders.

Shaders are written in a C-like language called GLSL. Don’t worry too much about studying up on the reference at this point – you don’t even need to know GLSL for this book, for reasons you’ll see shortly!

There are two types of shaders:

- **Vertex shaders** are programs that get called once per vertex in your scene. So if you are rendering a simple scene with a single square, with one vertex at each corner, this would be called four times. Its job is to perform some calculations such as lighting, geometry transforms, etc., figure out the final position of the vertex, and also pass on some data to the fragment shader.
- **Fragment shaders** are programs that get called once per pixel (sort of) in your scene. So if you're rendering that same simple scene with a single square, it will be called once for each pixel that the square covers. Fragment shaders can also perform lighting calculations, etc, but their most important job is to set the final color for each pixel.

`GLKBaseEffect` is a helper class that implements some common shaders for you. The goal of `GLKBaseEffect` is to provide most of the functionality available in OpenGL ES 1.0, to make porting apps from OpenGL ES 1.0 to OpenGL ES 2.0 easier.

To use a `GLKBaseEffect`, you do the following:

1. **Create a `GLKBaseEffect`.** Usually you create one of these when you create your OpenGL ES context. You can (and should) re-use the same `GLKBaseEffect` for different geometry, and just reset the properties. Behind the scenes, `GLKBaseEffect` will only propagate the properties that have changed to its shaders.
2. **Set `GLKBaseEffect` properties.** Here you can configure the lighting, transform, and other properties that the `GLKBaseEffect`'s shaders will use to render the geometry.
3. **Call `prepareToDraw` on the `GLKBaseEffect`.** Any time you change a property on the `GLKBaseEffect`, you need to call `prepareToDraw` prior to drawing to get the shaders set up properly. This also enables the `GLKBaseEffect`'s shaders as the current shader program.
4. **Enable pre-defined attributes.** When you make your own shaders, you set them up to take parameters called attributes, and you have to write code to get IDs that correspond to each attribute so you can set them in code. For `GLKBaseEffect`'s built in shaders, there are predefined constants for their attribute IDs such as `GLKVertexAttribPosition` or `GLKVertexAttribColor`. Your job is to enable any parameters that you want to pass in to the shaders, and then pass pointers to your data for each parameter.
5. **Draw your geometry.** Once you have everything set up, you can use normal OpenGL ES draw commands such as `glDrawArrays` or `glDrawElements`, and it will be rendered using the effect you've set up!

The nice thing about `GLKBaseEffect` is that if you use it, you don't necessarily have to write any shaders at all! Of course you're welcome to if you'd like – and you can mix and match, rendering some things with `GLKBaseEffect`, and some with your own shaders. If you look at the OpenGL Game template project, you'll see an example of exactly that!

In this book, you’re going to focus on just using `GLKBaseEffect`, since the entire point is to get you up to speed with the new GLKit functionality – plus it’s plain easier!

So let’s walk through the steps one at a time in code.

1. Create a `GLKBaseEffect`.

The first step is to simply create a `GLKBaseEffect`. In `HelloGLKitViewController.m`, add a private instance variable for a `GLKBaseEffect`:

```
GLKBaseEffect * _effect;
```

Then in `setupGL`, initialize it at the end of the method:

```
_effect = [[GLKBaseEffect alloc] init];
```

And set it to nil at the end of `cleanup`:

```
_effect = nil;
```

Now that you’ve created the effect, you’ll use it in conjunction with your vertex and index buffers to render the square. The first step is to use your effect’s projection matrix!

2. Set `GLKBaseEffect` properties.

The first property you need to set is the projection matrix. A projection matrix is how you tell the CPU to render 3D geometry onto a 2D plane. Think of it as drawing a bunch of lines out from your eye through each pixel in your screen. Whatever the frontmost 3D object each line hits determines the pixel that is drawn to the screen.

GLKit provides you with some handy functions to set up a projection matrix. The one you’re going to use allows you to specify the field of view along the y-axis, the aspect ratio, and the near and far planes:

The field of view is similar to different types of camera lenses. A small field of view (for example 10) is like a telephoto lens – it magnifies images by “pulling” them closer to you. A large field of view (for example 100) is like a wide angle lens – it makes everything seem farther away. A typical value to use for this is around 65-75.

The aspect ratio is the aspect ratio you want to render to (in this case, the aspect ratio of the view). It uses this in combination with the field of view (which is for the y-axis) to determine the field of view along the x-axis.

The near and far planes are the bounding boxes for the “viewable” volume in the scene. So if something is closer to the eye than the near plane or further away from the far plane, it won’t be rendered. This is a common problem to run into – you try

and render something and it doesn't show up. One thing to check is that it's actually between the near and far planes.

Let's try this out – add the following code to the bottom of `update`:

```
float aspect =
fabsf(self.view.bounds.size.width /
self.view.bounds.size.height);

GLKMatrix4 projectionMatrix = GLKMatrix4MakePerspective(
    GLKMathDegreesToRadians(65.0f), aspect, 4.0f, 10.0f);

_effect.transform.projectionMatrix = projectionMatrix;
```

In the first line, you get the aspect ratio of the `GLKView`.

In the second line, you use a built in helper function in the GLKit math library to easily create a perspective matrix – all you have to do is pass in the parameters discussed above. You set the near plane to 4 units away from the eye, and the far plane to 10 units away.

In the third line, you simply set the `GLKEffect`'s projection matrix to the result!

You need to set one more property now – the `modelViewMatrix`. The `modelViewMatrix` is the transform that is applied to any geometry that the effect renders.

The GLKit math library again comes to the rescue here with some really handy functions that make performing translations, rotations, and scales easy – even if you don't know much about matrix math. To see what I mean, add the following lines to the bottom of `update`:

```
GLKMatrix4 modelViewMatrix =
    GLKMatrix4MakeTranslation(0.0f, 0.0f, -6.0f);
_rotation += 90 * self.timeSinceLastUpdate;
modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix,
    GLKMathDegreesToRadians(_rotation), 0, 0, 1);
_effect.transform.modelviewMatrix = modelViewMatrix;
```

If you remember back to where you set up the vertices for the square, remember that the z-coordinate for each vertex was 0. If you tried to render it with this perspective matrix, it wouldn't show up because it's closer to the eye than the near plane!

So the first thing you need to do is move the square backwards. So in this first line, you use the `GLKMatrix4MakeTranslation` function to create a matrix you'll use to translate the square 6 units backwards.

Next, you'll make the square rotate for fun. You increment an instance variable that keeps track of the current rotation (which you'll add in a second), and use the `GLKMatrix4Rotate` method to modify the current transformation by rotating it as well. It takes radians, so you use the `GLKMathDegreesToRadians` method to perform the conversion. Yes, this math library has just about every matrix and vector math routine you'll need!

Finally, you set the `GLKEffect`'s `transform` property to the result.

Before you forget, add the rotation private instance variable:

```
float _rotation;
```

You'll play around with more of the `GLKBaseEffect` properties in the next chapter, since there's a lot of cool stuff and you've barely scratched the surface here. But let's continue on for now, so you can finally get something rendering!

3. Call `prepareToDraw` on the `GLKBaseEffect`.

For this step, add the following line to the bottom of `glkView:drawInRect::`

```
[_effect prepareToDraw];
```

w00t, that was easy! Just remember that you need to call this after any time you change properties on a `GLKBaseEffect`, before you draw with it, or the changes in the properties won't take effect.

4. Enable pre-defined attributes.

Next add this code to the bottom of `glkView:drawInRect::`

```
glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBuffer);
glEnableVertexAttribArray(GLKVertexAttribPosition);
glVertexAttribPointer(GLKVertexAttribPosition, 3, GL_FLOAT,
    GL_FALSE, sizeof(Vertex),
    (const GLvoid *) offsetof(Vertex, Position));
glEnableVertexAttribArray(GLKVertexAttribColor);
glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,
    GL_FALSE, sizeof(Vertex),
    (const GLvoid *) offsetof(Vertex, Color));
```

If you haven't programmed in OpenGL ES 2.0 before this code might look pretty confusing to you, so let me explain.

Every time before you draw, you have to tell OpenGL which vertex buffer objects you should use. So here you bind (in other words, "select") the vertex and index buffers you created earlier. Strictly, you don't have to do this for this app (because

they're already still bound from before), but usually you have to do this because in most games and apps you use many different vertex buffer objects.

Next, you have to enable the pre-defined vertex attributes you want the `GLKBaseEffect` to use. You use the `glkEnableVertexAttribArray` function to enable two attributes here – one for the vertex position, and one for the vertex color. GLKit has predefined constants you need to use for these – `GLKVertexAttribPosition` and `GLKVertexAttribColor`.

Next, you call `glVertexAttribPointer` to feed the correct values to these two input variables for the vertex shader.

This is a particularly important function so let's go over how it works carefully.

- The first parameter specifies the attribute name to set. You just use the predefined constants GLKit set up.
- The second parameter specifies how many values are present for each vertex. If you look back up at the `Vertex` struct, you'll see that for the position there are three floats (`x,y,z`) and for the color there are four floats (`r,g,b,a`).
- The third parameter specifies the type of each value – which is `float` for both Position and Color.
- The fourth parameter is always set to `false`.
- The fifth parameter is the size of the stride, which is a fancy way of saying “the size of the data structure containing the per-vertex data.” So you can simply pass `in sizeof(Vertex)` here to get the compiler to compute it for you.
- The final parameter is the offset within the structure to find this data. You can use the handy `offsetof` operator to find the offset of a particular field within a structure.

So now that you're passing in the position and color data to the `GLKBaseEffect`, there's only one step left...

5. Draw your geometry.

To draw the geometry, add this to the bottom of `glkView:drawInRect::`:

```
glDrawElements(GL_TRIANGLES, sizeof(Indices)/sizeof(Indices[0]),
               GL_UNSIGNED_BYTE, 0);
```

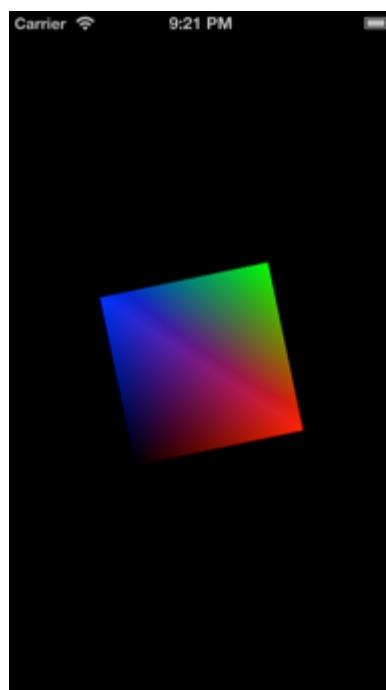
This is also an important function so let's discuss each parameter here as well.

- The first parameter specifies the manner of drawing the vertices. You specify `GL_TRIANGLES` here, which means the index buffer lists triangles one at a time, with three vertices per triangle.

There are different options you may come across in other tutorials like `GL_LINE_STRIP` or `GL_TRIANGLE_FAN`, but `GL_TRIANGLES` is the most generically useful (especially when combined with vertex buffer objects), so it's what we cover here.

- The second parameter is the count of vertices to render. You use a C trick to compute the number of elements in an array here by dividing the `sizeof(Indices)` (which gives you the size of the array in bytes) by `sizeof(Indices[0])` (which gives you the size of the first element in the array).
- The third parameter is the data type of each individual index in the `Indices` array. You're using an unsigned byte for that, so you specify that here.
- From the documentation, it appears that the final buffer should be a pointer to the indices. But since you're using vertex buffer objects it's a special case – it will use the `indices` array you already passed to OpenGL-land in the `GL_ELEMENT_ARRAY_BUFFER`.

Guess what? You're done! Compile and run the app and you should see a pretty rotating square on the screen!



Where To Go From Here?

At this point, you have hands-on experience making a simple GLKit based app with OpenGL ES 2.0 – completely from scratch!

If you're new to OpenGL ES 2.0, you've also gotten a great grounding on some of the most important techniques, such as vertex buffers, index buffers, and vertex attributes.

However, there's more cool stuff in store for you with GLKit. Keep reading for the next chapter, where you'll move to full 3D, and demonstrate some of the cool effects you can get with `GLKBaseEffect`, such as lighting, fog, and more!

Chapter 9: Intermediate OpenGL ES 2.0 with GLKit

By Ray Wenderlich

In the previous chapter, you learned the basics of using OpenGL ES 2.0 with GLKit and created a simple app with a rotating square onto the screen.

In the process, you learned a great deal about `GLKView` and `GLKViewController` and the basics of using `GLKEffect`.

In this chapter, you're going to take things to full 3D and convert your square into a rotating 3D cube! In addition, you'll learn a lot more about the cool things you can do with `GLKBaseEffect`, such as lighting effects, multitexturing, and fog effects.

By the time you're done, you'll have hands-on experience with the four main areas of GLKit and be ready to continue your studies of OpenGL ES 2.0.

Gratuitous vertex array objects

Note: When I first wrote this chapter, I included this section on vertex array objects because Apple was really pushing using these, and I thought it would improve performance of your app by using them. However, after some testing I found that they don't actually improve performance in practice.

After some investigation, I found this great blog post by Daniel Pasco that explains the subject in more detail: <http://blackpixel.com/blog/399/iphone-vertex-buffer-object-performance/>

In this updated version of the book, I decided to keep this section anyway since they are still a useful way to keep your code nice and clean, and you might see them in other people's code even if you aren't personally using them. But do note that it might not increase your performance much by using them.

In the last chapter, you rendered your square with vertex buffer objects and index buffer objects, and bound to these objects each time you wanted to draw. This works, but isn't ideal.

If you were drawing a lot of different geometry to the scene, your code would get quite tedious. Every time you want to draw something, you'd have to bind the correct vertex and index buffers, enable the attributes you want for the shader, and specify where their data is located.

In theory, continually making all these calls each time is also slow - however this doesn't actually turn out to be the case on iOS devices, as explained in the note above.

Apple's recommendation to solve these issues is to use a technique called vertex array objects. Vertex array objects let you configure all this stuff once, and load back your settings when you're about to draw. They're pretty easy to use too, so let's try them out.

First add a new private instance variable to **HelloGLKitViewController.m**:

```
GLuint _vertexArray;
```

Then modify your `setupGL` method as follows:

```
- (void)setupGL {

    // Old stuff
    [EAGLContext setCurrentContext:_context];

    // New lines
    glGenVertexArraysOES(1, &_vertexArray);
    glBindVertexArrayOES(_vertexArray);

    // Old stuff
    glGenBuffers(1, &_vertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices),
                 Vertices, GL_STATIC_DRAW);

    glGenBuffers(1, &_indexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices),
                 Indices, GL_STATIC_DRAW);

    // New lines (were previously in glkView:drawInRect:
    glEnableVertexAttribArray(GLKVertexAttribPosition);
    glVertexAttribPointer(GLKVertexAttribPosition, 3, GL_FLOAT,
                         GL_FALSE, sizeof(Vertex),
```

```

    (const GLvoid *) offsetof(Vertex, Position));
glEnableVertexAttribArray(GLKVertexAttribColor);
glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,
    GL_FALSE, sizeof(Vertex),
    (const GLvoid *) offsetof(Vertex, Color));

// New line
glBindVertexArrayOES(0);

// Old stuff
_effect = [[GLKBaseEffect alloc] init];

}

```

The first group of new lines creates a new vertex array object and binds to it. After doing that, the rest of the setup calls you make will be associated with the new vertex array object.

After setting up the vertex and index buffer, you add the lines of code to set up the vertex attributes that were previously in draw. This configuration will also be associated with the new vertex array object.

Finally, you bind the current vertex array object to 0 since you're done with it.

Next, replace your `glkView:drawInRect:` method with this:

```

- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
    glClearColor(_curRed, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);

    [_effect prepareToDraw];

    glBindVertexArrayOES(_vertexArray);
    glDrawElements(GL_TRIANGLES,
        sizeof(Indices)/sizeof(Indices[0]), GL_UNSIGNED_BYTE, 0);
}

```

Notice how much simpler it is! Now your code is much easier to understand, especially if you had lots of different sets of vertices/indices/attributes you are rendering, so you should get into the habit of using these.

Build and run your app, and it should work as usual!

Gratuitous textures

There's one big aspect of GLKit that you haven't played around with yet – texture loading!

This was one of the new features in iOS 5 that I was most excited about, because the code to do this used to be horrendous, especially when you wanted to support a wide variety of image formats.

But now it's a lot easier! Let's dive right in and try it out.

In the resources for this chapter, you'll find an image called **tile_floor.png**. Drag it into your project.

Then add the following lines to the end of `setupGL`:

```
NSDictionary * options =
    @{@"GLKTextureLoaderOriginBottomLeft": @YES };
NSError * error;
NSString * path = [[NSBundle mainBundle]
    pathForResource:@"tile_floor" ofType:@"png"];
GLKTextureInfo * info = [GLKTextureLoader
    textureWithContentsOfFile:path options:options error:&error];
if (info == nil) {
    NSLog(@"Error loading file: %@", error.localizedDescription);
}
_effect.texture2d0.name = info.name;
_effect.texture2d0.enabled = true;
```

This code loads the `tile_floor.png` image and sends it to OpenGL ES as a texture that you can use to render with. All you had to do was use the `textureWithContentsOfFile:options:error:` method on the `GLKTextureLoader` singleton and pass in a path, and you get back a class with information about the texture, including the OpenGL ES "name" (in other words, "ID") that you can use for rendering.

Note that you pass a dictionary of options into the `GLKTextureLoader`. By default the origin of a texture you load is the upper left of the texture, but that's annoying because in OpenGL ES the origin is the lower left. So here you set a flag to make the texture coordinates match up to the OpenGL coordinates.

Note: If you're confused about the syntax for creating a dictionary here, check out Chapter 2 in *iOS 6 by Tutorials*, "Programming in Modern Objective-C."

Also notice that after you load the texture, you set a few more properties on the effect to set up the texture for use. You set the texture's name, and mark it as enabled.

Now to actually render a texture, you need to pass the texture coordinates of each pixel to the `GLKBaseEffect`'s shader. Modify your `Vertex` structure and array to include the texture coordinates as follows:

```
typedef struct {
    float Position[3];
    float Color[4];
    float TexCoord[2];
} Vertex;

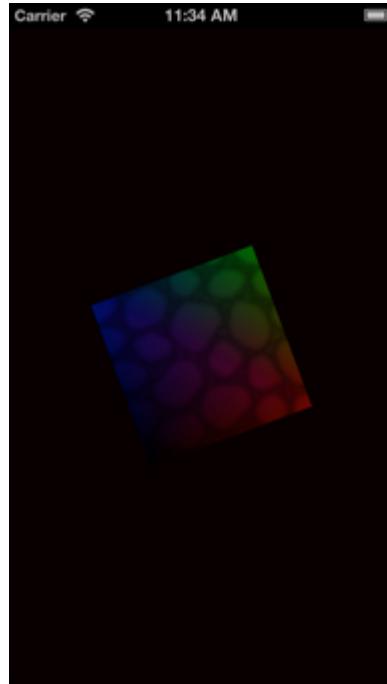
const Vertex Vertices[] = {
    {{1, -1, 0}, {1, 0, 0, 1}, {1, 0}},
    {{1, 1, 0}, {0, 1, 0, 1}, {1, 1}},
    {{-1, 1, 0}, {0, 0, 1, 1}, {0, 1}},
    {{-1, -1, 0}, {0, 0, 0, 1}, {0, 0}}
};
```

Also add these lines to the bottom of `setupGL`, before calling `glBindVertexArrayOES(0)`:

```
glEnableVertexAttribArray(GLKVertexAttribTexCoord0);
glVertexAttribPointer(GLKVertexAttribTexCoord0, 2, GL_FLOAT,
                     GL_FALSE, sizeof(Vertex),
                     (const GLvoid *) offsetof(Vertex, TexCoord));
```

These lines should be a good review of what you've already learned in the previous chapter. Try to walk through each of the parameters here and make sure you understand how it works. If you aren't sure, review step 4 in the *Introducing GLKBaseEffect* section in the previous chapter.

Build and run, and now your rotating square has a nice texture applied to it!



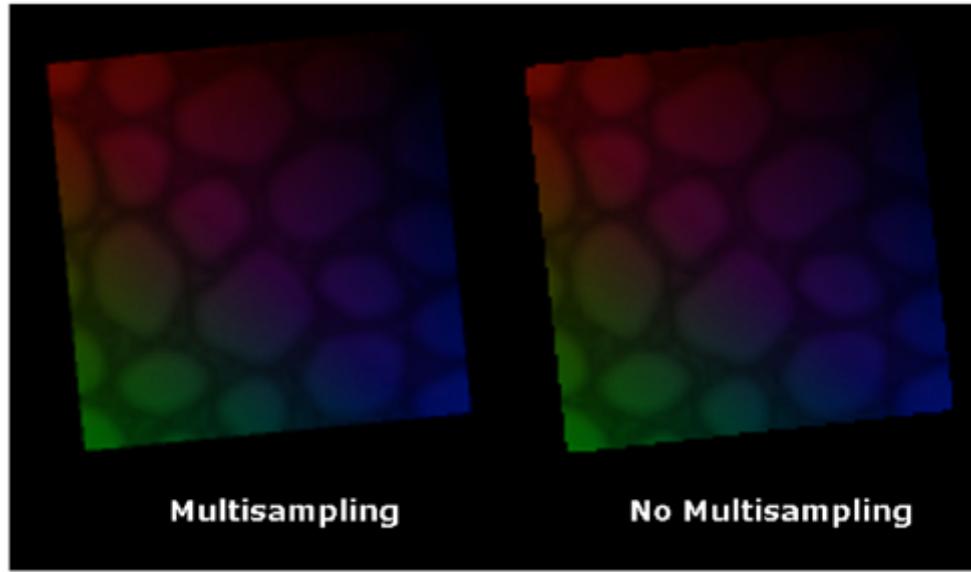
Gratuitous multisampling

In this section, I just wanted to show you another reason why GLKit is awesome. Run the app and watch the square rotate, and see if you notice jagged lines.

Then add the following line of code in `viewDidLoad`, right before the call to `[self setupGL]`:

```
view.drawableMultisample = GLKViewDrawableMultisample4X;
```

This single line enables multisampling (as you already know if you have read the optional *GLKit properties* section in the previous chapter). This renders your pixels at finer levels of granularity and merges the results to get an anti-aliased look, as you can see below:



Run the app and see if you notice the difference. Not bad for one line of code, eh? You'll be especially happy if you ever had to write the code the old long way! ☺

Moving to 3D

It's the moment you've been waiting for – now you're going to move into full 3D by converting this square into a cube!

Simply replace the `vertices` and `indices` arrays with the following:

```
const Vertex Vertices[] = {
    // Front
    {{1, -1, 1}, {1, 0, 0, 1}, {1, 0}},
    {{1, 1, 1}, {0, 1, 0, 1}, {1, 1}},
    {{-1, 1, 1}, {0, 0, 1, 1}, {0, 1}},
    {{-1, -1, 1}, {0, 0, 0, 1}, {0, 0}},
    // Back
    {{1, 1, -1}, {1, 0, 0, 1}, {0, 1}},
    {{-1, -1, -1}, {0, 1, 0, 1}, {1, 0}},
    {{1, -1, -1}, {0, 0, 1, 1}, {0, 0}},
    {{-1, 1, -1}, {0, 0, 0, 1}, {1, 1}},
    // Left
    {{-1, -1, 1}, {1, 0, 0, 1}, {1, 0}},
    {{-1, 1, 1}, {0, 1, 0, 1}, {1, 1}},
    {{-1, 1, -1}, {0, 0, 1, 1}, {0, 1}},
    {{-1, -1, -1}, {0, 0, 0, 1}, {0, 0}},
    // Right
    {{1, -1, -1}, {1, 0, 0, 1}, {1, 0}},
```

```

    {{1, 1, -1}, {0, 1, 0, 1}, {1, 1}},
    {{1, 1, 1}, {0, 0, 1, 1}, {0, 1}},
    {{1, -1, 1}, {0, 0, 0, 1}, {0, 0}},
    // Top
    {{1, 1, 1}, {1, 0, 0, 1}, {1, 0}},
    {{1, 1, -1}, {0, 1, 0, 1}, {1, 1}},
    {{-1, 1, -1}, {0, 0, 1, 1}, {0, 1}},
    {{-1, 1, 1}, {0, 0, 0, 1}, {0, 0}},
    // Bottom
    {{1, -1, -1}, {1, 0, 0, 1}, {1, 0}},
    {{1, -1, 1}, {0, 1, 0, 1}, {1, 1}},
    {{-1, -1, 1}, {0, 0, 1, 1}, {0, 1}},
    {{-1, -1, -1}, {0, 0, 0, 1}, {0, 0}}
};

const GLubyte Indices[] = {
    // Front
    0, 1, 2,
    2, 3, 0,
    // Back
    4, 6, 5,
    4, 5, 7,
    // Left
    8, 9, 10,
    10, 11, 8,
    // Right
    12, 13, 14,
    14, 15, 12,
    // Top
    16, 17, 18,
    18, 19, 16,
    // Bottom
    20, 21, 22,
    22, 23, 20
};

```

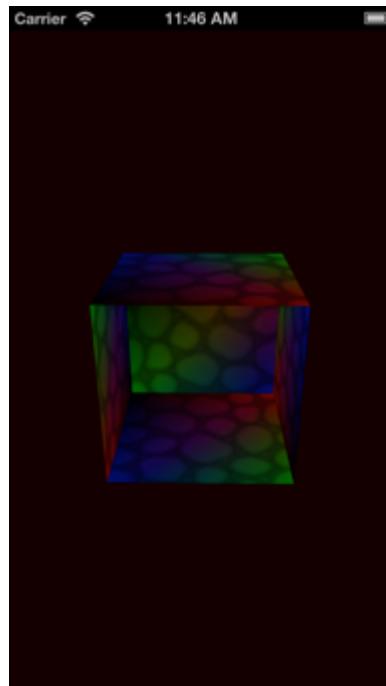
I got these by simply sketching them out on a piece of paper – it's a good exercise if you want to do the same!

However there is one tricky bit – you need to make sure you specify the indices in counter-clockwise order (from the perspective of looking at the box from the outside), for reasons I'll get into shortly.

Next, make some minor changes to the model view matrix setup at the end of `update`, to switch the rotation axis to the y-axis and rotate it slightly along the x-axis:

```
GLKMatrix4 modelViewMatrix =
    GLKMatrix4MakeTranslation(0.0f, 0.0f, -6.0f);
_rotation += 90 * self.timeSinceLastUpdate;
modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix,
    GLKMathDegreesToRadians(25), 1, 0, 0);
modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix,
    GLKMathDegreesToRadians(_rotation), 0, 1, 0);
_effect.transform.modelviewMatrix = modelViewMatrix;
```

Build and run and it sort of works... but there are some strange oddities! First of all it looks like the cube is semi see through sometimes:



That is because you haven't given OpenGL any criteria for how to tell when a face of the cube is facing the front, or when a face of the cube is facing the back. Because of this, sometimes it is drawing the back face on top of where the front face should be, or similarly the side faces in front of the front face.

Luckily that is an easy fix. Since you were careful to define the vertices in counter-clockwise order, you can enable an OpenGL flag called `GL_CULL_FACE`. This makes OpenGL skip drawing any frame that is backwards facing (i.e. not visible, since you have a cube!)

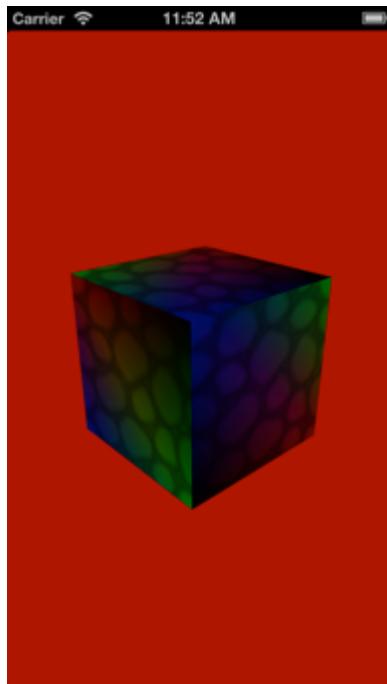
To do this, simply add the following line of code to `setupGL` after setting the context:

```
glEnable(GL_CULL_FACE);
```

Note: You could have solved this drawing issue by enabling depth testing and the `GLKView` depth buffer. However, using backface culling like this is much more efficient since it doesn't have to draw the back facing triangles at all. This can be a big performance improvement for more complicated games.

However, depth testing is still necessary when you have geometry that may cover other objects.

Build and run – and it works, you now have a rotating 3D cube with OpenGL ES 2.0 and GLKit!



Enabling lighting

Let's start with a bang (or should I say flash?) and try out some of the cool lighting effects you can get rather easily with `GLKBaseEffect`.

So far you have just set three properties on `GLKBaseEffect` – the `transform.projectionMatrix`, the `transform.modelViewMatrix`, and `texture2d0`. There are many other properties you can configure as well – including properties to set up lighting!

You can set up to three different lights on a `GLKBaseEffect` – the properties you use to configure them are named `light0`, `light1`, and `light2`. Each of these are instances of `GLKEffectPropertyLight`, which is a class that has many properties you can set to configure how the light works.

Let's get a simple light working, then you'll dive deeper into all the various ways you can configure it.

Add the following lines of code to the end of `setupGL`:

```
_effect.light0.enabled = GL_TRUE;  
_effect.light0.diffuseColor = GLKVector4Make(1, 1, 1, 1.0);  
_effect.light0.position = GLKVector4Make(1, 1, 0, 1);
```

Here you set three properties on the light:

- **enabled**: Turns the light on (default is off).
- **diffuseColor**: You'll learn more about this later, but for now just think of this as the color of the light.
- **position**: This vector indicates the position or direction of the light in 3D space. The last component of the vector indicates whether it is positional (1) or directional (0). Here you've set the light to be positional at (1, 1, 0) in 3D space.

w00t we're done, right? Compile and run, and you'll see this:



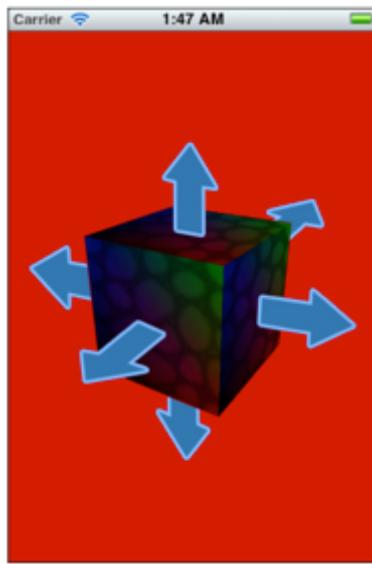
Sadly, it appears that you have no light at all. What gives?

The answer comes in the next section – vertex normals!

Vertex normals

In order for the shader to know how to calculate lighting, it needs to know how the light should bounce off a particular surface. The way you handle this is you give OpenGL ES something called a vertex normal, which is a fancy way of saying a unit vector that is perpendicular to the surface at that point.

To see what I mean, here's a diagram of your cube with the normals for each surface:



You specify the normals at each vertex rather than each surface, which allows you to get some neat effects such as bump mapping and more realistic looking surfaces. But for your simple cube, the vertex normals will be the same as the surface normals for each vertex on that face.

To add vertex normals for each vertex, you need to update your `Vertex` structure and array. So update it to the following:

```
typedef struct {
    float Position[3];
    float Color[4];
    float TexCoord[2];
    float Normal[3];
} Vertex;

const Vertex Vertices[] = {
    // Front
    {{1, -1, 1}, {1, 0, 0, 1}, {1, 0}, {0, 0, 1}},
    {{1, 1, 1}, {0, 1, 0, 1}, {1, 1}, {0, 0, 1}},
    {{-1, 1, 1}, {0, 0, 1, 1}, {0, 1}, {0, 0, 1}},
```

```

    {{{-1, -1, 1}, {0, 0, 0, 1}, {0, 0}, {0, 0, 1}}},
    // Back
    {{1, 1, -1}, {1, 0, 0, 1}, {0, 1}, {0, 0, -1}},
    {{-1, -1, -1}, {0, 1, 0, 1}, {1, 0}, {0, 0, -1}},
    {{1, -1, -1}, {0, 0, 1, 1}, {0, 0}, {0, 0, -1}},
    {{-1, 1, -1}, {0, 0, 0, 1}, {1, 1}, {0, 0, -1}},
    // Left
    {{-1, -1, 1}, {1, 0, 0, 1}, {1, 0}, {-1, 0, 0}},
    {{-1, 1, 1}, {0, 1, 0, 1}, {1, 1}, {-1, 0, 0}},
    {{-1, 1, -1}, {0, 0, 1, 1}, {0, 1}, {-1, 0, 0}},
    {{-1, -1, -1}, {0, 0, 0, 1}, {0, 0}, {-1, 0, 0}},
    // Right
    {{1, -1, -1}, {1, 0, 0, 1}, {1, 0}, {1, 0, 0}},
    {{1, 1, -1}, {0, 1, 0, 1}, {1, 1}, {1, 0, 0}},
    {{1, 1, 1}, {0, 0, 1, 1}, {0, 1}, {1, 0, 0}},
    {{1, -1, 1}, {0, 0, 0, 1}, {0, 0}, {1, 0, 0}},
    // Top
    {{1, 1, 1}, {1, 0, 0, 1}, {1, 0}, {0, 1, 0}},
    {{1, 1, -1}, {0, 1, 0, 1}, {1, 1}, {0, 1, 0}},
    {{-1, 1, -1}, {0, 0, 1, 1}, {0, 1}, {0, 1, 0}},
    {{-1, 1, 1}, {0, 0, 0, 1}, {0, 0}, {0, 1, 0}},
    // Bottom
    {{1, -1, -1}, {1, 0, 0, 1}, {1, 0}, {0, -1, 0}},
    {{1, -1, 1}, {0, 1, 0, 1}, {1, 1}, {0, -1, 0}},
    {{-1, -1, 1}, {0, 0, 1, 1}, {0, 1}, {0, -1, 0}},
    {{-1, -1, -1}, {0, 0, 0, 1}, {0, 0}, {0, -1, 0}}
};

}

```

You should sketch out the normals for each surface on a piece of paper, and make sure you understand why the normal is set the way it is.

Now that you have this new information in your `Vertex` structure, you need to enable the built-in `GLKEffect` vertex attribute for normals and pass along the pointer to the data. So add the following to `setupGL`, right before the call to `glBindVertexArrayOES(0)`:

```

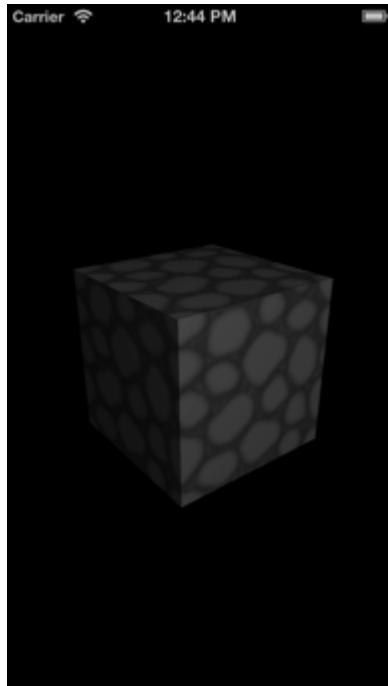
glEnableVertexAttribArray(GLKVertexAttribNormal);
glVertexAttribPointer(GLKVertexAttribNormal, 3, GL_FLOAT,
    GL_FALSE, sizeof(Vertex),
    (const GLvoid *) offsetof(Vertex, Normal)));

```

This should be familiar to you from the last chapter. Try to go through each parameter in the above method and make sure you understand what they do. If you don't recall, refer back to the last chapter for a full explanation.

One more thing. As you're playing around with lighting the pulsating effect is going to be confusing. So delete the code in `update` that modifies the `_curRed` value to disable that.

Build and run, and guess what – you've got lighting, all without writing a single line of shader code!



Note: You may notice that the cube appears gray (the color of the `tile_floor` texture) rather than the neat rainbow-colored cube as before. It turns out that when you enable lighting, by default the `GLKBaseEffect` shader will ignore any color properties you pass in. Instead, it uses a property on the light called the surface material, which you'll learn about later.

If this is not what you want (and you want the color values you pass in used instead), you can set the `GLKEffect`'s `colorMaterialEnabled` property to `YES`. If you do this, make sure you don't have any properties set on the material, such as `specularColor`.

Light colors and materials

As I alluded to earlier, there are different types of light that influence the final color of a pixel:

- **Diffuse light:** This is the light that is emitted from a particular point, and the light is stronger the more the surface faces toward the light. This type of light really helps make objects feel 3D.
- **Ambient light:** This is the light that is applied equally to the geometry, no matter what direction it is facing. It simulates the natural light that is just bouncing everywhere around a room, that might even hit underneath surfaces. Generally you want to set this to a much smaller intensity than your diffuse color.
- **Specular light:** This is the light that is reflected almost like a mirror off a surface. It provides “shiny spots” on objects (think of a shiny spot on a marble).

When you set up a light on a `GLKEffect`, you can specify the colors for the diffuse, ambient and specular types independently. To make things even more configurable, you can also set the diffuse, ambient, and specular types on the “material” of the geometry itself. The colors of the light and material are combined in order to get the final color.

The easiest way to understand this is to try it out. Replace the code where you set up the light with the following:

```
_effect.light0.enabled = GL_TRUE;
_effect.light0.diffuseColor = GLKVector4Make(0, 1, 1, 1);
_effect.light0.ambientColor = GLKVector4Make(0, 0, 0, 1);
_effect.light0.specularColor = GLKVector4Make(0, 0, 0, 1);

_effect.lightModelAmbientColor = GLKVector4Make(0, 0, 0, 1);
_effect.material.specularColor = GLKVector4Make(1, 1, 1, 1);

_effect.light0.position = GLKVector4Make(0, 1.5, -6, 1);
_effect.lightingType = GLKLightingTypePerPixel;
```

In the first section you set up three types of colors on the light. You set the diffuse color to a teal color, and set the ambient and specular colors to black (which is the equivalent of “no light”).

You then set the `lightModelAmbientColor`, which defines the global ambient light in the scene. For demonstration purposes, you want to make sure that no light is affecting your cube except what you specifically set up on light 0, so you turn this off by setting it to black/no light”. The default value is {0.2, 0.2, 0.2, 1.0} by the way.

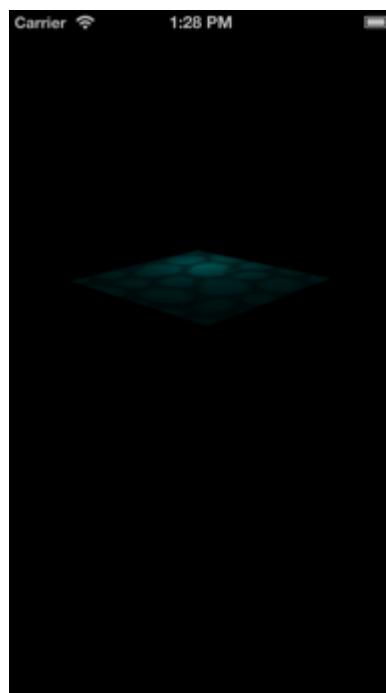
You then set the specular color of the material to white/full light”. It is important to set this, because the default value is {0, 0, 0, 1}, which means there would be no specular highlight at all, even if you set the light’s `specularColor`.

You move the position of the light to be right above where the box is (remember the box is 1 unit tall, and you translated the box 6 units back so it is visible).

Finally, you set a flag on the effect to make the lighting type “per pixel.” The default value is “per vertex”, which means that lighting is calculated once per

vertex and then interpolated across the surfaces. For some lighting effects this works OK, but sometimes you get strange effects with it set to this (especially when your polygons span across many pixels, like they do here). You get better lighting behavior if you set it to “per pixel”, but of course the tradeoff is increased processing time.

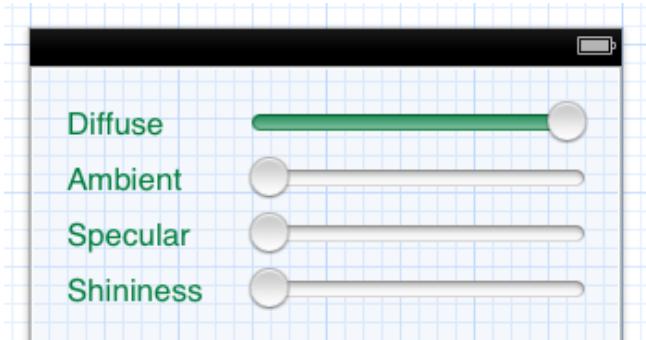
Build and run your code, and now you should see a cool teal light shining down on top of the box!



Since you disabled the global ambient component and since the light is right above the top face of the box, the light isn’t reaching any of the other faces and hence they aren’t getting any light.

Let’s experiment with these color values a bit more so you can get a better feel of how they work. Rather than having to continuously tweak these and recompile, let’s add some sliders to the view controller so you can dynamically modify them in real time!

Open up **MainStoryboard.storyboard**, and drag some labels and sliders into your view controller and make the following arrangement:



Here are some notes on the setup:

- For the first three sliders, set the min value to 0 and the max value to 1. But for the last slider, set the min value to 0 and the max value to 250.
- For the diffuse slider, set the current value to 1. For the rest, set the current value to 0.

Then bring up your assistant editor, make sure **HelloGLKitViewController.m** is visible, and control-drag from the **diffuse slider** down below the `@interface` on the class extension. Set the connection type to **Action** and connect it to **diffuseChanged**.

Repeat this for the other sliders, connecting them to **ambientChanged**, **specularChanged**, and **shininessChanged**, respectively.

Then open **HelloGLKitViewController.m** and implement these methods as follows:

```
- (IBAction)diffuseChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.diffuseColor =
        GLKVector4Make(0, slider.value, slider.value, 1);
}

- (IBAction)ambientChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.ambientColor =
        GLKVector4Make(0, slider.value, slider.value, 1);
}

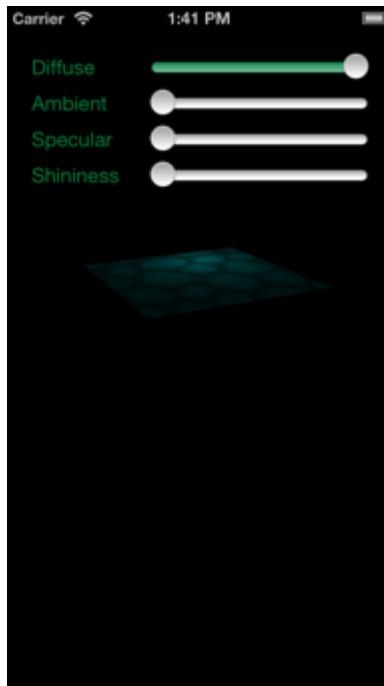
- (IBAction)specularChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.specularColor =
        GLKVector4Make(0, slider.value, slider.value, 1);
}

- (IBAction)shininessChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
```

```
_effect.material.shininess = slider.value;  
}
```

The callbacks simply update the various properties on the material. You've seen the first three before, but you haven't seen shininess yet. The higher the shininess value, the more "focused and reflective" the specular component appears.

To see what I mean, you can play around! Build and run, and you'll see the following:



And then start playing around! By experimenting, you should get a good feel of how the various color components work. Be sure to try the following:

- Drag the ambient slider up to see everything light up on the screen – even surfaces not facing the light.
- Drag the specular component and you'll see the top appear to lighten up. This doesn't look very useful until you drag the shininess value up – then you'll see what appears to be a reflection of the light, making the surface appear nice and shiny! The higher you drag the shininess, the more focused the reflection is.

While you're here, feel free to go back to the code and tweak the colors of the light or material if you'd like to play around with those too.

When you're ready, come back here and we'll continue to cover some more cool properties you can set on lights!

Spotlights and attenuation

You can easily configure a light to work as a spotlight by setting the following properties:

- **position**: To make a spotlight, when you set the light's position the last component has to be 1, indicating the light is positional rather than directional. You've already done this.
- **spotDirection**: Once you have a position for the light, you have to specify the direction the spotlight is shining by setting this property to a vector.
- **cutoff**: This defaults to 180 degrees, which means the light does not act as a spotlight. So if you want a spotlight, just set it to less than 180 degrees, and it will indicate the range at which no light is emitted.
- **exponent**: By making this value larger, you make the light brighter toward the center of the spotlight, and darker the further out you get from the center.

You've already set the `position`, and you'll add some sliders in a bit to set the `cutoff` and `exponent`, but you do need to set the `spotDirection`. So add the following to the bottom of `setupGL`:

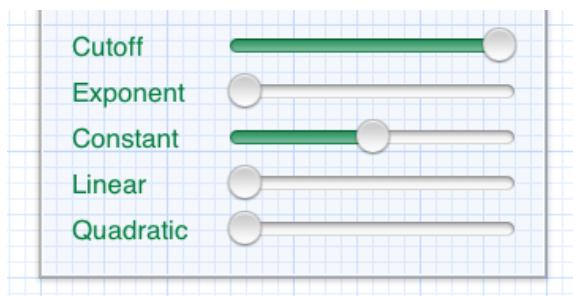
```
_effect.light0.spotDirection = GLKVector3Make(0, -1, 0);
```

This sets the light to point straight down onto the cube.

Another set of properties that are useful to set on lights are its attenuation properties. These allow you to make the light get darker the further away from the light the geometry is. There are three different values you can set here – `constantAttenuation`, `linearAttenuation`, and `quadraticAttenuation`.

The best way to understand these is through experimentation, so you will add some sliders so you can play around with them and see how they work. But if you're curious and want to see the equation that is used behind the scenes, see the [GLKEffectPropertyLight Class Reference](#).

OK – let's try these out. Open up **MainStoryboard.storyboard**, and add some more sliders and labels into the bottom of the view, like the following:



After you add the sliders, connect them to action methods like you did before. Here are some notes on setting everything up:

- The cutoff slider should have min 0, max 180, current 180. Connect it to cutoffValueChanged.
- The exponent slider should have min 0, max 100, current 0. Connect it to exponentValueChanged.
- The constant slider should have min 0, max 2, current 1. Connect it to constantValueChanged.
- The linear slider should have min 0, max 2, current 0. Connect it to linearValueChanged.
- The quadratic slider should have min 0, max 2, and current 0. Connect it to quadraticValueChanged.

Then switch to **HelloGLKitViewController.m** and implement the methods like the following:

```
- (IBAction)cutoffValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.spotCutoff = slider.value;
}

- (IBAction)exponentValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.spotExponent = slider.value;
}

- (IBAction)constantValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.constantAttenuation = slider.value;
}

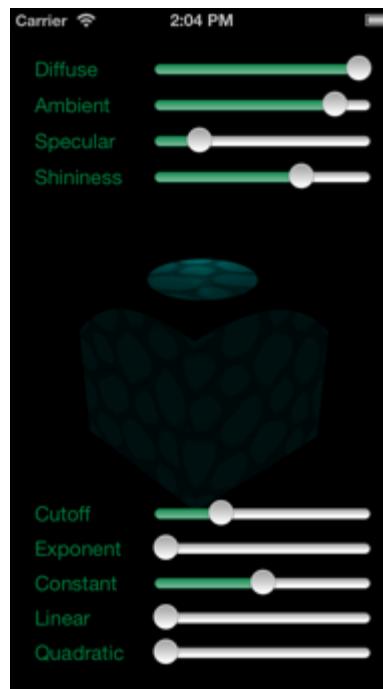
- (IBAction)linearValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.linearAttenuation = slider.value;
}

- (IBAction)quadraticValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    _effect.light0.quadraticAttenuation = slider.value;
}
```

Here you simply set each of these values so you can experiment with them. That's it – build and run the app, and play around! Be sure to try the following:

- Drag the cutoff slider down until you see a circular spotlight area.
- With a large cutoff, drag the exponent down to see the center area focused with light even though the cutoff is large.

- Play around with the constant, linear, and quadratic attenuation to see the effects of influencing how far the light shines before it fades away (and how quickly it does!)



A moving light

I'd like to show you one more thing about lights, and then we're done covering those for now.

Let's add one more light to your scene, that revolves around your cube, just for fun!

First add a private instance variable for the light rotation:

```
float _lightRotation;
```

Then initialize a new light at the end of `setupGL` with the following properties:

```
_effect.light1.enabled = GL_TRUE;
_effect.light1.diffuseColor = GLKVector4Make(1.0, 1.0, 0.8, 1.0);
_effect.light1.position = GLKVector4Make(0, 0, 1.5, 1);
```

Here you make the light yellowish, and set the initial position to be 1.5 along the z-axis.

Next add the following code inside `update`, right after setting the projection matrix:

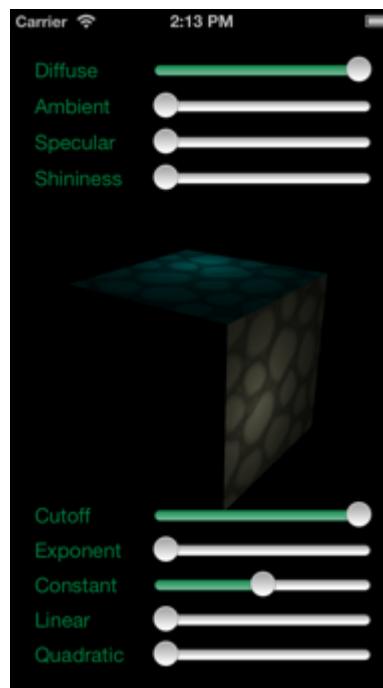
```
GLKMatrix4 lightModelViewMatrix = GLKMatrix4MakeTranslation(0.0f,
0.0f, -6.0f);
```

```
_lightRotation += -90 * self.timeSinceLastUpdate;
lightModelViewMatrix = GLKMatrix4Rotate(lightModelViewMatrix,
    GLKMathDegreesToRadians(25), 1, 0, 0);
lightModelViewMatrix = GLKMatrix4Rotate(lightModelViewMatrix,
    GLKMathDegreesToRadians(_lightRotation), 0, 1, 0);
_effect.transform.modelviewMatrix = lightModelViewMatrix;
_effect.light1.position = GLKVector4Make(0, 0, 1.5, 1);
```

The first thing you do is construct a model view matrix to transform the light's position each frame. You start by moving the light backwards so it stays near the box. Then you rotate it each frame – but the opposite direction of how the cube is rotating.

You set the transform model view matrix with the new transform. When you set the position of the light, it uses whatever the current transform is in the `modelViewMatrix` to arrive at its final position.

Build and run, and you should see a sweet rotating light around the cube!



Congratulations, you now have hands-on experience with every property that you can set on `GLKBaseEffect` related to lighting!

Note: Actually there is just one more – `lightModelTwoSided`. By default, the lighting algorithms only apply light to the side of the surface that is toward the light (with respect to the surface normal). If you want the light to be applied

to both sides, you can set this to `YES`. But note the tradeoff is decreased performance.

Multi-Texturing

The multi-texturing support in `GLKBaseEffect` is quite limited, but let's see what it can do.

From the resources for this chapter, drag the `item_powerup_fish.png` image into your project. Then add the following code at the end of `setupGL`:

```
path = [[NSBundle mainBundle]
    pathForResource:@"item_powerup_fish" ofType:@"png"];
info = [GLKTextureLoader textureWithContentsOfFile:path
    options:options error:&error];
if (info == nil) {
    NSLog(@"Error loading file: %@", error.localizedDescription);
}
_effect.texture2d1.name = info.name;
_effect.texture2d1.enabled = true;
_effect.texture2d1.envMode = GLKTextureEnvModeDecal;
```

Here you're loading a texture with `GLKTextureLoader` just like you did before, except this time you are setting it into the `texture2d1` property (the only other texture slot available in `GLKBaseEffect`).

The other difference is you set the `envMode` to `GLKTextureEnvModeDecal`. This tells the `GLKBaseEffect` to blend the first and second textures, based on the second texture's alpha values.

Still in `setupGL`, add these lines of code right before the call to `glBindArrayOES(0)`:

```
 glEnableVertexAttribArray(GLKVertexAttribTexCoord1);
 glVertexAttribPointer(GLKVertexAttribTexCoord1, 2, GL_FLOAT,
    GL_FALSE, sizeof(Vertex),
    (const GLvoid *) offsetof(Vertex, TexCoord));
```

Here you pass in the texture coordinates for the second texture. Note you used the same texture coordinates as the first texture to make things easy, but you could pass in different coordinates if you need to.

Compile and run, and you'll see the glowy fish on top of your textures:



One thing to notice about this technique is the light does not affect the decal texture – it always shows in full color. This may or may not be something you desire in your games.

For more complicated or different effects, you can always write your own custom shaders.

Fog

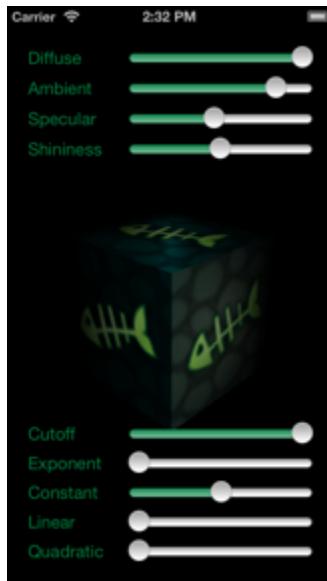
`GLKBaseEffect` also supports a fog effect, which can be useful to put where your far clipping plane is to make the transition of far away objects more smooth (instead of having them randomly disappear!) It can also be fun just for a neat fog effect. ☺

Using it is simple. Add the following code at the end of `setupGL`:

```
_effect.fog.color = GLKVector4Make(0, 0, 0, 1.0);
_effect.fog.enabled = YES;
_effect.fog.end = 5.5;
_effect.fog.start = 5.0;
_effect.fog.mode = GLKFogModeLinear;
```

Here you set the color of the fog black (since your background is black in this app). You also set the fog as enabled, and mark where the wall of fog begins and ends (in amount of units from the eye) and make the mode of the fog linear.

That's it – compile and run, and you should see a cool fog effect!



Where to go from here?

Sadly this is a book on the new APIs that came out in iOS 5, not a book on OpenGL ES 2.0, so we're going to have to call it quits from here.

However, at this point you should have a good grounding of the most important aspects of GLKit and should be ready to use it in your own apps, and continue your studies of OpenGL ES 2.0.

The best books I know of at the moment about OpenGL ES 2.0 is [Learning OpenGL ES for iOS](#) by Erik Buck and [iPhone 3D Programming](#) by Philip Rideout.

Also, we have several other free tutorials on OpenGL ES on raywenderlich.com, such as this tutorial which shows you how to make a simple 2D game with OpenGL ES 2.0 and GLKit: <http://www.raywenderlich.com/9743/how-to-create-a-simple-2d-iphone-game-with-opengl-es-2-0-and-glkit-part-1>

Best of luck with your future adventures with OpenGL ES 2.0 and GLKit, and I hope these chapters have helped you get started on your journey!

Chapter 10: Beginning UIKit Customization

By Steve Baranski and Adam Burkepile

To be successful on the App Store, your app needs to stand out. The vanilla user-interface “look and feel” provided by Apple just doesn’t cut it any more in a crowded market.

Many of the most popular apps on the App Store present standard iOS UI elements in a non-standard fashion:

- Twitter employs a custom `UITabBar`
- Instagram uses both a custom `UITabBar` and a custom `UINavigationBar`
- Epicurious for iPad customizes elements of the standard split-view interface

Prior to iOS 5, many developers had to take somewhat unconventional approaches to achieve these results. Although subclassing and overriding `drawRect:` was the recommended approach, many resorted to the dreaded “method swizzling”.

But with iOS 5 and above, those dark days are over! iOS 5 added a bunch of new APIs you can use to easily customize the appearance of various UIKit controls, and iOS 6 added even more.

To illustrate some of these new APIs, in this chapter you’re going to take a “plain vanilla” app about surfing trips and customize the UI to get a more “beach-themed” look-and-feel.



Getting Started

I've created a simple app for you to start with so you can focus on the meat of this chapter that is included in the resources for this chapter.

Go ahead and open the project - it's called *Surf's Up*. Then take a look around the code and Storyboard. You'll see that the primary view presents a list of the surfing trips, and the detail view allows us to capture more information about each trip individually. With that context, Build & Run the app (Cmd-R) to see what you have to start with.



Huh. Yes, this app is functional, but it's hardly representative of the fun one would expect to have on a surfing trip. Let's survey the scene in more detail.

Let's start with the detail page. Things look pretty standard there, eh?

A plain `UIBarButtonItem` on the `UINavigationBar` at the top, stock `UITabBar` elements at the bottom, and the following "standard" data entry components including the following:

- `UILabels` with "System" Helvetica fonts
- `UITextField`
- `UISlider`
- `UISwitch`
- `UISegmentedControl`
- `UINavigationBarShadow`
- `UIStepper`
- `UIProgressView`
- `UIPageControl`

In this chapter, you'll completely customize the detail screen to give it some style, using the new APIs available in iOS 5 and 6. So with an idea of what's in store, let's convert this app from "zero" to "hero".

Adding a Background Image

If you open up the Images folder in your project, you'll see that you already have some images you can use to customize the UI included in the project – you just need to modify the code to make use of them.

Inside the images folder is a file called **bg_sand.png**. You're going to start your UI customization by making this the background image in the detail view.

Open **DetailViewController.m** and create a **viewDidLoad** method like this:

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

    self.view.backgroundColor = [UIColor
        colorWithPatternImage:[UIImage imageNamed:@"bg_sand"]];
}
```

If you aren't familiar with this technique, you can actually make a "color" based on an image like you see here. This is an easy way to set the background of a view, because there's no "backgroundImage" property, but there is a "backgroundColor" property!

Compile and run to verify that it worked:



I don't know about you but I can feel the sand between my toes already!

Customizing UINavigationController

If you look inside the images folder, you'll see two images that you want to use to customize the navigation bar: **surf_gradient_textured_32.png** and **surf_gradient_textured_44.png**.

You want to repeat these from left to right across the navigation bar. There are two different heights because the height of the navigation bar shrinks when the phone goes into landscape.

iOS 5 offers two new APIs that can help us with this:

- `UINavigationBar` has a new `backgroundImage` property you can use to set a custom background image like this.
- `UIImage` has a new `resizableImageWithCapInsets` method you can use to create a resizable image. The cap insets allow you to specify the portions of the image that should not be repeated, such as if you have rounded corners for a button on the edges that shouldn't be repeated.

You could go into the detail view and use these new APIs to set the navigation bar's background image directly. But then you'd have to go and do the same thing inside the list view, and any other views you might have in your app!

Obviously this would get old quick. Recognizing this, iOS 5 offers a cool new feature that allows us to customize user interface elements once, allowing it to "stand in" for other elements within the same level in the containment hierarchy.

So starting with the navigation bar, you're going to use this concept of the "appearance proxy" to customize some elements that will be repeated throughout the app.

Let's see how it looks. Inside **SurfsUpAppDelegate.m**, create a new method right above **application:didFinishLaunchingWithOptions:**:

```
- (void)customizeAppearance
{
    // Create resizable images
    UIImage *gradientImage44 =
        [[UIImage imageNamed:@"surf_gradient_textured_44"]
         resizableImageWithCapInsets:
         UIEdgeInsetsMake(0, 0, 0, 0)];
    UIImage *gradientImage32 =
        [[UIImage imageNamed:@"surf_gradient_textured_32"]
         resizableImageWithCapInsets:
         UIEdgeInsetsMake(0, 0, 0, 0)];

    // Set the background image for *all* UINavigationBars
    [[UINavigationBar appearance]
     setBackgroundImage:gradientImage44
     forBarMetrics:UIBarMetricsDefault];
    [[UINavigationBar appearance]
     setBackgroundImage:gradientImage32
     forBarMetrics:UIBarMetricsLandscapePhone];

    // Customize the title text for *all* UINavigationBars
    [[UINavigationBar appearance] setTitleTextAttributes:
     [NSDictionary dictionaryWithObjectsAndKeys:
      [UIColor colorWithRed:255.0/255.0
                     green:255.0/255.0
                      blue:255.0/255.0
                     alpha:1.0],
      UITextAttributeTextColor,
      [UIColor colorWithRed:0.0
                     green:0.0
                      blue:0.0
                     alpha:0.8],
```

```
UITextAttributeTextShadowColor,  
[NSValue valueWithUIOffset:UIOffsetMake(0, -1)],  
UITextAttributeTextShadowOffset,  
[UIFont fontWithName:@"Arial-Bold" size:0.0],  
UITextAttributeFont,  
nil]];  
}
```

The first two lines create stretchable images using the `resizableImageWithCapInsets:` method discussed earlier. Note that this method replaces `stretchableImageWithLeftCapWidth:topCapHeight:`, which is now deprecated.

For the cap insets, you basically specify the fixed region of a given image in top, left, bottom, right. What's left is stretched over the remainder of the region to which the image is applied. In this particular image you want the whole thing stretched, so you pass 0 for all of the fixed caps.

The next two lines invoke the appearance proxy, designating these stretchable images as background images, for the bar metrics specified.

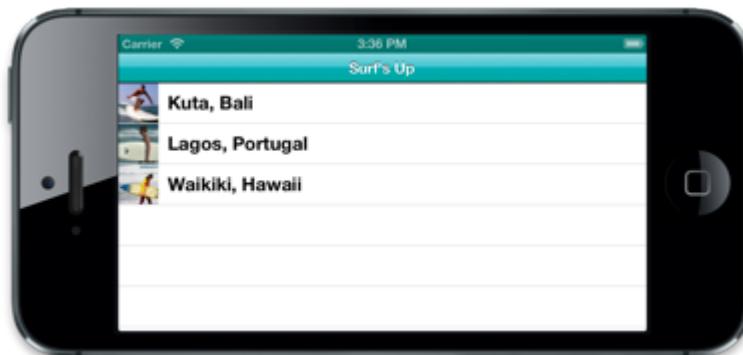
The last line stylizes the title that appears in your detail view. To do so, you pass a dictionary of title text attributes. The available keys include the following:

- `UITextAttributeFont`
- `UITextAttributeTextColor`
- `UITextAttributeTextShadowColor`
- `UITextAttributeTextShadowOffset`

Almost done – just add the line to call this method at the top of `application:didFinishLaunchingWithOptions:`:

```
[self customizeAppearance];
```

Compile and run, and now you should see the navigation bar has the teal background image applied in both orientations, with stylized title text as well!



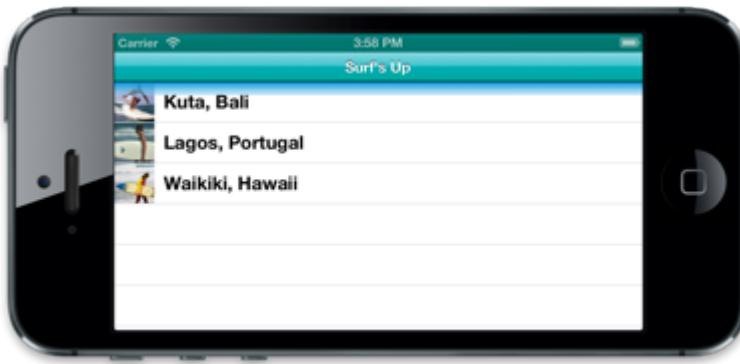
Changing the UINavigationBar Shadow

One of the new things that were added in iOS 6 was a small shadow underneath the navigation bar that adds a nice little transition from the navigation bar to the content. You also have the ability to customize that shadow to something else. Let see how easy that is to do now.

You already have the image **navBarShadow.png** and **navBarShadow@2x.png** added to the project so just add the following line to the bottom of your **customizeAppearance** method:

```
[[UINavigationBar appearance] setShadowImage:[UIImage  
 imageNamed:@"navBarShadow"]];
```

Build and run, and you will see the following:



Viola! Easy, huh? Kinda like a nice cool wave flowing up the sand.

Customizing UIBarButtonItem

Open up the Images directory and look at **button_textured_24.png** and **button_textured_30.png**. You want to use these to customize the look and feel of the buttons that appear in the UINavigationBar.

Notice that you're going to set up these button images as resizable images. It's important to make them resizable because the button widths will vary depending on what text is inside.

For these buttons, you don't want the 5 leftmost pixels to stretch, nor the 5 rightmost pixels, so you'll set the left and right cap insets to 5. The pixels in between will repeat as much as is needed for the width of the button.

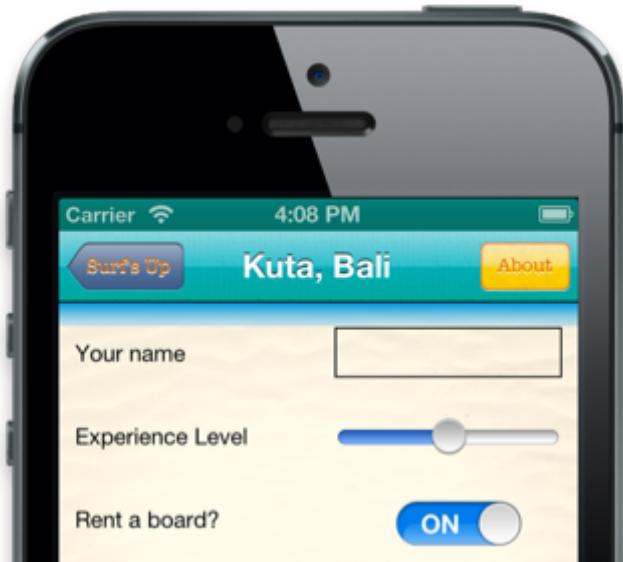
Let's try this out! You'll use the appearance proxy to customize all the **UIBarButtonItem**s at once, like you did last time. Add the following code to the end of **customizeAppearance**:

```
UIImage *button30 = [[UIImage imageNamed:@"button_textured_30"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 5, 0, 5)];
UIImage *button24 = [[UIImage imageNamed:@"button_textured_24"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 5, 0, 5)];
[[UIBarButtonItem appearance] setBackgroundImage:button30
    forState:UIControlStateNormal
    barMetrics:UIBarMetricsDefault];
[[UIBarButtonItem appearance] setBackgroundImage:button24
    forState:UIControlStateNormal
    barMetrics:UIBarMetricsLandscapePhone];

[[UIBarButtonItem appearance] setTitleTextAttributes:
    [NSDictionary dictionaryWithObjectsAndKeys:
        [UIColor colorWithRed:220.0/255.0
            green:104.0/255.0
            blue:1.0/255.0
            alpha:1.0],
        UITextAttributeTextColor,
        [UIColor colorWithRed:1.0 green:1.0 blue:1.0 alpha:1.0],
        UITextAttributeTextShadowColor,
        [NSValue valueWithUIOffset:UIOffsetMake(0, 1)],
        UITextAttributeTextShadowOffset,
        [UIFont fontWithName:@"AmericanTypewriter" size:0.0],
        UITextAttributeFont,
        nil]
    forState:UIControlStateNormal];
```

This looks familiar. You create the stretchable images for the buttons and set them as the background for both display in both portrait & landscape orientation. You then format the text to match the typewriter-style font you saw at the outset of the chapter.

Note you can set different images for different types of buttons such as the "Done" type.



The “back” bar button item needs special customization, because it should look different – like it’s pointing backwards. Take a look at the images you’re going to use to see what I mean: **Images\button_back_textured_24.png** and **Images\button_back_textured_30.png**.

Add the following code at the bottom of `customizeAppearance` to take care of the back bar button item:

```
UIImage *buttonBack30 = [[UIImage
                           imageNamed:@"button_back_textured_30"]
                          resizableImageWithCapInsets:UIEdgeInsetsMake(0, 13, 0, 5)];
UIImage *buttonBack24 = [[UIImage
                           imageNamed:@"button_back_textured_24"]
                          resizableImageWithCapInsets:UIEdgeInsetsMake(0, 12, 0, 5)];
[[UIBarButtonItem appearance]
 setBackButtonBackgroundImage:buttonBack30
 forState:UIControlStateNormal
 barMetrics:UIBarMetricsDefault];
[[UIBarButtonItem appearance]
 setBackButtonBackgroundImage:buttonBack24
 forState:UIControlStateNormal
 barMetrics:UIBarMetricsLandscapePhone];
```

Note that you use different cap inset values because the back image has a wider left hand side that shouldn’t stretch. Also note that there is a separate property on `UIBarButtonItem` for `backButtonBackgroundImage` that you use here.

Compile and run, and you should now see some cool customized `UIBarButtonItem`s in your `UINavigationBar`!



Customizing UITabBar

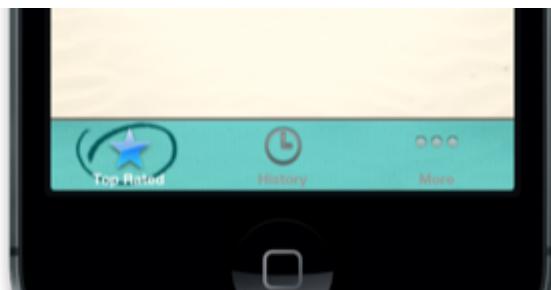
To customize a `UITabBar`, iOS 5 offers an API to let you change the background image of the toolbar, and the image to indicate the selected item. Take a look at **Images\tab_bg.png** and **Images\tab_select_indicator.png** to see the images you'll use for these.

Although your mockups only depict one `UITabBar`, these will in all likelihood have the same appearance if others appear, so you'll use the appearance proxy to customize this as well.

Add the following code to the bottom of `customizeAppearance`:

```
UIImage *tabBackground = [[UIImage imageNamed:@"tab_bg"]
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 0, 0, 0)];
[[UITabBar appearance] setBackgroundImage:tabBackground];
[[UITabBar appearance] setSelectionIndicatorImage:
    [UIImage imageNamed:@"tab_select_indicator"]];
```

Compile and run again... nice! The background and selected image are nice touches.



Note you can also specify "finished" and "unfinished" images if you wish to modify the manner in which the selected & unselected images appear.

Customizing UISlider

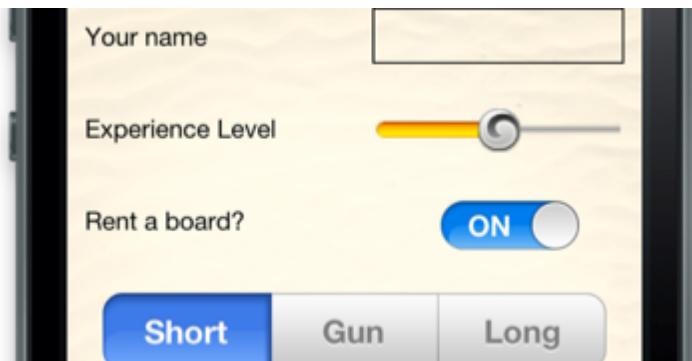
Open up **Images\slider_minimum.png**, **Images\slider_maximum.png**, and **Images\thumb.png** to see the images that you're going to use to customize the UISlider.

iOS 5 makes it ridiculously easy to customize the `UISlider` by just setting the `maximumTrackImage`, `minimumTrackImage`, and `thumbImage` properties of a `UISlider`.

Let's try it out. Add the following code to the bottom of `customizeAppearance`:

```
UIImage *minImage = [[UIImage imageNamed:@"slider_minimum.png"]  
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 5, 0, 0)];  
UIImage *maxImage = [[UIImage imageNamed:@"slider_maximum.png"]  
    resizableImageWithCapInsets:UIEdgeInsetsMake(0, 5, 0, 0)];  
UIImage *thumbImage = [UIImage imageNamed:@"thumb.png"];  
  
[[UISlider appearance] setMaximumTrackImage:maxImage  
    forState:UIControlStateNormal];  
[[UISlider appearance] setMinimumTrackImage:minImage  
    forState:UIControlStateNormal];  
[[UISlider appearance] setThumbImage:thumbImage  
    forState:UIControlStateNormal];
```

Compile and run, and check out your cool and stylish `UISlider`!



Customizing UISegmentedControl

Now you'll customize your segmented control. This component is a little bit more complicated; you have both selected & unselected backgrounds, as well as varying states for the adjacent regions (e.g., selected on left, unselected on right; unselected on the left & selected on the right; unselected on both sides).

Take a look at the images you'll use for this to see what I mean:

Images\segcontrol_sel.png, **Images\segcontrol_uns.png**,
Images\segcontrol_sel-uns.png, and **Images\segcontrol_uns-uns.png**.

Then add the code to make use of these to the bottom of `customizeAppearance`:

```
UIImage *segmentSelected =
    [[UIImage imageNamed:@"segcontrol_sel.png"]
     resizableImageWithCapInsets:UIEdgeInsetsMake(0, 15, 0, 15)];
UIImage *segmentUnselected =
    [[UIImage imageNamed:@"segcontrol_uns.png"]
     resizableImageWithCapInsets:UIEdgeInsetsMake(0, 15, 0, 15)];
UIImage *segmentSelectedUnselected =
    [UIImage imageNamed:@"segcontrol_sel-uns.png"];
UIImage *segUnselectedSelected =
    [UIImage imageNamed:@"segcontrol_uns-sel.png"];
UIImage *segmentUnselectedUnselected =
    [UIImage imageNamed:@"segcontrol_uns-uns.png"];

[[UISegmentedControl appearance]
    setBackgroundImage:segmentUnselected
        forState:UIControlStateNormal
        barMetrics:UIBarMetricsDefault];
[[UISegmentedControl appearance]
    setBackgroundImage:segmentSelected
        forState:UIControlStateNormal
        barMetrics:UIBarMetricsDefault];

[[UISegmentedControl appearance]
    setDividerImage:segmentUnselectedUnselected
        forLeftSegmentState:UIControlStateNormal
        rightSegmentState:UIControlStateNormal
        barMetrics:UIBarMetricsDefault];
[[UISegmentedControl appearance]
    setDividerImage:segmentSelectedUnselected
        forLeftSegmentState:UIControlStateNormal
        rightSegmentState:UIControlStateNormal
        barMetrics:UIBarMetricsDefault];
[[UISegmentedControl appearance]
    setDividerImage:segUnselectedSelected
        forLeftSegmentState:UIControlStateNormal
        rightSegmentState:UIControlStateNormal
        barMetrics:UIBarMetricsDefault];
```

Compile and run, and now your `UISegmentedControl` has a completely different look!



Customizing UISwitch

In iOS 5 you only had the ability to customize the tint of the On side of the switch (which was kinda a weird decision if you ask me). But luckily in iOS 6 you now have access to the Off side and the thumb (middle) part.

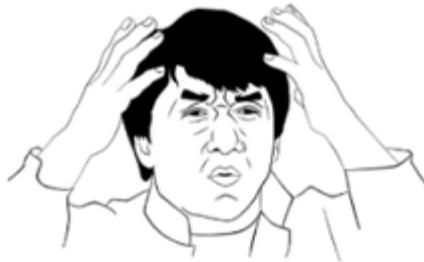
So, add the following code to set the tint colors `customizeAppearance` method:

```
[[UISwitch appearance] setOnTintColor:  
    [UIColor colorWithRed:0  
                  green:175.0/255.0  
                   blue:176.0/255.0  
                  alpha:1.0]];  
  
[[UISwitch appearance] setTintColor:  
    [UIColor colorWithRed:1.000  
                  green:0.989  
                   blue:0.753  
                  alpha:1.000]];  
  
[[UISwitch appearance] setThumbTintColor:[UIColor cyanColor]];
```

Compile and run, and check out your newly colored switch!



The switch fits the look now but “ON” and “OFF” don’t really make sense. I mean if someone asked you if you’d like a glass of water and you said “ON”, they’d probably look at you kinda weird!



Well, it just so happens that in iOS 6 you gained the ability to customize the images inside of the switch as well. It’s not text, but if you make an image that is text, it works just as well. Add the following lines at the end of the `customizeAppearance` method:

```
[[UISwitch appearance] setOnImage:[UIImage
                               imageNamed:@"yesSwitch"]];
[[UISwitch appearance] setOffImage:[UIImage
                               imageNamed:@"noSwitch"]];
```

Now the switch says “Yes/No” instead of “On/Off”! You could change this to an icon if you’d like too.

Things are looking pretty good, but you still have a couple of items outstanding. You need to update the labels and set the background of your custom UITextField.

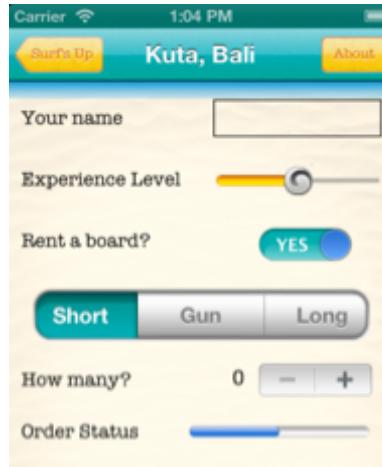
Customizing UILabel

The labels are one part of the detail view you won’t customize via the appearance proxy. Open the storyboard and navigate to the `DetailViewController`. Start by selecting the first label (i.e., “Your name”) in the main view, the in the Utilities view (i.e., the right pane), select the Attributes Inspector and set the following:

- **Font:** Custom
- **Family:** American Typewriter
- **Style:** Regular
- **Size:** 16

Repeat this for the four remaining labels: "Experience Level", "Rent a board?", "How many?", and "Order Status".

Compile and run, and now your labels have a neat typewriter feel!



Customizing UITextField

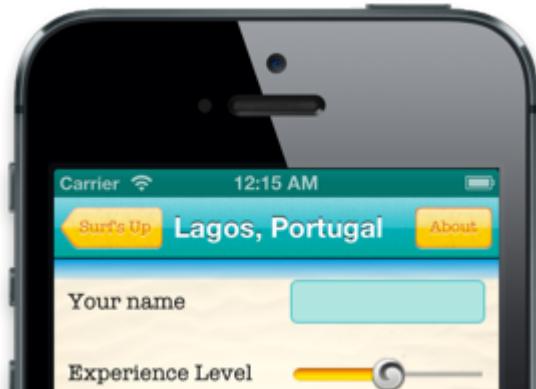
Our `UITextField` has already been set to use `UITextBorderStyleLine`. Since you're still in Interface Builder, let's set the font to American Typewriter, Size 12, Regular. Now if you look at the Identity Inspector, you'll see that the Custom Class has been defined as something other than `UITextField` – `CustomTextField`. If you look in the Navigator pane on the left, there is a group called Custom Views. Expand that, and you will see that you have a type called exactly that.

Right now the `drawRect:` method of your `UITextField` delegates to the superclass implementation. But in order to paint the teal background, you are going to override `drawRect:` as another customization technique.

Replace the implementation of `drawRect` in **CustomTextField.m** with the following code:

```
- (void)drawRect:(CGRect)rect
{
    UIImage *textFieldBackground =
        [[UIImage imageNamed:@"text_field_teal.png"]
         resizableImageWithCapInsets:UIEdgeInsetsMake(15.0, 5.0,
                                                     15.0, 5.0)];
    [textFieldBackground drawInRect:[self bounds]];
}
```

Here you create yet another stretchable image with appropriate insets, and draw it in the rectangle defined by the bounds of this view (i.e., your `UITextField`). Build and Run, and you'll see the following:

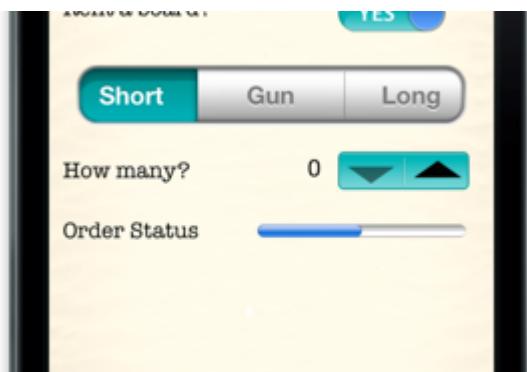


Customizing UIStepper

With iOS 5 you gained the ability to change the tint color of the stepper and now in iOS 6 you can change the individual icons in the sides and middle of the control. Let's see how you can do this now. Add the following code to the end of your `customizeAppearance` method in **SurfsUpAppDelegate.m**:

```
[[UIStepper appearance] setTintColor:  
    [UIColor colorWithRed:0  
                  green:175.0/255.0  
                  blue:176.0/255.0  
                 alpha:1.0]];  
  
[[UIStepper appearance] setIncrementImage:  
    [UIImage imageNamed:@"up"]  
    forState:UIControlStateNormal];  
  
[[UIStepper appearance] setDecrementImage:  
    [UIImage imageNamed:@"down"]  
    forState:UIControlStateNormal];
```

Build and run, and enjoy your stylish stepper!



Customizing UIProgressView

The `UIProgressView` is a pretty simple control to customize, just a track tint and a completed tint. Add the following code to the end of your `customizeAppearance` method:

```
[ [UIProgressView appearance] setProgressTintColor:  
    [UIColor colorWithRed:0  
                  green:175.0/255.0  
                  blue:176.0/255.0  
                 alpha:1.0]];  
  
[ [UIProgressView appearance] setTrackTintColor:  
    [UIColor colorWithRed:0.996  
                  green:0.788  
                  blue:0.180  
                 alpha:1.000]];
```

Give it another build and run and see your lovely progress bar!

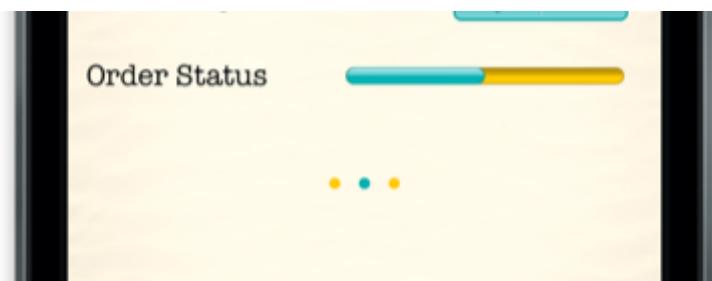


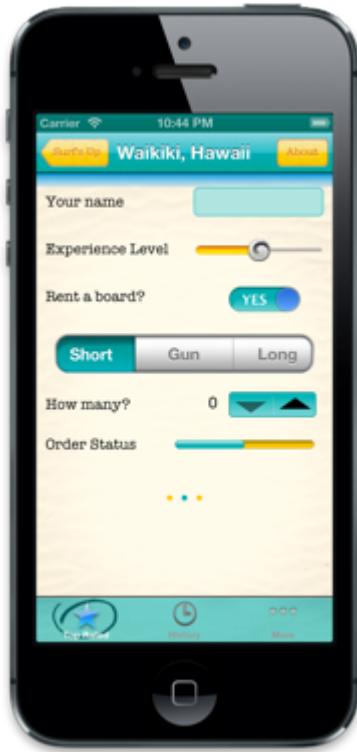
Customizing UIPageControl

UIPageControl is the last control you're going to customize. You can customize the page indicators as well as a separate color for the current page, both properties are new to iOS 6. Add the following code to the end of your `customizeAppearance` method:

```
[ [UIPageControl appearance] setCurrentPageIndicatorTintColor:  
    [UIColor colorWithRed:0  
                  green:175.0/255.0  
                  blue:176.0/255.0  
                 alpha:1.0]];  
  
[ [UIPageControl appearance] setPageIndicatorTintColor:  
    [UIColor colorWithRed:0.996  
                  green:0.788  
                  blue:0.180  
                 alpha:1.000]];
```

Build and run... ahh, how beautiful!





Congratulations – the detail view is complete, and you've learned a ton about customizing the looks of your apps!

Where To Go From Here?

Congratulations - you now have experience customizing the most common controls in UIKit with the new `UIAppearance` APIs! You'll never have to make a plain vanilla UIKit app again :]

If you want to learn more about customizing `UIKit` controls, keep reading the next chapter, where you'll learn how to customize the table view, port the app to the iPad, and much more!

Chapter 11: Intermediate UIKit Customization

By Steve Baranski and Adam Burkepile

In the previous chapter, you took a "plain vanilla" app with the default UIKit controls, and customized the look-and-feel to add some style and flair. You customized the background image, `UINavigationBar`, `UIBarButtonItem`, `UITabBar`, `UISlider`, `UISegmentedControl`, `UISwitch`, `UILabel`, `UITextField`, `UISwitch`, `UIProgressView`, `UIStepper`, and `UIPageControl`.

Wow, you were busy! ☺

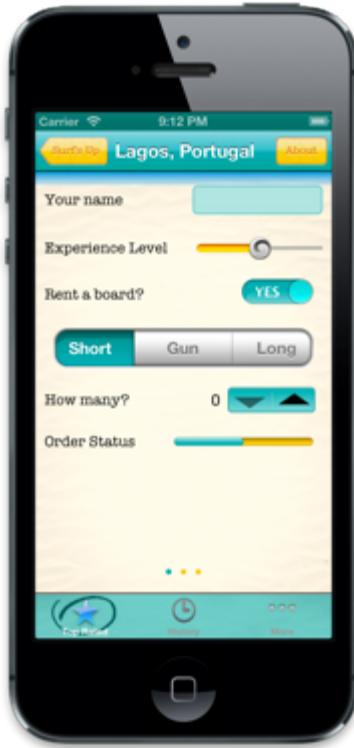


In this chapter, you're going to continue and learn about customizing the remaining elements: the `UITableView` and the `UINavigationBar` title image. Most of this tutorial is focused around customizing the `UITableView`, by adding custom cells, layout, and artwork.

Even if you're already familiar with customizing `UITableViews`, you might want to read this chapter, because it covers some new APIs introduced in iOS 5 and 6 that you may not be familiar with.

Later on in the chapter you're going to customize a popover controller. In order to do this, you have to transition Surf's Up to a Universal Binary. To keep the focus purely on UIKit Customization, I've done that work for you already.

In the resources for this chapter, you'll find a folder named **Surf's Up Starter**. Open up the Xcode project, and build and run the app in the iPhone Simulator. You'll see a similar project to the one you had before:



The code and supporting file structure, however, is quite different.

You'll notice that you have some additional groups, and in each of those groups, there are now files with `_iPad` & `_iPhone` suffixes to reflect subclasses and storyboards specific to the target device. This is now a universal app, capable of running on the iPhone and iPad.

Without further ado, let's surf the iOS customization wave and dive into finishing this app!

Customizing the UINavigationBar Title

In the previous chapter, you customized parts of the navigation bar via the appearance proxy, but you still need to add the title graphic. Let's do that first, then you'll proceed to customize the **UITableView**.

The title graphic can actually be placed in a **UIImageView** and set as the **titleView** of your primary view controller's navigation item. This sounds more complicated than it actually is, so let's dive right into code to see what it looks like.

Open up **SurfsUpAppDelegate_iPhone.m** and find the `application:didFinishLaunchingWithOptions:` method. You give it a title, which is currently displayed in the center navigation bar, and in the back button in your detail view. Right after the line of code that sets the title, add the following line of code:

```
UINavigationController* navController =
    (UINavigationController*)[[UIApplication
        sharedApplication].windows[0] rootViewController];
UIViewController* rootVC = navController.viewControllers[0];
[[rootVC navigationItem] setTitleView:
    [[UIImageView alloc] initWithImage:
        [UIImage imageNamed:@"title.png"]]];
```

Repeat this for **SurfsUpAppDelegate_iPad.m**:

```
UISplitViewController* splitVC = (UISplitViewController*)
    [[UIApplication sharedApplication].windows[0]
        rootViewController];
UINavigationController* navController =
    splitVC.viewControllers[0];
UIViewController* rootVC = navController.viewControllers[0];
[[rootVC navigationItem] setTitleView:
    [[UIImageView alloc] initWithImage:
        [UIImage imageNamed:@"title.png"]]];
```

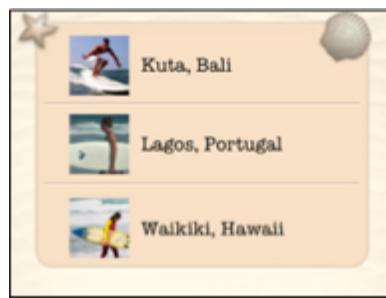
Build and run, and check out how it looks:



Excellent – I like where this is headed.

Customizing UITableView: Overview

To finish the customization of the primary view, turn your attention to **SurfsUpViewController**, a subclass of **UITableViewController**. Even though you're going to use the "plain" style for the **UITableView**, the artwork you're going to use to customize it will make it more closely resemble the "grouped" style. Here's a screenshot of the look trying to get for this table:



As you can see with this table, there are four possible cases to consider for this effect to be achieved:

1. A single row in the table view
2. The top row in a table with multiple results

3. A middle row in a table with multiple results
4. A bottom row in a table with multiple results

Although it's clear that each of these cases has its own background images, there are several characteristics that each of the cells have in common:

- **Inset surfing photo** associated with each individual trip
- **Font** used for the name of each surfing trip

You can take advantage of this shared scenario; you'll create one subclass of `UITableViewCell` in code and then associate four distinct cell types with that class. Finally, you'll employ automatic cell loading – a new feature in iOS 5 – to streamline the use of these custom cells in the table view controller.

Creating a Custom UITableViewCell

Let's start by creating a new group in Xcode. Right-click on the root "Surf's Up" folder, and select New Group. Name it Custom Cells. Then create a new file in this group with the **Objective-C class template**, named **CustomCell**, and a subclass of `UITableViewCell`.

Now select **CustomCell.h** – you're going to add two properties that each of the custom cells will rely upon. Specify the following properties in your header:

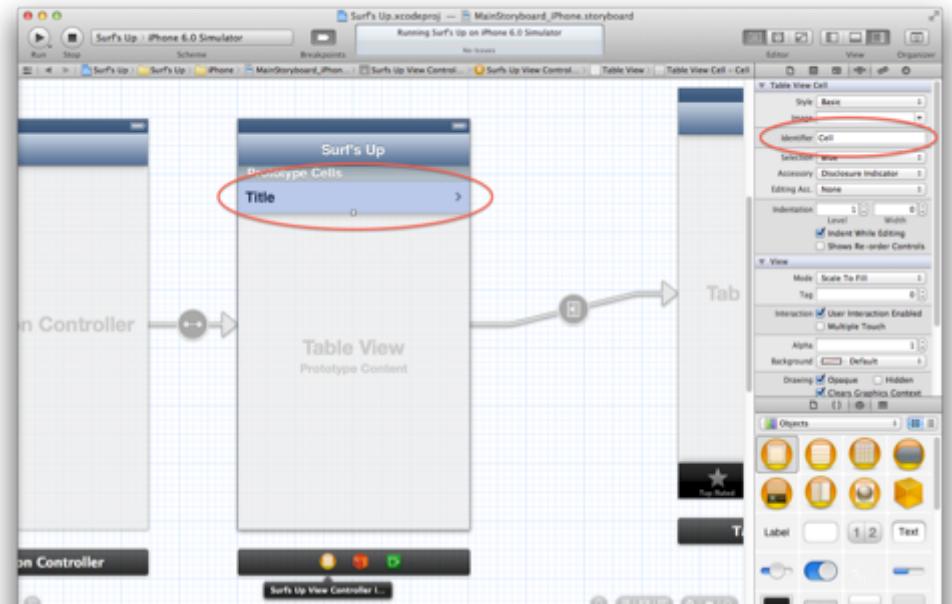
```
@property (nonatomic, strong) IBOutlet UIImageView *tripPhoto;  
@property (nonatomic, strong) IBOutlet UILabel *tripName;
```

These outlets will allow you to "wire in" the image view and label for the cell that you'll create in Interface Builder to this subclass of `UITableViewCell`. This will come in handy later when you want to refer to the image view and label in code!

Creating a UITableViewCell in Interface Builder

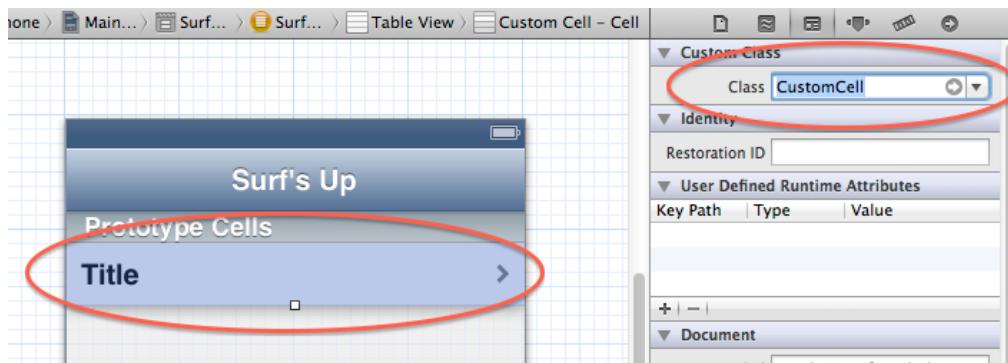
Next, you're going to create the `UITableViewCell` for the topmost row in the storyboard. This is something new that added in iOS 5 and makes it much easier to create custom cells. Instead of having to create custom cell programmatically (the worst) or create a cell in a separate xib and load it into the table through the controller (better), you can create prototype cells directly in the UITableView.

Getting started, open up the Utility Pane (i.e., the rightmost tab in the View toolbar) and drag a Table View Cell into the middle of the screen:

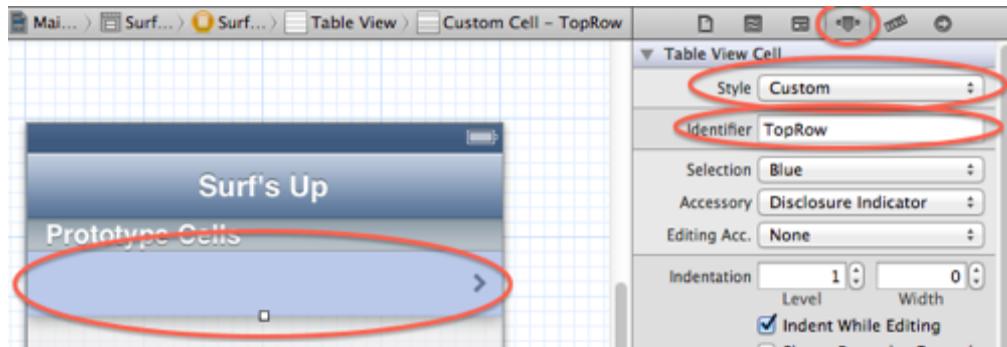


Open the **MainStoryboard_iPhone.storyboard** and find the Details View Controller. Here you can see the plain old normal tableview cell. Note that the identifier is simply "Cell". You only have one type of cell right now but you are going to add some new ones and this **Identifier** field is how you will create a new cell of the correct type.

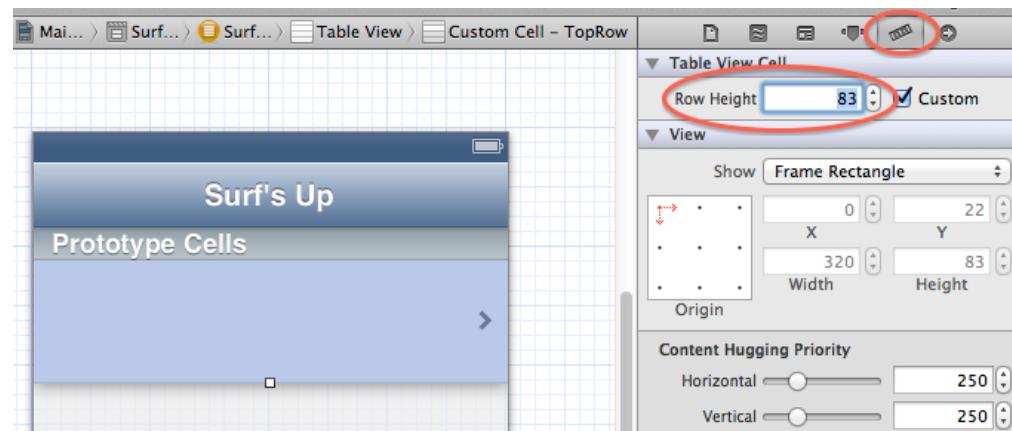
Select the cell, then select the third option on the right – the Identity Inspector. In the Custom Class input field, enter the name of class you created earlier, **CustomCell**.



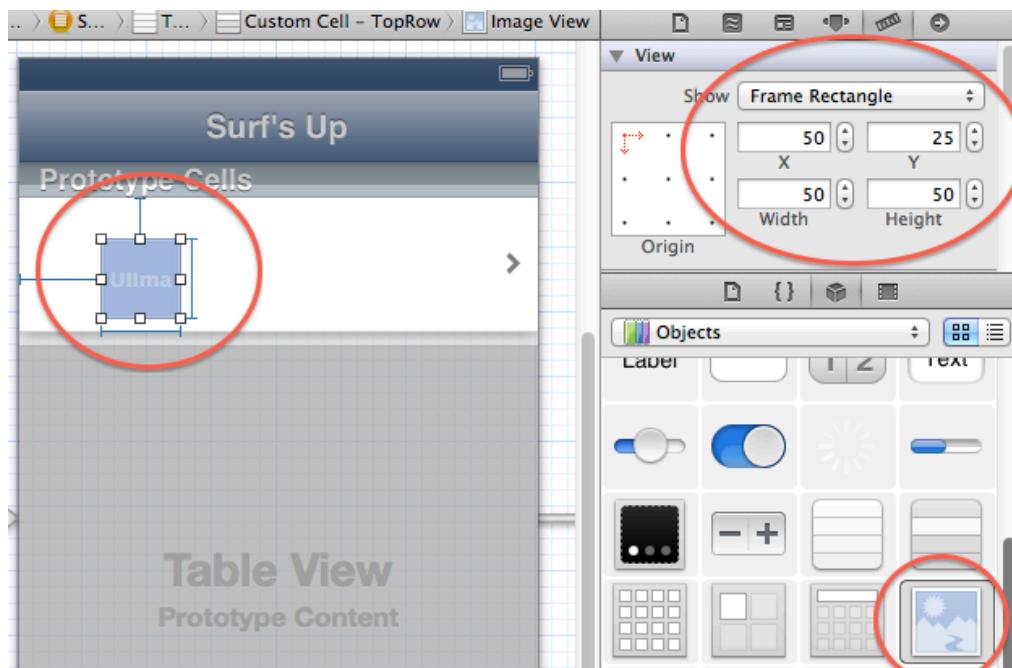
Now select the Attributes Inspector (the fourth tab). Change the Style to "Custom" and enter "TopRow" in the Identifier input field. This is the identifier your table view will use to manage cells of this type in the reuse queue.



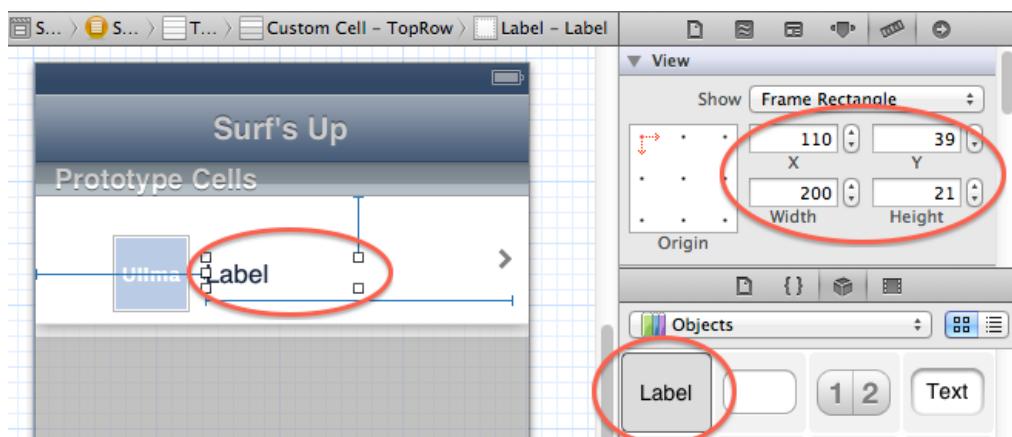
Now select the Size Inspector (fifth tab) and enter 83 as the height in pixels of the cell. This corresponds to the height of the custom images used to create the top row.



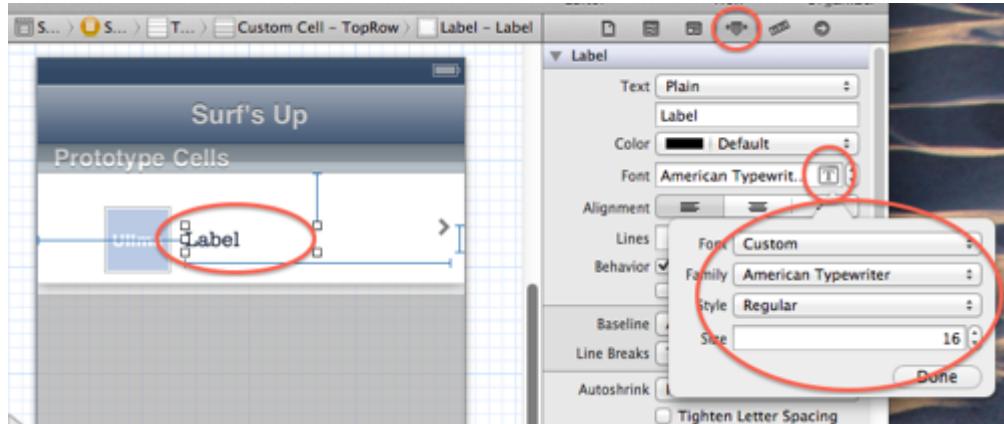
Next, drag an Image View from the library into the cell. You'll use this to display the surf trip photos. Select the image view in the left IB region and select the Size Inspector. Enter X = 50, Y = 25, W = 50, and H = 50. This may look a bit off-center vertically, but recall the top row has a starfish and a seashell that require a vertical offset.



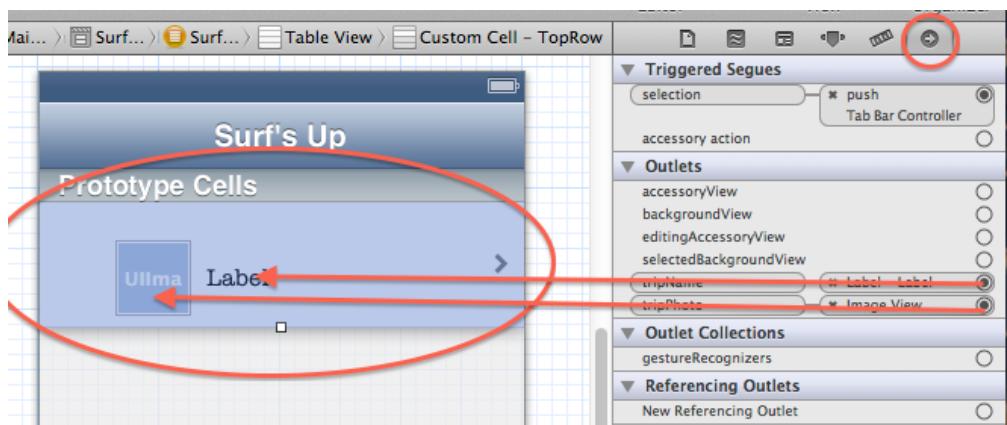
Next add your **UILabel** in a similar fashion - drag it from the Library into the cell, and set its position/size to X = 110, Y = 39, W = 200, H = 21.



With the **UILabel** still selected, select the Attributes Inspector. Set the Font to Custom, Family to American Typewriter, Style to Regular, and Size to 16.



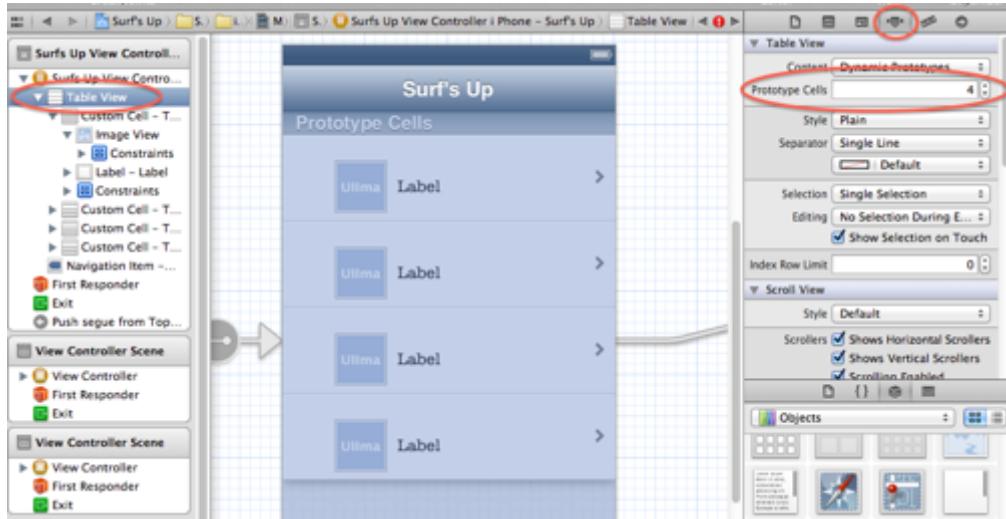
Finally, you need to connect the `IBOutlets` defined in your custom cell class to the corresponding elements in your cell. To do so, select the entire custom cell in the left pane and the Connections Inspector in the right. Drag the circle next to `tripName` to the label to connect it, then drag the circle next to `tripPhoto` to the image to connect it too.



Other Cells

After completing these steps for the top tableview cell, most of the work for the remaining cells amounts to “rinse & repeat” from the steps used to create the cell for the top row.

To create the different cell types, select the tableview, then increment **Prototype Cells** count to 4.

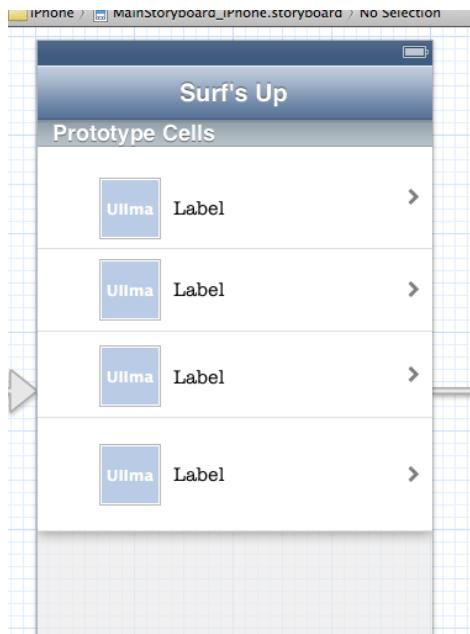


Look at that! It was even intelligent enough to copy the style from the last cell! So you just have to change the Identifier property and make the size adjustments.

The following table summarizes each of the unique parameters from the three remaining cell types:

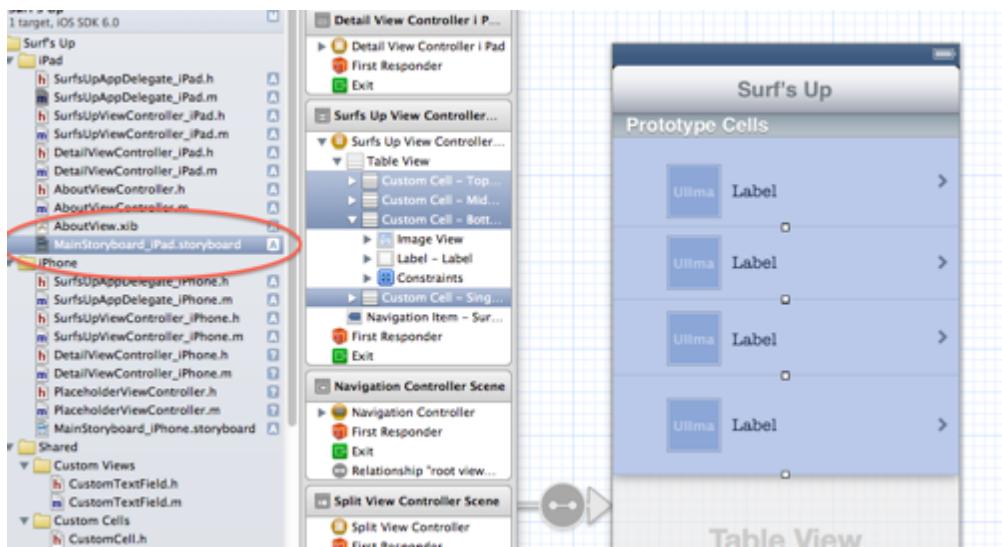
<i>Cell</i>	<i>Name/Reuse ID</i>	<i>Height</i>	<i>Image Frame (x, y, w, h)</i>	<i>Label Frame (x, y, w, h)</i>
Middle	MiddleRow	67	50, 8, 50, 50	110, 22, 200, 21
Bottom	BottomRow	70	50, 11, 50, 50	110, 26, 200, 21
Single	SingleRow	92	50, 21, 50, 50	110, 35, 200, 21

When you're all done, your table filled with the prototype cells should look something like this:

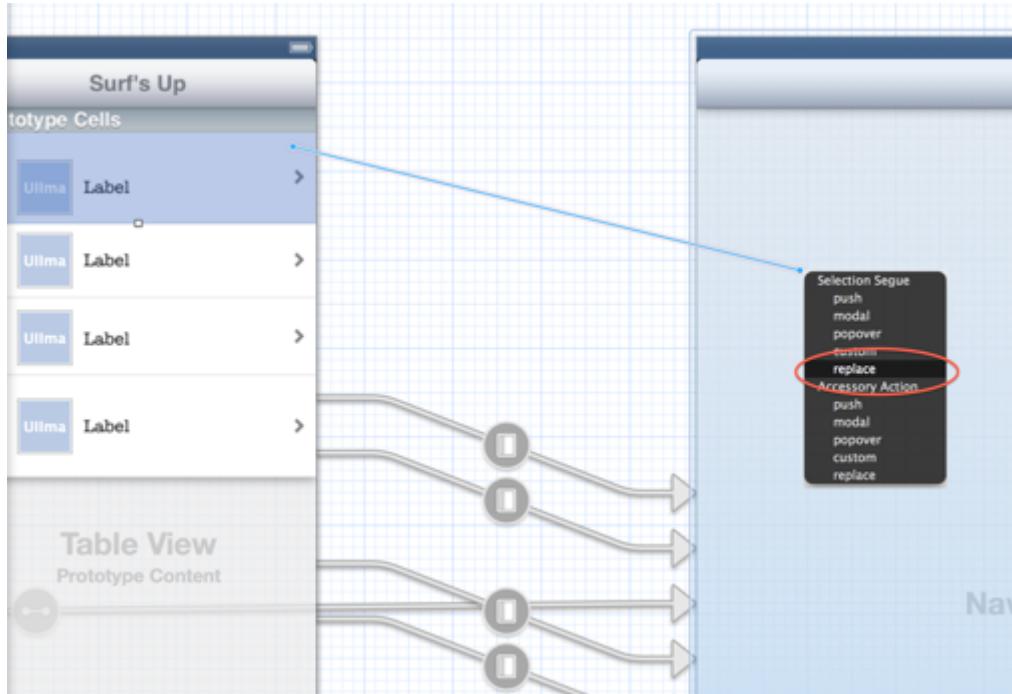


Last thing you need to do is connect the cells to the next view controller (the tabbar controller). So control-drag from a cell to the tabbar controller and select the **push** segue. Repeat for the other three cell types.

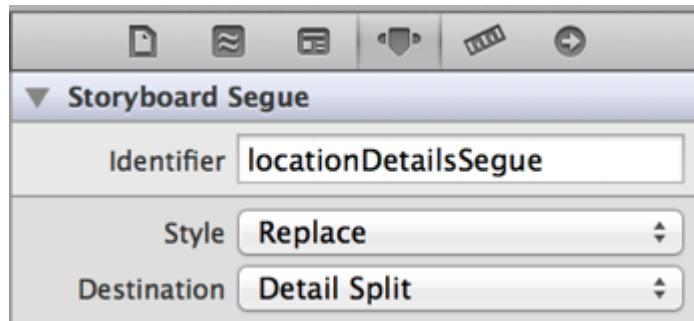
Because you are using different storyboards for the iPhone and iPad versions, you need to create these in the iPad storyboard as well. Luckily you can just copy and paste the cells instead of creating them again. Select one of the cells then hold down shift and click the other three to select them all. Copy them and switch to the iPad storyboard. Delete the regular old "Cell" tableview cell and then select the tableview and paste. Viola!



One thing that you do have to do is recreate the segue so that the app actually does something when you tap on one of the cells. Control-drag from a cell to the navigation controller (just to the right) and select the **Replace** segue.



Then select the segue (arrow part connecting the two View Controllers) and change the Identifier to **locationDetailsSegue**. The other defaults should be fine.



Repeat these steps for the other three cell types.

Using the Custom UITableViewCells in the UITableViewController

So you've created four custom cells and one custom class. Now it's time to integrate it with your table view controller. Open up **SurfsUpViewController.m** and make the following modifications:

```
// Add to top of file
#import "CustomCell.h"
```

```
// Add constants for each of our reuse identifiers, right after //
the imports (these are the cell identifiers)
NSString * const REUSE_ID_TOP = @"TopRow";
NSString * const REUSE_ID_MIDDLE = @"MiddleRow";
NSString * const REUSE_ID_BOTTOM = @"BottomRow";
NSString * const REUSE_ID_SINGLE = @"SingleRow";
```

Now you'll turn your attention to the logic required to identify which custom cell to use. In **SurfsUpViewController.m**, add the following method:

```
- (NSString *)reuseIdentifierForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    NSInteger rowCount = [self tableView: [self tableView]
                                numberOfRowsInSection:0];
    NSInteger rowIndex = indexPath.row;

    if (rowCount == 1)
    {
        return REUSE_ID_SINGLE;
    }

    if (rowIndex == 0)
    {
        return REUSE_ID_TOP;
    }

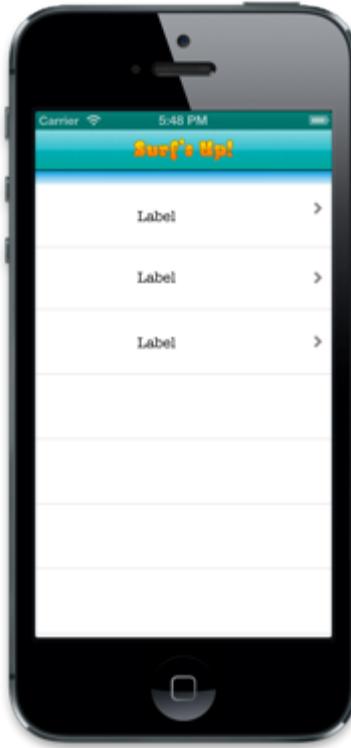
    if (rowIndex == (rowCount - 1))
    {
        return REUSE_ID_BOTTOM;
    }

    return REUSE_ID_MIDDLE;
}
```

And replace **tableView:cellForRowAtIndexPath:** with this:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *reuseID =
        [self reuseIdentifierForRowAtIndexPath:indexPath];
    UITableViewCell *cell = [[self tableView]
        dequeueReusableCellWithIdentifier:reuseID];
    return cell;
}
```

This code just checks the current row in the table view and compares it to the total number of elements. Based on this information, you know whether you should display the top, middle, bottom, or "single row" case. You haven't evaluated your progress in a while, so compile and run to see where you're at:



Wow – it appears that you've actually regressed! You don't have your photos & titles, nor do you have the custom background images. If it's any consolation, each of the first three rows does appear to have a different cell height. This is encouraging, as it suggests that you are returning the proper reuse identifiers for each cell.

So, what's wrong? Well, when you replaced the old code, you actually removed your cell configuration logic. You lost your specification of the trip photo and the naming of the trip. Moreover, IB doesn't appear to let you set the `backgroundView` and `selectedBackgroundView` properties, so you need to create `UIImageViews` that use your custom cell background images.

Let's work on that now. Create a new method `configureCell:forRowAtIndexPath:` and begin by adapting your existing methods for determining the appropriate trip photo and name to use for a given row. Add the following method to the file:

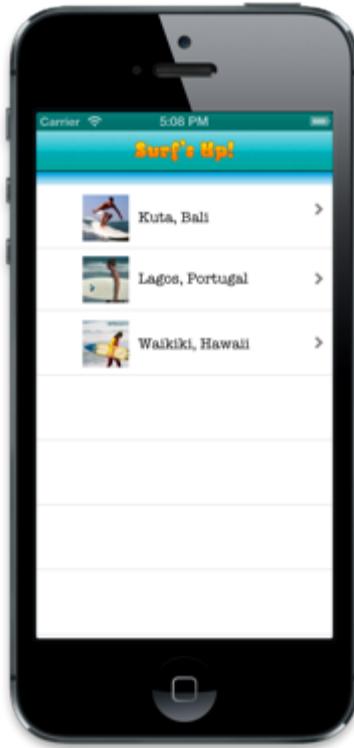
```
- (void)configureCell:(CustomCell *)cell
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    [[cell tripPhoto] setImage:
        [self tripPhotoForRowAtIndexPath:indexPath]];
```

```
[[cell tripName] setText:  
    [self tripNameForRowAtIndexPath:indexPath]];  
}
```

Now you need to call that method from `tableView:cellForRowAtIndexPath:`. Add the following line immediately after you've dequeued a cell for use / reuse:

```
[self configureCell:(CustomCell *)cell  
    forRowAtIndexPath:indexPath];
```

Compile and run, and you'll see that you're back to where you were originally:



You're using custom cells, but they aren't very good looking yet! So let's take care of that next.

Beautifying UITableViewCells

To make the cells good looking, you need to incorporate the background images for each of the respective cells.

To do that, you're going to create two new methods: one will return the background image for the normal state and one will return the background image for the selected state. These methods will accept an index path as input, then instantiate a

`UIImageView` stretched from the state your designer provided it to properly fit the size of the table view cells.

Add the code for this right above `configureCell`:

```
- (UIImage *)backgroundImageForRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *reuseID = [self
reuseIdentifierForRowAtIndexPath:indexPath];
    if ([REUSE_ID_SINGLE isEqualToString:reuseID] == YES)
    {
        UIImage *background = [UIImage
 imageNamed:@"table_cell_single.png"];
        return [background
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 43.0, 0.0,
64.0)];
    }
    else if ([REUSE_ID_TOP isEqualToString:reuseID] == YES)
    {
        UIImage *background = [UIImage
 imageNamed:@"table_cell_top.png"];
        return [background
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 43.0, 0.0,
64.0)];
    }
    else if ([REUSE_ID_BOTTOM isEqualToString:reuseID] == YES)
    {
        UIImage *background = [UIImage
 imageNamed:@"table_cell_bottom.png"];
        return [background
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 34.0, 0.0,
35.0)];
    }
    else // REUSE_ID_MIDDLE
    {
        UIImage *background = [UIImage
 imageNamed:@"table_cell_mid.png"];
        return [background
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 30.0, 0.0,
30.0)];
    }
}

- (UIImage *)selectedBackgroundImageForRowAtIndexPath:(NSIndexPath *)indexPath
```

```
{  
    NSString *reuseID = [self  
reuseIdentifierForRowAtIndexPath:indexPath];  
    if ([REUSE_ID_SINGLE isEqualToString:reuseID] == YES)  
    {  
        UIImage *background = [UIImage  
 imageNamed:@"table_cell_single_sel.png"];  
        return [background  
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 43.0, 0.0,  
64.0)];  
    }  
    else if ([REUSE_ID_TOP isEqualToString:reuseID] == YES)  
    {  
        UIImage *background = [UIImage  
 imageNamed:@"table_cell_top_sel.png"];  
        return [background  
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 43.0, 0.0,  
64.0)];  
    }  
    else if ([REUSE_ID_BOTTOM isEqualToString:reuseID] == YES)  
    {  
        UIImage *background = [UIImage  
 imageNamed:@"table_cell_bottom_sel.png"];  
        return [background  
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 34.0, 0.0,  
35.0)];  
    }  
    else // REUSE_ID_MIDDLE  
    {  
        UIImage *background = [UIImage  
 imageNamed:@"table_cell_mid_sel.png"];  
        return [background  
resizableImageWithCapInsets:UIEdgeInsetsMake(0.0, 30.0, 0.0,  
30.0)];  
    }  
}
```

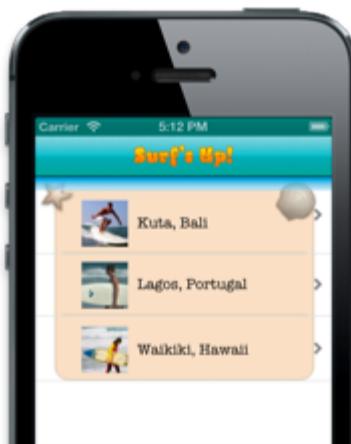
Because you've applied your table logic to determine the reuse identifier, you rely on it to load the appropriate image for the corresponding cell. If you look at the images, however, each of them is narrower than the full width of the screen. Once you've loaded your image, you resize your image by specifying `UIEdgeInsets` – these essentially define the portion of the image that is not stretched. Because the height of your cells matches that of your images, you simply set your top & bottom insets to 0.

With these methods implemented, add the code to incorporate them at the bottom of `configureCell:forIndexPath:` method like so:

```
CGRect cellRect = [cell frame];
UIImageView *backgroundView = [[UIImageView alloc]
                               initWithFrame:cellRect];
[backgroundView setImage:[self
                     backgroundImageForRowAtIndexPath:indexPath]];
[cell setBackgroundView:backgroundView];

UIImageView *selectedBackgroundView =
    [[UIImageView alloc] initWithFrame:cellRect];
[selectedBackgroundView setImage:[self
                     selectedBackgroundImageForRowAtIndexPath:indexPath]];
[cell setSelectedBackgroundView:selectedBackgroundView];
```

Compile and run again, and you should see the following:



Success! Well, sort of. Your cells now include the custom graphics, and subsequently mimic the “grouped” style. Unfortunately, however, you have some residual artifacts from creating the controller in the “plain” style.

Customizing the Table View Background

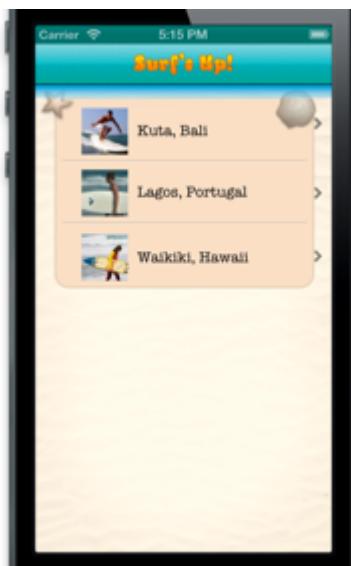
The last of your customizations address the remnants of creating the table with “plain” look & feel. You’ll tidy this up in `viewDidLoad`. Add the following lines to your method:

```
[[self tableView] setSeparatorStyle:
UITableViewCellSeparatorStyleNone];
[[self tableView] setBackgroundView:
[[UIImageView alloc] initWithImage:
```

```
[UIImage imageNamed:@"bg_sand.png"]];
```

The first line removes the separator lines that separate your individual table view cells. The second line introduces the sandy background. As with the navigation bar before, you're simply creating a `UIImageView` with your background graphic, and then designating it as the background view for the table.

Compile and run one more time, and you'll see the following:



Order has been restored! You've taken advantage of a number of iOS 5 features so far: you've customized the navigation bar, created four unique table view cells, and leveraged automatic cell loading to render the table view.

Customizing a Popover



In earlier stages of this chapter, you leveraged the new features of iOS 5 and 6 to create an iPhone app with a custom look & feel. You may recall that your detail

screen from Part 1 exposed an About button. Admittedly, it's a bit contrived to have an About button on the detail screen, but that aside, it's less excusable for your button to serve as a "road to nowhere".

So in this part of the chapter, you'll use the About button to present a custom popover. If you've used iBooks on the iPad you've seen how nice an authentic appearance can be. Popovers were one of the most significant new UI elements introduced with the iPad. The color of their borders, however, were like Model T cars.

"Any customer can have a car painted any colour that he wants so long as it is black."

-- Henry Ford

While it was certainly possible to create a popover that mimicked the look of the iOS standard popovers, it was difficult to match the *feel* of the popovers.

Fortunately, Apple has heard our pleas for customization, and answered them in iOS 5.

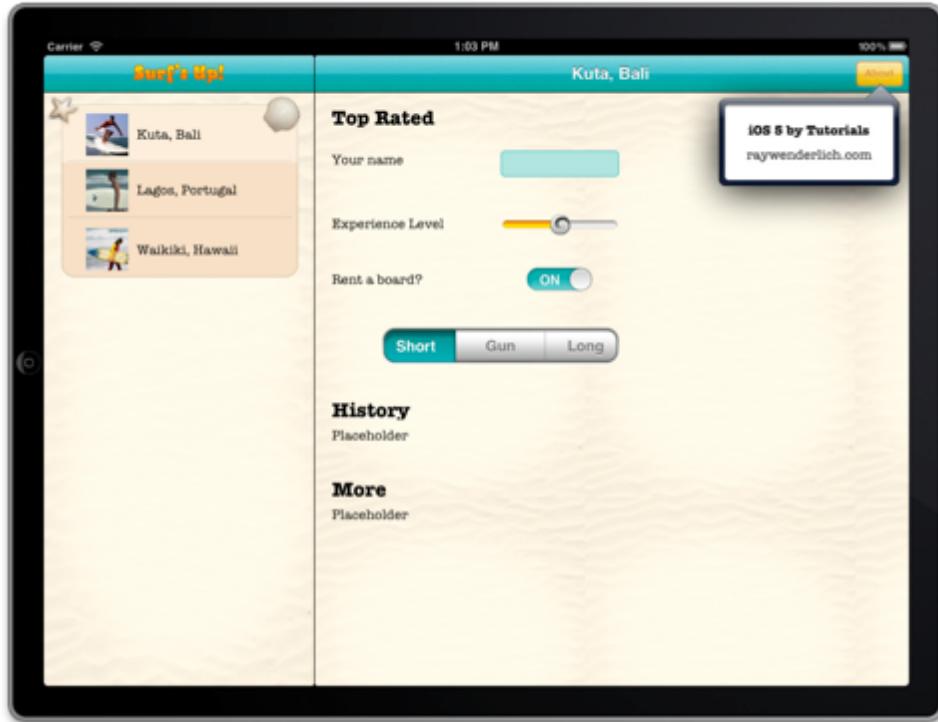
Unfortunately, however, popovers are still limited to iPads. If you attempt to present a popover on the iPhone, it may appear to "work" at compile-time, but at run time you will be presented with the following message:

```
-[UIPopoverController initWithContentViewController:] called when  
not running under UIUserInterfaceIdiomPad.'
```

In portrait, the detail view takes advantage of the iPad's increased screen real estate, "flattened" from three tabs to a single screen. You'll also note that the **UINavigationBar** adopts the look & feel you customized earlier in the tutorial – isn't the appearance proxy wonderful?

Build and run the app in the iPad Simulator. As you know, you'll need to select the iPad Simulator from the Scheme (see below), and then press Cmd-R as before.

You should see something like the following:



Fortunately, your custom table view still looks pretty good as the master view. The “model” layer is admittedly shallow for in this example – selecting a trip in the master view changes the title of the detail view controller rather than pushing a new view controller onto the navigation controller’s stack.

If you look in the project, there are new files for your newly introduce popover:

- **AboutViewController.h & AboutViewController.m** – the view controller used to present your popover
- **AboutView.xib** – the view your popover will present
- A resizable image named **popover_stretchable.png**

Unfortunately, however, you have the same black border around your popover. This throws off the surfing vibe you worked so hard to create previously. So let’s tidy that up.

Customizing the Popover

There are two facets of customizing your popovers. First, you must create an object to render that view, a subclass of type `UIPopoverBackgroundView` that’s new to iOS 5. Second, you must set a value for the `popoverBackgroundViewClass`, a new property of `UIPopoverController`.

First, you’ll create a subclass of `UIPopoverBackgroundView` to serve as the background for your About view. Create a new file with the **Objective-C class**

template, named **AboutBackgroundView**, and a subclass of **UIPopoverBackgroundView**.

Note it may be necessary to add the following import statement to **AboutBackgroundView.h** to successfully build the project after creation:

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

If you've done any kind of customization before, you may think to yourself that this is where you proceed to override **drawRect:: UIPopoverBackgroundView**, however, requires a bit more consideration. As with all framework code, the reader is encouraged to consult Apple's documentation by clicking Help\Documentation and API Reference.

To sum up the API reference, there are few takeaways to be mindful of:

"The background contents of your view should be based on stretchable images.

The images you use for your popover background view should not contain any shadow effects. The popover controller adds a shadow to the popover for you."

You must override two methods – **arrowOffset** and **arrowHeight** – to describe the geometry of the popover arrow. The fact that these are static methods implies that these values should not change.

You must describe the geometry of the content view in relation to its container, the popover. This is also a static method, **contentViewInsets**.

Finally, you must override properties describing the **arrowOffset** and **arrowDirection**. The mutability of these properties implies that they can be changed, but given the fixed position of your "anchoring" About button, these will be fixed for your purposes.

The backing image for your popover, **popover_stretchable.png**, looks like this.



Once you're finished, you'll have a nice teal border with a lighter turquoise background. To accommodate that, you must change the opaque background of the content view (initialized to white in **AboutView.xib**), you must add this implementation to **AboutBackgroundView.m**:

```
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
```

```

    if (self)
    {
        [self setBackgroundColor:[UIColor clearColor]];
    }
    return self;
}

```

Now you must override `drawRect:` to create a resizable image and present it:

```

- (void)drawRect:(CGRect)rect
{
    UIEdgeInsets popoverInsets =
        UIEdgeInsetsMake(68.0f, 16.0f, 16.0f, 34.0f);
    UIImage *popover =
        [[UIImage imageNamed:@"popover_stretchable.png"]
         resizableImageWithCapInsets:popoverInsets];
    [popover drawInRect:rect];
}

```

This approach is similar to your earlier use of resizable images with the appearance proxy. Again, the stretching relies on insets that your graphic designer may provide you. In this case, the insets are as follows:

- **Top:** 68 pixels
- **Left:** 16 pixels
- **Bottom:** 16 pixels
- **Right:** 34 pixels

Your static methods should be implemented as follows:

```

+ (CGFloat)arrowBase
{
    return 26.0f;
}
+ (CGFloat)arrowHeight
{
    return 16.0f;
}
+ (UIEdgeInsets)contentViewInsets{
    return UIEdgeInsetsMake(40.0f, 6.0f, 8.0f, 7.0f);
}

```

Finally, you must override properties as follows:

```

- (void)setArrowDirection:(UIPopoverArrowDirection)direction
{ // no-op }
- (UIPopoverArrowDirection)arrowDirection {return
UIPopoverArrowDirectionUp; }

```

```
- (void)setArrowOffset:(CGFloat)offset { // no-op }
- (CGFloat)arrowOffset {return 0.0f; }
```

Now that you've created your custom background view, you need to instruct your popover to use that view when it is presented. To do so, first import `AboutBackgroundView.h` at the top of `DetailViewController_iPad.m`:

```
#import "AboutBackgroundView.h"
```

Then add the following line to `viewDidLoad`:

```
[[self aboutPopover] setPopoverBackgroundViewClass:
    [AboutBackgroundView class]];
```

Build and run – and voila!

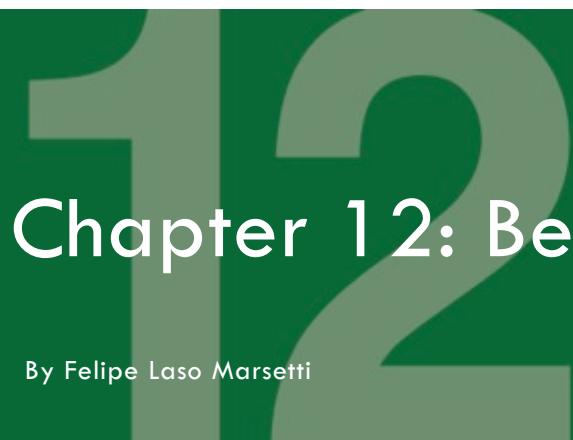
Where To Go From Here?

It's been quite a ride! You've customized UI elements on both iPhone & iPad interfaces, and you've used a number of different iOS 5 and 6 techniques to do so:

- Customization via Interface Builder
- Customization of `UIView` instances
- Customization by subclassing `UIView`
- Customization via the appearance proxy (`UIAppearance` protocol), and overriding the appearance proxy when necessary.
- Customization by subclassing `UIPopoverBackgroundView`

In these chapters you learned about many of the new user interface customization capabilities in iOS 6, but if you're curious to learn even more, check out Chapter 22 in *iOS 6 by Tutorials*, "What's New with User Interface Customization."

Hopefully you've had as much fun as I've had, and I look forward to seeing your customized UIKit apps in the future!



Chapter 12: Beginning Twitter

By Felipe Laso Marsetti

These days, social networks are a huge part of our daily lives. Not only do we access social networks via their dedicated websites like twitter.com or facebook.com, but we also find social features in apps, websites, blogs, video games, and more.

Adding some social features into your apps can really help you increase the popularity of your app, help you identify and retain customers, and can boost the polish and added value of your app.

Until iOS 5 and iOS 6, adding social features into apps was a pain. Not only did you have to use a different API for each social network, users had to constantly log into each one of them for every app they used.

I can't remember how many times I've had to log into Facebook or Twitter within a game or app. It can get quite tedious both as a developer and as a user to repeat the same thing over and over again for every application, up to the point where users won't even bother because they don't want to have to log in again.

Thankfully for you, Apple has taken a huge step forward in this regard by having Twitter natively incorporated in iOS 5 (and Facebook and Sina Weibo in iOS 6)! Now all a user needs to do is log into Twitter, Facebook, or Sina Weibo once and each app can make use of your accounts stored on the device.

Note: Since the focus of this book is on iOS 5, this book focuses on the iOS 5 way of doing things through the Twitter framework.

However, iOS 6 deprecated the Twitter Framework in favor of the all-new Social Framework. If you are interested in learning the changes and how to interact not only with Twitter but also with Facebook or Sina Weibo, please refer to Chapters 11 and 12 in *iOS 6 by Tutorials*, Beginning and Intermediate Social Framework.

You might wonder “why bother reading these chapters, if the Twitter framework is deprecated?!” Well, there are two reasons:

- 1) **Supporting older versions of iOS.** The social framework is new to iOS 6, so if you want to add Twitter support in iOS 5, you’d have to use the Twitter framework instead.
- 2) **Learning how to make a Twitter client.** The next chapter covers detailed information on how to make a simple Twitter client using API calls to Twitter’s interface. Most of the information in this chapter still applies to the new Social Framework, and might be useful if you want to do some advanced things with Twitter’s APIs.

How Does It Work?

iOS 5 includes several ways to interact with Twitter. The simplest, and possibly the one you will most likely implement, is the `TWTweetComposeViewController`. That name is quite a handful so we will affectionately call it “Tweet Sheet” just as Apple does.

In this chapter you will see that the tweet sheet is very easy to implement. With just a couple of lines of code you can have a full tweet composer within your app! You don’t have to worry about contacting the Twitter backend, handling user logins, or anything like that.

To do this, you just use a built-in class called the `TWTweetComposeViewController`. It’s similar to using the `MFMailComposeViewController` and the `MFMessageComposeViewController` for mail and SMS, respectively.

Here is a code snippet for creating and presenting the `TWTweetComposeViewController`:

```
if ([TWTweetComposeViewController canSendTweet])  
{  
    TWTweetComposeViewController *tweetSheet =  
    [[TWTweetComposeViewController alloc] init];  
    [tweetSheet setInitialText:@"Initial Tweet Text!"];  
    [self presentModalViewController:tweetSheet animated:YES];  
}
```

All you do is determine whether the device can send tweets, create an instance of the tweet sheet, attach any links or images, put some initial text and present it modally, that’s it! All within Xcode and through the use of Objective-C.

In fact it's so easy, that if you want you can stop reading here and skip over to the next chapter, where you'll learn a more advanced API you can use to directly communicate with the Twitter API and get everything from a user's profile to his or her timeline. But if you want to try out this "simple tweet" capability yourself with a simple project, keep reading!

Overview

In this introductory Twitter chapter you'll learn the use of the `TWTweetComposeViewController` (i.e. the "tweet sheet") that will enable you to tweet any text, image or link you want from within your applications. It looks something like this:



The advantage of using the tweet sheet is that it's built right onto iOS. By using it you get the following benefits:

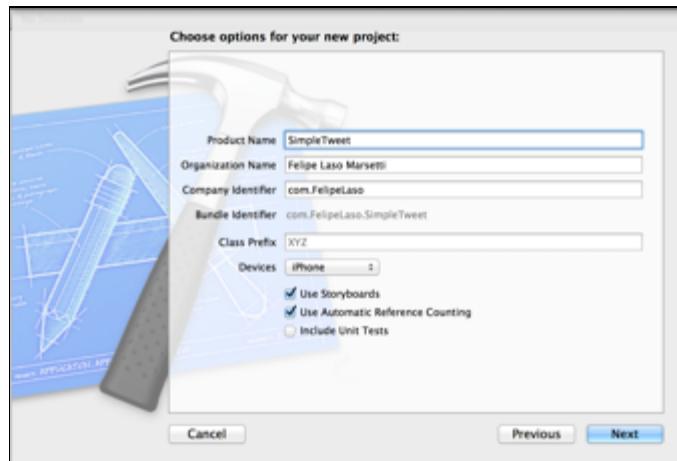
- A standard interface throughout the OS
- Automatic use of the user's system Twitter account
- Automatic check for a tweet less than 140 characters long
- Easy image and link attachments
- Easy to program, no need to implement OAuth or connect to the Twitter backend

As you can see you have lots of advantages and incentives to use this and, being that it's so simple, there's no excuse not to include Twitter functionality in your applications!

Getting Started

Ok fellow programmers, it's now time to get your fingers typing and creating an awesome Twitter enabled app. In this example you're going to create a simple app that will allow the user to tweet whatever text they like, and even include images or links within their tweet.

Create a new project with Xcode with the **iOS\Application\Single View Application** template. Enter **SimpleTweet** for the product name, set the device family to **iPhone**, and make sure that **Use Storyboard** and **Use Automatic Reference Counting** are checked (leave the other checkbox unchecked).



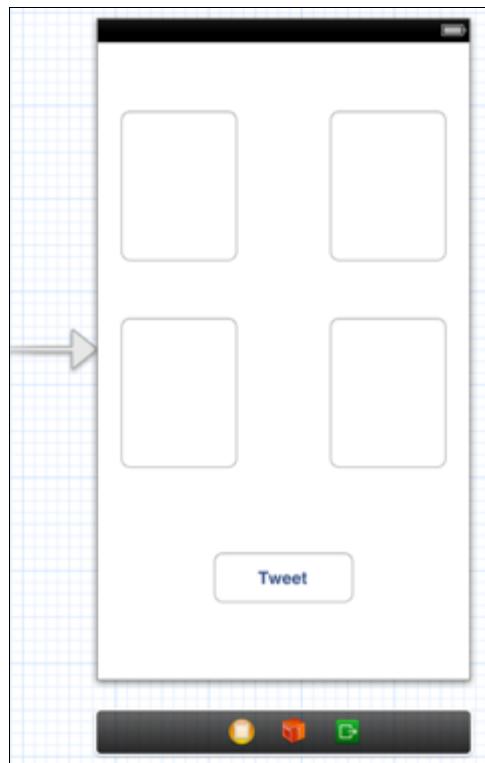
Go ahead and click next one more time and select a location where you want to save your project.

Now that you have your project created let's discuss a bit of what it's going to do. Your app will allow the user to enter the text for their tweet as well as show some buttons for including an optional image and link on their tweet.

You are only going to support Portrait orientation so you need to set that up within your project settings. In your Project Navigator select the SimpleTweet project and make sure you select the SimpleTweet target inside of it, go to the Summary tab and deselect all orientations except for Portrait:



Now open up `MainStoryboard.storyboard` and add 4 `UIButtons` to the view controller as follows:



You have four large buttons that the user will be able to toggle in order to add an image and link to their tweet.

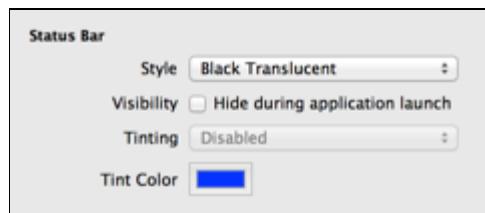
You're going to make the buttons have an image from 4 different tutorials at raywenderlich.com. Drag the images from the resources folder from this chapter into your project, and set up the buttons to use the images (and add some labels too) as shown below:



To make this interface, simply:

- Set the background image of the buttons to the appropriate images, and set the button type to Custom
- Add 5 labels to show what each button corresponds to, and set their text color to White
- Change the background color to a dark gray
- Make the Tweet button's text black

Additionally, for looks make the status bar a translucent black by changing the setting in your project's target:



Compile and run and make sure everything looks ok so far. Now onto the implementation!

Making Some Connections

Next you need to make some connections from your storyboard to the view controller. Open **MainStoryboard.storyboard** and make sure the Assistant Editor is visible, and displaying **ViewController.m**.

Control-drag from each of the four labels to the class extension, and connect them to Outlets named **button1Label**, **button2Label**, **button3Label**, and **button4Label**.

Similarly, control-drag from each of the four buttons to the class extension, and connect them to Actions named **button1Tapped:**, **button2Tapped:**, **button3Tapped:**, and **button4Tapped:**.

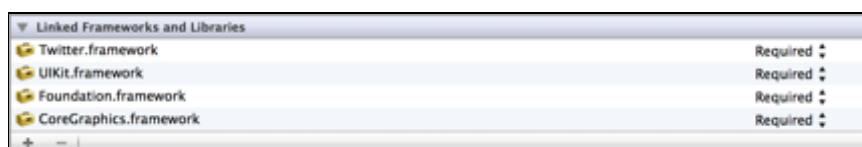
Finally, control-drag from the "Tweet" button to the class extension, and connect it to an action named **tweetTapped:**.

When you are done, your class extension should like this:

```
@interface ViewController ()  
  
@property (weak, nonatomic) IBOutlet UILabel *button1Label;  
@property (weak, nonatomic) IBOutlet UILabel *button2Label;  
@property (weak, nonatomic) IBOutlet UILabel *button3Label;  
@property (weak, nonatomic) IBOutlet UILabel *button4Label;  
  
- (IBAction)button1Tapped:(id)sender;  
- (IBAction)button2Tapped:(id)sender;  
- (IBAction)button3Tapped:(id)sender;  
- (IBAction)button4Tapped:(id)sender;  
- (IBAction)tweetTapped:(id)sender;  
  
@end
```

Awesome! Now implement these methods to send out some tweets!

In order for you to use the `TWTweetComposeViewController` you need to add the Twitter framework to your project. Select your project in the project navigator and then the **SimpleTweet** target. Go to the **Build Phases** tab and click on the **+** button inside the **Link Binary With Libraries** section, on the window that appears navigate to the **Twitter.framework** file and click **Add**:



Next, open **ViewController.m** and add the following import at the top of the file:

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

That's all you need in order for your file to see the Twitter API, now add some code so you display the tweet sheet when the user taps the Tweet button. Go to your `tweetTapped:` method add the following code:

```
- (IBAction)tweetTapped:(id)sender
{
    if ([TWTweetComposeViewController canSendTweet])
    {
        TWTweetComposeViewController *tweetSheet =
        [[TWTweetComposeViewController alloc] init];
        [tweetSheet setInitialText: @"Tweeting from iOS 5 By
Tutorials! :)];
        [self presentModalViewController:tweetSheet animated:YES];
    }
    else
    {
        UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Sorry"
message:@"You can't send a tweet right now, make sure your device
has an internet connection and you have at least one Twitter
account setup"
delegate:self
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
        [alertView show];
    }
}
```

Yes, believe it or not that's all you need to do in order to send a tweet (mind you we haven't included any links or images), it doesn't get any easier than this! Let's go over the code you added to your `tweetTapped:` method.

First thing you do is check to see if you can send a tweet, you accomplish this by calling the `canSendTweet` class method on `TWTweetComposeViewController`. This method will return NO if the device cannot access the network or if the user hasn't setup a Twitter account yet.

If your application can send a tweet then all you do is create an instance of the `TWTweetComposeViewController`, use the `setInitialText:` method to load up the tweet sheet with some default text and present it modally. If you cannot send a tweet then you just show a simple alert view to provide the user with feedback.

Build and run your project, touch the Tweet button and this is what we get:



Awesome, huh? If you get the Alert dialog, be sure you have your Twitter account set up by loading the Settings app and selecting the Twitter category.

There is one thing worth mentioning and that is that the `TWTweetComposeViewController` has a completion handler property that you can use to pass in your own block of code once the tweet sheet is dismissed.

This is the signature for the completion handler's block:

```
void (^TWTweetComposeViewControllerCompletionHandler)(  
    TWTweetComposeViewControllerResult result);
```

You can see that the method provides you with a "result" parameter of type `TWTweetComposeViewControllerResult` that can be any of the following values:

- `TWTweetComposeViewControllerResultCancelled`: Indicates the user has cancelled composing a tweet
- `TWTweetComposeViewControllerResultDone`: Indicates that the user has finished composing a tweet. In order to set the completion handler, you can use the `completionHandler` property on a `TWTweetComposeViewController`. Here's an example (but you're not going to add this to your project since you don't need it):

```
TWTweetComposeViewController *tweetSheet =  
[ [ TWTweetComposeViewController alloc] init];  
    tweetSheet.completionHandler =  
^(TWTweetComposeViewControllerResult result){  
    if (result ==  
TWTweetComposeViewControllerResultCancelled)
```

```
{  
    // Cancelled the Tweet  
}  
else  
{  
}  
};
```

If you don't set a completion handler, by default it will dismiss the modal tweet sheet. Just keep in mind that if you do implement your own completion handler then you need to dismiss the tweet sheet yourself.

Adding Images and Links

Now let's implement the logic to allow the user to select one of the tutorials and have its image and link added to the tweet. Add the following properties to the class extension in **ViewController.m**:

```
@property (strong, nonatomic) NSString *imageString;  
@property (strong, nonatomic) NSString * urlString;
```

All you are doing here is creating two private string properties to store the image's name and the link to the tutorial website as well as a private method to set your labels' text color back to white.

Next, implement the `clearLabels` method to set the text color of each label to white:

```
- (void)clearLabels  
{  
    self.button1Label.textColor = [UIColor whiteColor];  
    self.button2Label.textColor = [UIColor whiteColor];  
    self.button3Label.textColor = [UIColor whiteColor];  
    self.button4Label.textColor = [UIColor whiteColor];  
}
```

Yup, that's all this private method will do for you.

The way you are going to implement this is as follows: when the user selects a tutorial you store its image name and link within your private properties and set the label's color to red in order to indicate the current selection.

If the user selects another tutorial then you just set all the labels back to white, store the new image and url and set its label to red. Add the following code for your button-tapped methods:

```
- (IBAction)button1Tapped:(id)sender
{
    [self clearLabels];
    self.imageString = @"CheatSheetButton.png";
    self.urlString =
@"http://www.raywenderlich.com/4872/objective-c-cheat-sheet-and-
quick-reference";
    self.button1Label.textColor = [UIColor redColor];
}

- (IBAction)button2Tapped:(id)sender
{
    [self clearLabels];
    self.imageString = @"HorizontalTablesButton.png";
    self.urlString = @"http://www.raywenderlich.com/4723/how-to-
make-an-interface-with-horizontal-tables-like-the- pulse-news-app-
part-2";
    self.button2Label.textColor = [UIColor redColor];
}

- (IBAction)button3Tapped:(id)sender
{
    [self clearLabels];
    self.imageString = @"PathfindingButton.png";
    self.urlString =
@"http://www.raywenderlich.com/4946/introduction-to-a-
pathfinding";
    self.button3Label.textColor = [UIColor redColor];
}

- (IBAction)button4Tapped:(id)sender
{
    [self clearLabels];
    self.imageString = @"UIKITButton.png";
    self.urlString = @"http://www.raywenderlich.com/4817/how-to-
integrate-cocos2d-and-uikit";
    self.button4Label.textColor = [UIColor redColor];
}
```

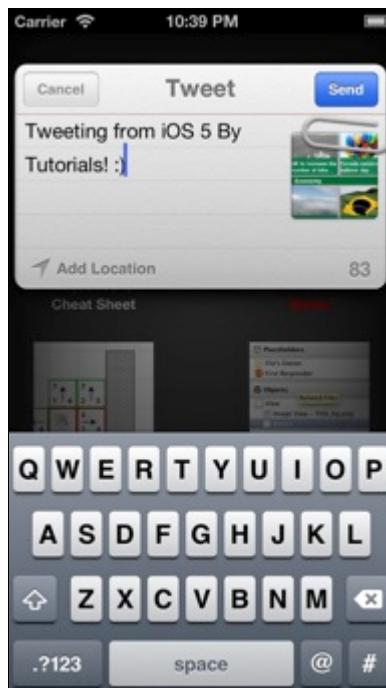
In each of the methods you just clear the labels, set the image string to the appropriate image, set the URL to the corresponding tutorial and change the label's text color to red. You could have implemented things a bit differently in order to avoid writing the same code in all 4 methods but since this example is very simple, you'll leave it at that.

Now go over to `tweetTapped` and add the following code right before the call to `[self presentViewController:...]`:

```
if (self.imageString)
{
    [tweetSheet addImage:[UIImage imageNamed:self.imageString]];
}
if (self.urlString)
{
    [tweetSheet addURL:[NSURL URLWithString:self.urlString]];
}
```

You just added two if statements to check whether you have an image and url, if you do then you just add them to your tweet sheet by using the `addImage:` and `addURL:` methods respectively.

Once again, this is all you need! Go ahead and run your project; this time make sure you select one of the tutorials and hit the Tweet button. This is what you get:



If you look at the tweet sheet you will notice a paper clip with 2 attachments, one is the link to your website and the other is the image you added. When the user sees the Tweet, they'll see two URLs in the tweet - one for the image, and one for the link. Try it out and see what it looks like!

And with that my friend, you are done!

Where To Go From Here?

As you can see it's very easy to integrate Twitter into your own applications and there is really no reason why you shouldn't do so.

With just a few lines of code you can give your apps an extra layer of polish and give them that highly appreciated social component. Whether it's to tweet a high score or share a picture from within your app, it's possible with the `TWTweetComposeViewController` class.

What's really beautiful and elegant about having native Twitter support is that you get a single, prevalent interface throughout iOS as well as the little details like character count and check, attachment images when included, location support and more.

Keep in mind that as of this moment URLs and images each take up 19 characters from your tweet, this could change in the future as the URL shortening services become saturated. A good habit to develop would be to check for tweet character length before attaching images or links.

You could try and experimenting with things a bit, perhaps dynamically setting the initial text of the tweet sheet depending on what tutorial was selected, or even include more images or links within your tweet.

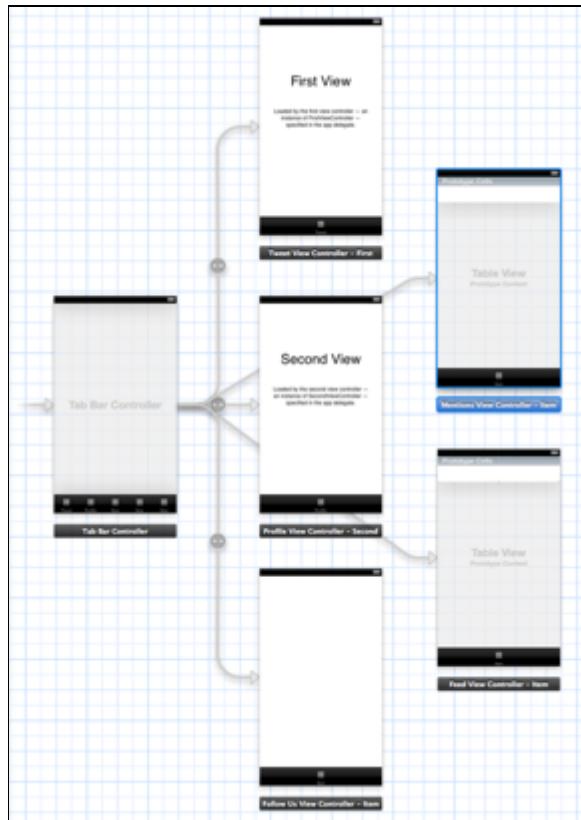
Have fun and think of unique ways to incorporate Twitter into your app because your users are going to love it!

If you want to do some more cool stuff with Twitter, keep reading the next chapter, where we'll cover how to use the Accounts framework as well as how to access Twitter in order to acquire information such as mentions, messages, a user's feed and more!

You can also look at the Beginning and Intermediate Social Framework chapters in *iOS 6 by Tutorials* to see how to use the new `SIComposeViewController` in and Social API in favor of the new deprecated Twitter Framework.

Chapter 13: Intermediate Twitter

By Felipe Lasso Marsetti



Just as you did earlier, you need to rename the Tab Bar Items for your new view controllers so they are properly titled in your tab bar. Select each view controller's Tab Bar Item and change their title in the Attributes Inspector, I've named mine Mentions, Feed and Follow Us:



I'd reorder the items on the tab bar to the following order (from left to right):

- Feed
- Mentions
- Tweet
- Follow Us
- Profile

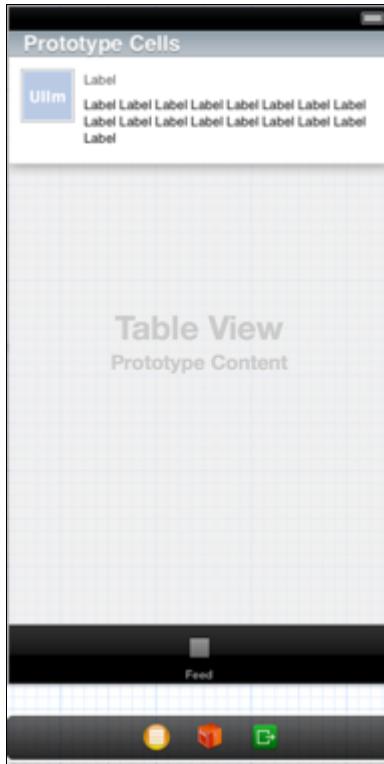
To do this you just need to click and drag on the tab bar buttons and arrange them in the position you like.

You're almost done customizing your storyboard - all you need is a custom cell for the Feed and Mentions view controllers. For the user's feed and mentions you want the table to show cells with the profile image of the person who sent the tweet, their username and the actual tweet contents.

You could use one of the cell styles already available but you wouldn't be able to show everything you want inside our cell, plus this is a great way of learning more about storyboards and how incredibly simple it is to creating custom cells in iOS 5.

Select the cell inside the Feed View Controller and go to the Attributes Inspector, in the Identifier field write `TweetCell` so you can create these custom cells from your code. Now go to the Size Inspector and change the row height to 90, make sure the Custom checkbox is selected. There's now a larger cell where you can place the custom elements you need.

From the Object Library drag two labels and an image view onto the cell, feel free to layout and customize it to your liking, this is what mine looks like:



You not only have to specify the row height within your cell but also within your table view, otherwise your cells would be cut off because the table view is setup to display smaller cells. Select the table view in your view controller and in the Size Inspector change the row size to 90 as well.

Since you want to use the same cell for the Mentions View Controller all you need to do is copy and paste our custom cell inside the mentions view controller's table view. Just select the current prototype cell inside the Mentions View Controller, delete it and then paste your custom cell. Don't forget to change the reuse identifier to `TweetCell`, and the table's row height to 90 pixels as well!

In order for you to be able to set the profile picture in your cell as well as the user name and tweet text, you need to create a subclass of `UITableViewCell` where you define the outlets for the cell.

Back in your Project Navigator right click the **RazeTweet** folder and select **New File**. Select the **Cocoa Touch\Objective-C Subclass** template, click Next, use `TweetCell` as the name of the class and in the Subclass of field just type in `UITableViewCell`. Click **Next** one more time, choose where to save the header and implementation files and click **Create**.

Now that you have your table view cell subclass let's add some outlets for your image and labels. Put this in the interface inside `TweetCell.m`:

```
@interface TweetCell : UITableViewCell
```

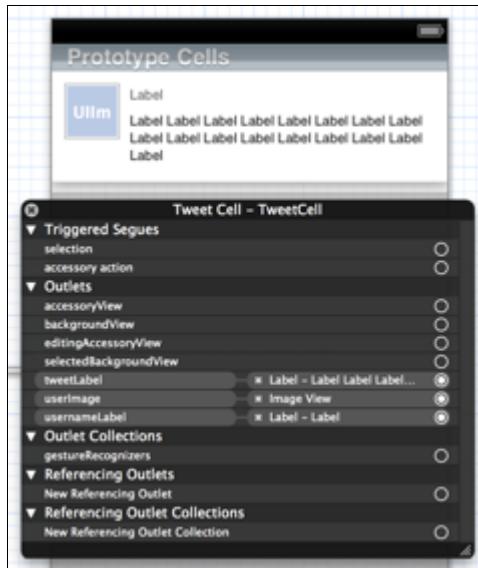
```
@property (strong, nonatomic) IBOutlet UILabel *tweetLabel;
@property (strong, nonatomic) IBOutlet UIImageView *userImage;
@property (strong, nonatomic) IBOutlet UILabel *usernameLabel;

@end
```

All you are doing here is declaring two `UILabels`, one for your tweet and one for the username, and a `UIImageView` for the tweeting user's profile image.

Switch back to **MainStoryboard.storyboard** one last time so you can assign the new `TweetCell` subclass to the custom cells you created earlier.

Inside the storyboard select the cell you prototyped earlier and go to the Identity Inspector, change the class name to `TweetCell`. Then connect the labels and image view to the `TweetCell` outlets by right clicking on each item (the labels and the image view) and dragging from "New Referencing Outlet" onto the cell itself, or to the cell in the Document outline, and connect it to the appropriate outlet:



Make sure you connect the image view to `userImage`, the top label to `usernameLabel`, and the bottom label to `tweetLabel`. Also make sure you do this process for both prototype cells (i.e. the one in the Feed View Controller as well as the Mentions View Controller).

If you run your application now you will notice that the tab bar has 5 buttons for your corresponding view controllers, each with the correct title, but in the Feed or Mentions View Controllers you can see taller rows but not your custom cell. Don't worry, you are going to implement each view controller one at a time and use actual data from Twitter.

Before doing any interaction with the Twitter API you need to retrieve the user's Twitter accounts, so let's do so in the next section.

Getting a user's account

You already covered the use of the Accounts framework at the beginning of the chapter so the code you write here should look familiar or at least make a lot of sense.

Open up **AppDelegate.h** and replace it with the following:

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

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) ACAccountStore *accountStore;
@property (strong, nonatomic) NSMutableDictionary *profileImages;
@property (strong, nonatomic) ACAccount *userAccount;
@property (strong, nonatomic) UIWindow *window;

@end
```

The first thing you do is import Accounts.h so you can use account related classes within the app delegate. Next you add properties for an ACAccountStore, ACAccount and an NSMutableDictionary of profile images. Notice how the properties are using ARC's new strong keyword, this will act similarly to a retain in case you are more familiar with manual reference counting.

Note: If you are having trouble understanding Automatic Reference Counting and the changes made in iOS 5 then please refer to Chapters 1 and 2, "Beginning and Intermediate ARC".

Remember how at the beginning of the chapter I mentioned that you should keep a single instance of the account store throughout the life of your app and how accounts retrieved from a particular store are bound to it? Well, storing them in the app delegate is the one way to do it. Whenever a class or view controller requires access to the account store or the user's Twitter account, you can just retrieve it from the app delegate.

The NSMutableDictionary will be used to store the profile images for the user's feed and mentions. You don't want to be fetching images every time the user changes view controllers or reloads their tweets.

What you'll do instead is first look for the images in the dictionary, and if you can't find them you'll download them from Twitter. Small optimizations like these can make your app quicker, consume less battery and seem more responsive to your users.

Switch to **AppDelegate.m**, and add this code to retrieve the user's Twitter accounts in the `applicationDidFinishLaunchingWithOptions:` method:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.accountStore = [[ACAccountStore alloc] init];
    self.profileImages = [NSMutableDictionary dictionary];

    ACAccountType *twitterType = [self.accountStore
accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierTwitter];
    [self.accountStore requestAccessToAccountsWithType:twitterType
withCompletionHandler:^(BOOL granted, NSError *error) {
        if (granted)
        {
            NSArray *twitterAccounts = [self.accountStore
accountsWithAccountType:twitterType];

            if ([twitterAccounts count])
            {
                self.userAccount = [twitterAccounts
objectAtIndex:0];
                [[NSNotificationCenter defaultCenter]
postNotification:[NSNotification
notificationWithName:@"TwitterAccountAcquiredNotification"
object:nil]];
            }
            else
            {
                NSLog(@"No Twitter Accounts");
            }
        }
    }];
}

return YES;
}
```

Let's go over the code added to the `applicationDidFinishLaunchingWithOptions:` method.

```
self.accountStore = [[ACAccountStore alloc] init];
self.profileImages = [NSMutableDictionary dictionary];
```

First you create a new instance of `ACAccountStore` and save it in the `accountStore` property you created in the app delegate. After that you just create an autoreleased

mutable dictionary that will be ready to store all the profile images you download from Twitter.

```
ACAccountType *twitterType = [self.accountStore  
accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierTwitter];
```

Before actually retrieving the user's Twitter accounts you must create an `ACAccountType` object to use with the account store, this is what you do up next by asking the account store to give you a twitter account type object using the `accountTypeWithAccountTypeIdentifier:` method.

Now that you created a mutable dictionary for the profile pictures, an account store and an account type object you are ready to ask the user for permission to user his or her Twitter accounts.

```
[self.accountStore requestAccessToAccountsWithType:twitterType  
withCompletionHandler:^(BOOL granted, NSError *error) {
```

To do this you must use the `requestAccessToAccountsWithType:withCompletionHandler:` method which takes two parameters: an account type object and a completion handler. Notice that you pass in the newly created `ACAccountType` object for the Twitter account type.

In the completion handler you receive a `BOOL` to let you know if you were granted access to the accounts and an `NSError` object in case there was a problem with the request. Using this boolean you write a simple if statement inside the completion handler to determine if you were granted access to the Twitter accounts or not.

```
if (granted)  
{  
    NSArray *twitterAccounts = [self.accountStore  
accountsWithAccountType:twitterType];  
  
    if ([twitterAccounts count])  
    {  
        self.userAccount = [twitterAccounts objectAtIndex:0];  
        [[NSNotificationCenter defaultCenter]  
postNotification:[NSNotification  
notificationWithName:@"TwitterAccountAcquiredNotification"  
object:nil]];  
    }  
    else  
    {  
        NSLog(@"No Twitter Accounts");  
    }  
}
```

If the user has given you access to the Twitter accounts then you create an `NSArray` and fetch all of the accounts from the account store using the `accountsWithAccountType:` method.

Given that the user might not have any accounts setup at the moment it's a good idea to verify that there is indeed at least one account in the array of accounts you just created, if there are accounts in the array you just fetch the first one that's available.

For your applications you might want to ask the user which account they want to use, or perhaps even use both, but in this chapter you will keep things simple and just select the first account.

```
[ [NSNotificationCenter defaultCenter]
    postNotification:[NSNotification
        notificationWithName:@"TwitterAccountAcquiredNotification"
        object:nil]];
```

After you retrieve the first account from the device you post a notification using the default `NSNotificationCenter`, let me explain this a little bit.

Because your initial view controller will probably load before the user is finished granting permission to access the Twitter accounts on the device, or before the system accounts are retrieved, you need to post a notification to let your initial view controller know that you have an account and are ready to request information from Twitter.

If there are no accounts in our array you just write a little message to the console using `NSLog`, but in your apps you might wish to present a message to the user asking them to configure a Twitter account.

This wasn't too complicated right? Just initialize and retrieve the necessary objects before performing a request to access the system's Twitter accounts.

Compile and run your project and give it a try - if all works well, it should run without any errors to the console. If you are getting any errors or your array doesn't contain any accounts then try the following things:

- Go into Settings and make sure you have setup at least one Twitter account.
- In the Twitter section of Settings make sure you have allowed your application to access the Twitter Accounts.

Using twitter docs and dev console

Now that you have successfully retrieved access to a user's Twitter account, you are ready to start communicating with the Twitter API directly.

But how do you know what to pass into `TWRequest` and what Twitter will return? Let's take a small detour and cover the basics of using the Twitter documentation and using the Twitter Developer Console.

There are two ways to browse the Twitter API documentation, the first is by going to: <https://dev.twitter.com/docs>

And the other way is browsing it from within Xcode, yup you read that right, Xcode has a link to the Twitter documentation that you can use! Open up the Organizer by going to Window/Organizer or by using the keyboard combination of SHIFT + COMMAND + 2.

Once the Organizer is open select the documentation Tab and search for `TWRequest`. Select the `TWRequest` class reference from the results and in the Overview section there's a link titled "Twitter API Documentation". You can click this link in order to display the Twitter docs from within Xcode.

There's a lot of useful information you can read here like the Twitter Rules Of The Road, the Developer Console Documentation for testing API calls and the REST API specifications. You'll first learn how to read the Twitter API and then you'll cover the usage of the Developer Console included in the Twitter App for Mac OS X.

Right below the title that reads The Rest API you can click on a link titled REST API (pardon the redundancy) to check out the available calls you can make to Twitter.

Once inside you will find everything you need in order to communicate with Twitter and know what request to perform. There are sections for:

- Timelines
- Tweets
- Search
- Direct Messages
- Friends & Followers
- Users
- Suggested Users
- Favorites
- Lists
- Accounts
- Notification
- Saved Searches
- Local Trends
- Places & Geo
- Trends

- Block
- Spam Reporting
- OAuth
- Help
- Legal
- Deprecated

Whew, quite a lengthy list there but it's clearly categorized to help you find whatever you need from Twitter.

Scroll back up to the top and in the Timelines section you are going to use the very first item on the list, the `GET statuses/home_timeline` call, so go ahead and click it. Here's what it looks like:

The screenshot shows the Twitter Developers documentation page for the REST API v1.1. The specific endpoint being viewed is `GET statuses/home_timeline`. The page includes the following sections:

- Resource URL:** https://api.twitter.com/1.1/statuses/home_timeline.json
- Parameters:**
 - count** (optional): Specifies the number of records to retrieve. Must be less than or equal to 200. Defaults to 20. Example Value: 5
 - since_id** (optional): Returns results with an ID greater than (that is, more recent than) the specified ID. There are limits to the number of Tweets which can be accessed through the API. If the limit of Tweets has occurred since the since_id, the since_id will be forced to the oldest ID available. Example Value: 12345
 - max_id** (optional): Returns results with an ID less than (that is, older than) or equal to the specified ID. Example Value: 56789
 - trim_user** (optional): When set to either `true`, `t` or `1`, each tweet returned in a timeline will include a user object including only the status authors numerical ID. Omit this parameter to receive the complete user object. Example Value: `true`
 - exclude_replies** (optional): This parameter will prevent replies from appearing in the returned timeline. Using `exclude_replies` with the `count` parameter will mean you will receive up to `count` tweets — this is because the `count` parameter retrieves that many tweets before filtering out retweets and replies. Example Value: `true`
- 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:** Please [Sign In](#) with your Twitter account in order to use the OAuth tool.
- Related Documentation:** [GET statuses/user_timeline](#)
- Related Questions:** [How do I properly navigate a timeline?](#)
- Tags:** [finding tweets](#) (48)

There's plenty of information here for you to read, such as a description of the parameters, sample request and result, resource information on the right, and more. This call is perfect for pulling the user's feed so you'll definitely use it for your Feed View Controller.

It's not just a matter of copying and pasting the resource URL though, there are a few things you need to read and notice before using it with `TWRequest`.

First up notice that in the Resource URL the link ends with a .format. Then take a look at the Resource Information pane on the right and read the list of Response Formats. What this is telling you is that you can replace the .format at the end of the resource URL with any of those formats, which is what you'll receive the data in.

You are going to use .json with all of your calls because iOS 5 now includes the `NSJSONSerialization` class which makes it very easy to parse what the call returns into native Foundation objects.

In the Resource Information pane you also need to look at whether this call requires authentication or not as well as the supported HTTP methods. For this call you do need to pass in an account though there are other Twitter calls that don't require an account.

Always pay attention to this since you will need to specify an account to use with your request when using an authenticated call, otherwise you will not receive any data. You also need to know what HTTP methods are supported for when you make the call to the API inside your app.

Finally you should check out the Parameters section so you can specify what information you want to retrieve. We'll be retrieving the 30 latest tweets from the user's feed but you can do things like excluding retweets, excluding replies and a few other options here.

These parameters are for the home timeline call only. Each call in the Twitter API specifies its own Resource URL, parameters and authentication requirements so be sure to check out each method's specifications before actually writing any code.

Not too hard to read the docs right? We'll always use JSON with the resource URL, we have to take into account authentication and use any parameters we want when making the call.

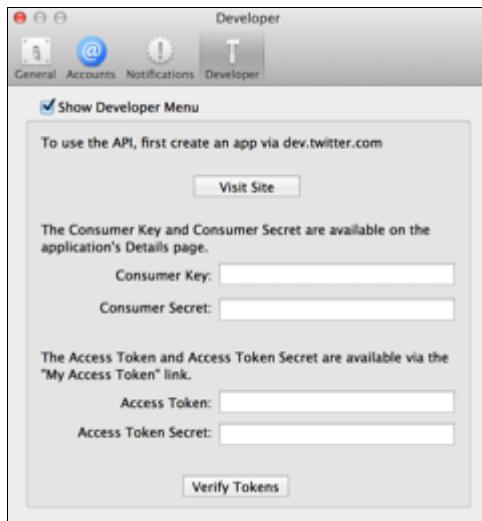
Great, now we have a good understanding of how to read the Twitter API Documentation and what we need in order to retrieve a user's home timeline feed so let's test this using the Developer Console that's part of the Twitter app for Mac OS X.

If you don't have Twitter for Mac installed then head on over to the Mac App Store and search for Twitter, it's a small, free download so get it installed on your machine.



Once you have Twitter installed on your Mac make sure to set up at least one account then go ahead and open the Preferences by going to Twitter/Preferences or using the keyboard shortcut of COMMAND + ,.

In the Preferences window select the Developer tab and enable the Show Developer Menu checkbox. Close the Preferences window and in the Menu Bar you should now have a Developer Menu Item, click it and then select Console (the only option available).

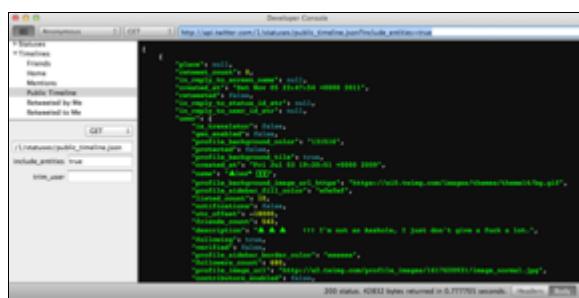


You now have the Console open and ready to begin testing some calls to the Twitter API, on the left side you will see a list of items that correspond to the category of the calls, the list is the same one you saw in the Twitter Docs.

Go ahead and expand the Timelines category and select Public Timeline from the available list, below the call list you have some options to fill in, these are the parameters that the call can receive (the same ones you can see in the documentation).

Notice also how at the top of the window you have a drop down menu with the HTTP Method, Authentication Type and the full URL with the response format and parameters. Leave the method to GET and use Anonymous with your request, in the request parameters leave their default value.

Once this is set up go to the URL field, click once to place the text cursor on it and hit Return, it should load the 200 latest tweets from the public timeline. This is how you can test the request calls and parameters to see what they return so feel free to test any methods you want to implement on your app.



If you require more info or help on how to use the Console then please refer to the Twitter developer page where you can find plenty of documentation along with discussion boards to ask questions and interact with other developers using Twitter.

Getting a user's feed and mentions

With your tests complete and your brains filled with knowledge of the Twitter API, let's download some actual tweets!

Open up **FeedViewController.m** and add the following code in order to get the user's home timeline feed. The first thing you need to do is include some headers:

```
#import "AppDelegate.h"
#import "FeedViewController.h"
#import "TweetCell.h"
#import <Twitter/Twitter.h>
```

FeedViewController.h should already be included but you also need the Twitter Framework's classes, your custom Tweet Cell and the App Delegate so you can retrieve the accounts and store the profile images downloaded.

Next you add a category for private properties:

```
@interface FeedViewController ()
```

```
@property (strong, nonatomic) NSArray *tweetsArray;  
@end
```

I've just marked the rest of the view controller's code with three dots (...) so you know there's more to our implementation file.

There's a category declaration for your `FeedViewController` that specifies an `NSArray` property that is strong and nonatomic.

Now add the following code to your `viewDidLoad` method:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    // If we've already retrieved a user account then  
    // get the user's feed, otherwise just register for the  
    // TwitterAccountAcquiredNotification.  
    AppDelegate *appDelegate = [[UIApplication sharedApplication]  
delegate];  
  
    if (appDelegate.userAccount)  
    {  
        [self getFeed];  
    }  
  
    // Register ourselves to listen for the  
    // TwitterAccountAcquiredNotification after the view is loaded  
    [[NSNotificationCenter defaultCenter] addObserver:self  
                                             selector:@selector(getFeed)  
                                               name:@"TwitterAccountAcquiredNotification"  
                                             object:nil];  
}
```

Remember that notification you posted back in our app delegate when you finish getting a user's Twitter accounts? Well you register the Feed View Controller as an observer of that notification and ask it to call the `getFeed` method when the notification occurs.

You also call `getFeed` only if there is at least one account available. You do this so you can load the feed in case the user decides to switch to other view controllers, causing your feed view controller to be released from memory. If this were to occur then you would not reload your feed because it's only called once during the app's lifetime after you receive the notification that an account was acquired.

Another thing you want to do is reload your tweets when the user shakes the device, in order to do this you must implement a few methods, first the `canBecomeFirstResponder`, `viewDidAppear:` and `viewDidDisappear:` methods:

```
- (BOOL)canBecomeFirstResponder
{
    return YES;
}
```

First you tell the system you can become first responders by overriding the `canBecomeFirstResponder` method, then you become the actual first responder when your view appears and resigning your first responder status when no longer visible.

Finally in order to support shaking to reload the tweets you need to implement the following method:

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
    [self getFeed];
}
```

This method will be called when a shake occurs and inside of it you just call the `getFeed` method to update the Twitter feed.

Next you can delete the method for `numberOfSectionsInTableView:` and make this small change to the `numberOfRowsInSection:` method:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return self.tweetsArray.count;
}
```

This will use the same amount of tweets stored in your array.

Things will get a bit more interesting from here on out, it might look like a lot of code but don't worry since none of it will look too strange or new to you, especially if you are comfortable with Grand Central Dispatch and Blocks.

Let's implement the `getFeed` method by adding the following code:

```
- (void)getFeed
{
    // Store The url for the home timeline (we are getting the
    results in JSON format)
```

```
NSURL *feedURL = [NSURL
URLWithString:@"http://api.twitter.com/1/statuses/home_timeline.json"];
// Create an NSDictionary for the twitter request parameters.
// We specify we want to get 30 tweets though this can be changed to
// what you like
NSDictionary *parameters = @{@"count" : @"30"};
// Create a new TWRequest, use the GET request method, pass in
// our parameters and the URL
TWRequest *twitterFeed = [[TWRequest alloc]
initWithURL:feedURL
parameters:parameters
requestMethod:TWRequestMethodGET];
// Get the shared instance of the app delegate
AppDelegate *appDelegate = [[UIApplication sharedApplication]
delegate];
// Set the twitter request's user account to the one we
// downloaded inside our app delegate
twitterFeed.account = appDelegate.userAccount;
// Enable the network activity indicator to inform the user
// we're downloading tweets
UIApplication *sharedApplication = [UIApplication
sharedApplication];
sharedApplication.networkActivityIndicatorVisible = YES;
// Perform the twitter request
[twitterFeed performRequestWithHandler:^(NSData *responseData,
NSHTTPURLResponse *urlResponse, NSError *error) {
if (!error)
{
    // If no errors were found then parse the JSON into a
    // foundation object
    NSError *jsonError = nil;

    id feedData = [NSJSONSerialization
JSONObjectWithData:responseData
options:0
```

```
error:&jsonError];
    if (!jsonError)
    {
        // If no errors were found during the JSON parsing
        then update our feed table
        [self updateFeed:feedData];
    }
    else
    {
        // In case we had an error parsing JSON then show
        the user an alert view with the error's description
        UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error"

message:[jsonError localizedDescription]

delegate:nil

cancelButtonTitle:@"OK"

otherButtonTitles:nil];
        [alertView show];
    }
}
else
{
    // In case we couldn't perform the twitter request
    successfully then show the user an alert view with the error's
    description
    UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error"

message:[error localizedDescription]

delegate:nil

cancelButtonTitle:@"OK"

otherButtonTitles:nil];
        [alertView show];
}

// Stop the network activity indicator since we're done
downloading data
```

```
    sharedApplication.networkActivityIndicatorVisible = NO;
} ];
}
```

Whoa, that's quite long but I've commented the code so you can have a better understanding of what goes on.

The first thing you do is create an `NSURL` object and use the Resource URL you got from the Twitter Docs, notice how you use `.json` at the end of the URL to specify that you want the data returned to be in the JSON format.

After that you create an `NSDictionary` with the parameters you want to send with the request, again these parameters are the ones specified in the Twitter Docs, you're asking for the 30 most recent tweets in the user's feed.

None of those lines should be new to you if you've worked with Cocoa before. The next line is definitely new though, you create a `TWRequest` object and initialize it with the URL and parameters dictionary you just created, you also specify the `TWRequestMethodGet` option for the `requestMethod:` parameter.

All of this information (format, parameters and request method) is what you read in the documentation.

Since this method requires authentication you get the `ACAccount` object from the app delegate and set it as the `TWRequest` object's account. Remember that this call requires authentication so this is how you pass in an account, `TWRequest` will handle the authenticated for you automatically just by providing it with an account.

Before performing the request you just enable the network activity indicator to let the user know you are downloading info from Twitter, some visual feedback so you show we're actually doing something.

Now that you have everything ready you call the `performRequestWithHandler:` method on your `TWRequest` object, it also receives a `block` object to use as a completion handler.

Note: Block objects are here to stay and you can definitely see that in Cocoa's newer APIs, most (if not all of them) use some sort of block object integration as well as GCD. If you are having trouble with this concept then check out the [Introduction to Grand Central Dispatch](#) tutorial available at raywenderlich.com.

The block returns three objects for us: an `NSData` object, an `NSError` and an `NSHTTPURLResponse`. Inside the completion handler you use an `if` statement to determine whether there were errors with the request or not (always handle errors!) and if there are any, show the error's localized description in an alert view.

If the request was successful you create an `NSError` object initialized to nil and an object of type id initialized with the object that's returned to us by calling the `JSONObjectWithData:` method from the `NSJSONSerialization` class.

If there are errors parsing the JSON data you show another alert view with the error's localized description, otherwise you simply call our `updateFeed:` method and pass in the `feedData` object containing our JSON Foundation objects.

Finally you just stop the network activity indicator since you're no longer downloading tweets.

You're done with the `getFeed` method, phew! Luckily for you the `updateFeed:` method is very short and trivial:

```
- (void)updateFeed:(id)feedData
{
    // We receive the NSArray of tweets and store it in our local
    // tweets array
    self.tweetsArray = (NSArray *)feedData;

    // Update the number of rows in the table view
    [self.tableView numberOfRowsInSection:self.tweetsArray.count];

    // Reload the table view's data
    [self.tableView reloadData];
}
```

You just cast our `feedData` to an `NSArray` pointer and store it within your `tweetsArray` property and reload the table view to display the new tweets.

Finally you must change the `cellForRowAtIndexPath:` method to use the information acquired from Twitter and show it using the custom Tweet Cell:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Dequeue a reusable cell if available or create a new one if
    // necessary
    TweetCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"TweetCell"];

    // Get the dictionary for the current tweet as well as the
    // user
    NSDictionary *currentTweet = self.tweetsArray[indexPath.row];
    NSDictionary *currentUser = currentTweet[@"user"];

    // Set the user name and tweet labels, use the default cell
    // image until we load ours
}
```

```
cell.usernameLabel.text = currentUser[@"name"];
cell.tweetLabel.text = currentTweet[@"text"];
cell.userImage.image = [UIImage
imageNamed:@"ProfileImage_Default.png"];

// Get an instance to the app delegate
AppDelegate *appDelegate = [[UIApplication sharedApplication]
delegate];

// Store the username in a string to make our code shorter
NSString *userName = cell.usernameLabel.text;

// If the current user's image is inside our app delegate's
profile image dictionary then get it, otherwise download the image
if ([appDelegate.profileImages objectForKey:userName])
{
    cell.userImage.image =
appDelegate.profileImages[userName];
}
else
{
    // Get a concurrent queue form the system
    dispatch_queue_t concurrentQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    // Perform an asynchronous operation using our concurrent
queue in order to get the user's image from the web
    dispatch_async(concurrentQueue, ^{
        // Store the image link into an NSURL object
        NSURL * imageURL = [NSURL
URLWithString:currentUser[@"profile_image_url"]];

        // create an NSData object to store our image's data
        __block NSData * imageData;

        // Download the image synchronously using our queue
variable created earlier
        dispatch_sync(concurrentQueue, ^{
            imageData = [NSData
dataWithContentsOfURL:imageURL];

            // After we download the image we store it in the
app delegate's profile images dictionary with the username as the
key
        });
    });
}
```

```

        [appDelegate.profileImages setObject:[UIImage
imageWithData:imageData] forKey:userName];
    });

        // Update the cell's user image on the main thread,
all UI operations must be performed on the main thread!!!
dispatch_sync(dispatch_get_main_queue(), ^{
    cell.userImage.image =
appDelegate.profileImages[userName];
});
}

return cell;
}

```

Once again do not worry if it looks like a lot of code, I will explain what everything does and there are comments to help you understand things even better. First up is a call to `dequeueReusableCellWithIdentifier:` to get an instance of the cell you made in the storyboard with the `TweetCell` identifier:

```

TweetCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"TweetCell"];

```

You no longer have to check if this returns `nil` and then create a new instance of the cell, it's all done automatically for you in case there are no reusable cells with the specified identifier.

After getting a Tweet Cell you get the current tweet and the user who sent that tweet. If you print out the `currentTweet` dictionary (or if you checkout the Twitter documentation) you will notice that the `user` key is actually another dictionary, which is why it's being stored into the `currentUser` dictionary.

```

// Get the dictionary for the current tweet as well as the user
NSDictionary *currentTweet = self.tweetsArray[indexPath.row];
NSDictionary *currentUser = currentTweet[@"user"];

```

Moving on, you set the tweet cell's `usernameLabel` and `tweetLabel` contents to what was retrieved from the tweet and user dictionaries, you also set the `userImage` with a default image I've created to be displayed until you download (or use) the actual profile image.

```

cell.usernameLabel.text = currentUser[@"name"];
cell(tweetLabel.text = currentTweet[@"text"];
cell.userImage.image = [UIImage
 imageNamed:@"ProfileImage_Default.png"];

```

Note: Go ahead and copy the **ProfileImageDefault.png** and **ProfileImageDefault@2x.png** files from the resources for this chapter.

If you want to know everything that's returned by the request you can print your dictionaries using `NSLog` to see all the keys and values available or, alternatively, you can check out the documentation for a detailed description of the keys and objects returned.



The screenshot shows the Twitter Developers API Explorer interface. At the top, it says "Example Request". Below that, it shows a GET request to `https://api.twitter.com/1.1/statuses/mentions_timeline.json?count=2ssince_id=14927799`. The response body is displayed as a JSON object with 45 numbered lines. The JSON structure includes entities, hashtags, user_mentions, and screen names for three users: Jason Costa, Matt Harris, and ThinkWall.

```

1. {
2.   "entities": {
3.     "coordinates": null,
4.     "favorited": false,
5.     "retweeted": false,
6.     "created_at": "Mon Sep 03 13:24:14 +0000 2012",
7.     "id": "242613977796685040",
8.     "id_str": "242613977796685040",
9.     "entities": {
10.       "urls": []
11.     },
12.     "hashtags": [],
13.     "user_mentions": [
14.       {
15.         "name": "Jason Costa",
16.         "id": "14927800",
17.         "id_str": "14927800",
18.         "indices": [
19.           0,
20.           11
21.         ],
22.         "screen_name": "jasoncosta"
23.       },
24.       {
25.         "name": "Matt Harris",
26.         "id": "779925",
27.         "id_str": "779925",
28.         "indices": [
29.           12,
30.           26
31.         ],
32.         "screen_name": "thematttaharris"
33.       },
34.       {
35.         "name": "ThinkWall",
36.         "id": "117426578",
37.         "id_str": "117426578",
38.         "indices": [
39.           109,
40.           119
41.         ],
42.         "screen_name": "thinkwall"
43.       }
44.     ]
45.   }
}

```

Next up you get your app delegate instance and create an `NSString` which contains the tweet's username, you'll use both of these up next in order to retrieve the profile image from your app delegate's dictionary or to download it from Twitter. The username will be the key to the image stored in the dictionary, an easy way to look for each image.

```

AppDelegate *appDelegate = [[UIApplication sharedApplication]
delegate];

NSString *userName = cell.usernameLabel.text;

```

Now comes the meaty part, either getting a profile image from the app delegate's dictionary or downloading it from Twitter using Grand Central Dispatch.

You use an if statement to check whether your app delegate's `profileImages` dictionary already contains the tweeting user's profile image or not. If it does then you just get it and set your cell's `userImage` property to it.

```
if ([appDelegate.profileImages objectForKey:userName])
{
    cell.userImage.image = appDelegate.profileImages[userName];
}
else
{
    // ...
}
```

If you can't find an image then you use some GCD magic to asynchronously download the image on a separate thread (so the UI doesn't freeze or hang) and then update the cell's image on the main thread:

```
else
{
    // Get a concurrent queue from the system
    dispatch_queue_t concurrentQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    // Perform an asynchronous operation using our concurrent
queue in order to get the user's image from the web
    dispatch_async(concurrentQueue, ^{
        // Store the image link into an NSURL object
        NSURL *imageURL = [NSURL
URLWithString:currentUser[@"profile_image_url"]];

        // Create an NSData object to store our image's data
        __block NSData *imageData;

        // Download the image synchronously using our queue
variable created earlier
        dispatch_sync(concurrentQueue, ^{
            imageData = [NSData
dataWithContentsOfURL:imageURL];

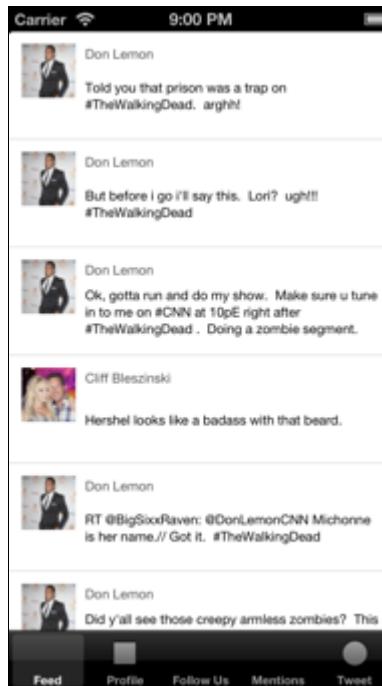
            // After we download the image we store it in the app
delegate's profile images dictionary with the username as the key
            [appDelegate.profileImages setObject:[UIImage
imageWithData:imageData] forKey:userName];
        });
    });
}
```

```
// Update the cell's user image on the main thread, all UI
operations must be performed on the main thread!!!
dispatch_sync(dispatch_get_main_queue(), ^{
    cell.userImage.image =
appDelegate.profileImages[userNames];
});
})
```

You create a new queue by getting a global system queue with default priority, then you call `dispatch_async` and initialize an `NSURL` object with the URL that's specified in the `currentUser`'s `profile_image_url` object. Inside this asynchronous thread we call `dispatch_sync` to perform the actual request that will download the image, you do this synchronously so you download one image at a time before moving on to the next one.

Once the image is downloaded you store it in the app delegate's `profileImages` dictionary using the username as the key and call `dispatch_sync` once more (this time in the main thread) to update the tweet cell's `userImage` property to the newly downloaded image.

Go ahead and run your app, if this is the first time running it you'll be asked to grant Twitter access to your accounts (make sure you've added some in Settings) and after a second or two the table view will be filled with Tweets!



Yay, how cool is that? This isn't the most polished interface and there's plenty that could be done to improve the experience for the user, for example you could use a

loading overlay so the user knows whether the app is doing something or not, but that's homework for all you awesome programmers ;)

You'll improve the colors and look a bit later, for now let's continue with more Twitter goodness. Next up is the Mentions View Controller which uses almost exactly the same code as the Feed View Controller, as a matter of fact go ahead and copy and paste everything from the `FeedViewController.m` file to the `MentionsViewController.m` file.

Make sure that in the process of copying the code you use the correct class names for the `@interface` and `@implementation` declarations.

The only change you will make is the Resource URL. In order to see what URL you should use let's refer to the Twitter API Documentation once again.

Take a look at the `GET statuses/mentions` call, notice how it's almost identical to the `home_timeline` call and uses the same parameters, the GET HTTP method and authentication.

Copy and paste the Resource URL from the documentation and replace it with the current `feedURL` inside the Mentions View Controller's `getFeed` method:

```
NSURL *feedURL = [NSURL  
URLWithString:@"http://api.twitter.com/1/statuses/mentions.json"];
```

Don't forget to append the correct format at the end, which in your case is `.json`, if you want you can change the amount of mentions to retrieve though I'll leave it at 30 as well.

Go ahead and run your application once again, now select the mentions view controller from the tab bar and check out your results:



Sweeeeet, you have done a lot of things with very little code. You could argue that it's been a lot of code but most of it isn't even related to Twitter or `TWRequest`, it's just error handling, GCD and UI elements.

Again this isn't the most polished example of how a Twitter app should be made, this goes to show all the work that goes on behind the scenes of a very polished app like the official Twitter iOS App.

As an added bonus you could add a reload button (perhaps even letting the user pull down on the table view to cause it to refresh), show a loading overlay, be able to select a tweet and display some options for it, and so on.

Plenty of things to do but once again, that's beyond the purpose of this chapter (which is using the Twitter and Accounts Frameworks) so I'll leave it for you to explore and enhance.

The "follow us" view controller

One feature that I'm sure many of you will be interested in implementing is a way for your users to follow you on Twitter with the click of a button, luckily for you this is just a matter of a simple call to the Twitter API.

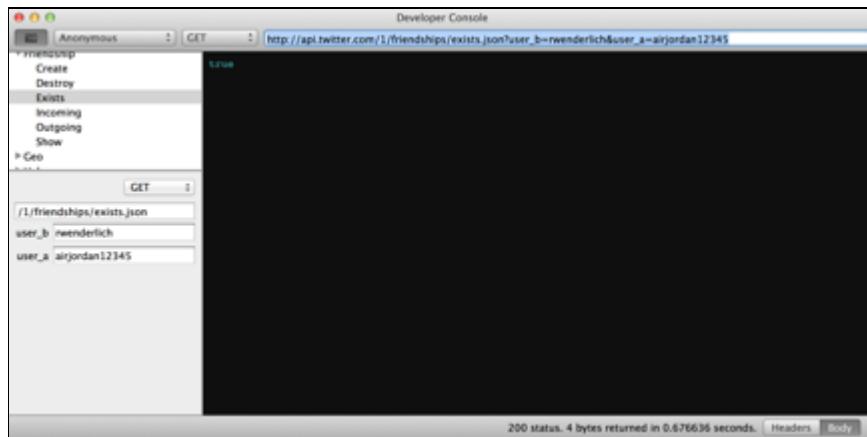
Remember that you created the `FollowUsViewController` exactly for this purpose. What you're going to do is ask Twitter if the user is already following you, in case the user is already a follower then you will change the view's label text to `Unfollow`, otherwise you will show `Follow` so that a user can become your follower without leaving the app.

If you check out the Twitter REST API documentation once again you will notice some useful methods for your purpose in the Friends & Followers section, the calls you are interested in are:

- GET friendships/exists
- POST friendships/create
- POST friendships/destroy

The exists call will let you know if user A is following user B and will return only true or false, the create and destroy calls will let you follow and unfollow a user respectively.

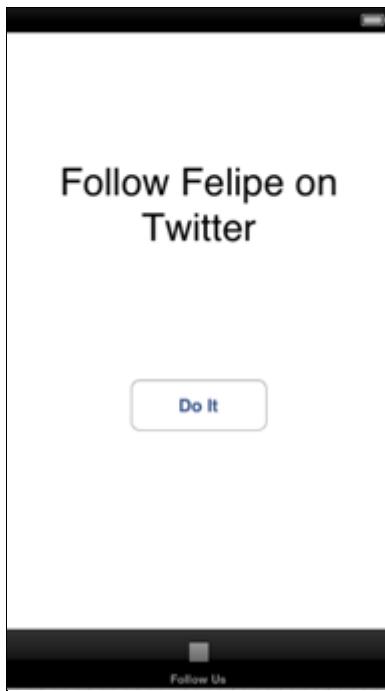
You can test these calls in the Twitter Developer Console, I've selected the exists call under the Friendship category and used my Twitter username to see if I follow Ray's Twitter account, you can see that it returns true since I do indeed follow Ray on Twitter.



Ok, now that you know what calls you need to make to Twitter, the documentation for these calls and how you are going to implement them within your app, it's time to get coding once more.

If you haven't already go ahead and open up MainStoryboard.storyboard so you can add some interface elements to the Follow Us View Controller.

From the Object Library drag a `UILabel` and a `UIButton` to the Follow Us View Controller, I've used the text "Follow Felipe On Twitter" for my label and set Do It as the title for my button, feel free to customize this view however you like though.



Now let's make an `IBAction` for when your button is tapped and a label outlet to change the text in your Follow Us view controller. As you may already know, there are two ways to do this, you can write the code yourself or alternatively you can use the Assistant Editor to drag from the connections box.

Whether you chose to connect the button using the Assistant Editor or by writing the code and then connecting it to the existing outlet, this is what

FollowUsViewController.m should look like:

```
#import "AppDelegate.h"
#import "FollowUsViewController.h"
#import <Twitter/Twitter.h>

#define kAccountToFollow @"airjordan12345"

@interface FollowUsViewController : UIViewController

@property (assign) BOOL isFollowing;
@property (weak, nonatomic) IBOutlet UILabel *textLabel;

- (IBAction)followTapped;

@end
```

There's just a `UILabel` outlet and a simple `IBAction` that will allow you to perform the Twitter call to follow or unfollow an account. Make sure you connect the label's

outlet and the action in your storyboard file. There's also a Boolean to store whether you are already following the account or not and some header imports.

Now in order for you to determine whether you need to follow or unfollow a user you are going to write a `checkFollowing` method to determine whether you're already a follower or not. Implement the method as follows:

```
- (void)checkFollowing
{
    // Store The url for the home timeline (we are getting the
    // results in JSON format)
    NSURL *feedURL = [NSURL
URLWithString:@"http://api.twitter.com/1/friendships/exists.json"]
;

    // Get the shared instance of the app delegate
    AppDelegate *appDelegate = [[UIApplication sharedApplication]
delegate];

    // Create an NSDictionary for the twitter request parameters.
    // We specify the users to check for.
    NSDictionary *parameters = @{@"screen_name_a" :
appDelegate.userAccount.username,
                                @"screen_name_b" :
kAccountToFollow};

    // Create a new TWRequest, use the GET request method, pass in
    // our parameters and the URL
    TWRequest *twitterFeed = [[TWRequest alloc]
initWithURL:feedURL

parameters:parameters

requestMethod:TWRequestMethodGET];

    // Set the twitter request's user account to the one we
    // downloaded inside our app delegate
    twitterFeed.account = appDelegate.userAccount;

    // Enable the network activity indicator to inform the user
    // we're downloading tweets
    UIApplication *sharedApplication = [UIApplication
sharedApplication];
    sharedApplication.networkActivityIndicatorVisible = YES;

    // Perform the twitter request
```

```
[twitterFeed performRequestWithHandler:^(NSData *responseData,
NSHTTPURLResponse *urlResponse, NSError *error) {
    if (!error)
    {
        NSString *responseString = [[NSString alloc]
initWithBytes:[responseData bytes]

length:[responseData length]

encoding:NSUTF8StringEncoding];

        // Set the label's text depending on whether we're
already a follower or not.
        if ([responseString isEqualToString:@"true"])
        {
            self.textLabel.text = @"Unfollow Felipe On
Twitter";

            _isFollowing = YES;
        }
        else
        {
            self.textLabel.text = @"Follow Felipe On Twitter";

            _isFollowing = NO;
        }
    }
    else
    {
        // In case we couldn't perform the twitter request
successfully then show the user an alert view with the error's
description
        UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
        [alertView show];
    }
}
```

```
// Stop the network activity indicator since we're done  
downloading data  
sharedApplication.networkActivityIndicatorVisible = NO;  
});  
}
```

This code should look pretty familiar by now since you used it on both the Feed and Mentions view controllers. You are using the URL to check if user A is already a follower of user B, you specify JSON as the return format. After this you set the parameters for both your usernames, create a request with the GET method, set the account to use with the request and perform the request.

Inside the request you only check for a true or false string since this is the return of the exists call to Twitter. Depending on what you get you change your private boolean to indicate you are already a follower or not and change the label's text accordingly, if an error occurred you display an alert view with the localized description of the error.

Now let's call the `checkFollowing` method every time the view loads:

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

This will automatically check if you are a follower whenever the Follow Us view controller is loaded.

With the `checkFollowing` method ready, you can now add the following code to the `followTapped` method:

```
- (IBAction)followTapped  
{  
    // Store The url for the follow calls (we are getting the  
    results in JSON format)  
    NSURL *feedURL;  
  
    // Use the proper URL depending on whether we're already a  
    follower or not.  
    if (_isFollowing)  
    {  
        feedURL = [NSURL  
URLWithString:@"http://api.twitter.com/1/friendships/destroy.json"  
];  
    }  
}
```

```
else
{
    feedURL = [NSURL
URLWithString:@"http://api.twitter.com/1/friendships/create.json"]
;
}

// Get the shared instance of the app delegate
AppDelegate *appDelegate = [[UIApplication sharedApplication]
delegate];

// Create an NSDictionary for the twitter request parameters.
// We specify the users to check for.
NSDictionary *parameters = @{@"follow" : @"true",
@"screen_name" : kAccountToFollow};

// Create a new TWRequest, use the GET request method, pass in
our parameters and the URL
TWRequest *twitterFeed = [[TWRequest alloc]
initWithURL:feedURL

parameters:parameters

requestMethod:TWRequestMethodPOST];

// Set the twitter request's user account to the one we
downloaded inside our app delegate
twitterFeed.account = appDelegate.userAccount;

// Enable the network activity indicator to inform the user
we're downloading tweets
UIApplication *sharedApplication = [UIApplication
sharedApplication];
sharedApplication.networkActivityIndicatorVisible = YES;

// Perform the twitter request
[twitterFeed performRequestWithHandler:^(NSData *responseData,
NSHTTPURLResponse *urlResponse, NSError *error) {
    if (!error)
    {
        // Change the label and the boolean to indicate we are
following or no longer following the account.
        if (!_isFollowing)
        {

```

```
        self.textLabel.text = @"Unfollow Felipe On
Twitter";

        _isFollowing = YES;
    }
else
{
    self.textLabel.text = @"Follow Felipe On Twitter";

    _isFollowing = NO;
}
}
else
{
    // In case we couldn't perform the twitter request
    // successfully then show the user an alert view with the error's
    // description
    UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
    [alertView show];
}

// Stop the network activity indicator since we're done
// downloading data
sharedApplication.networkActivityIndicatorVisible = NO;
}];
}
```

First up you check if you are already a follower or not and use the proper URL for this, if you are a follower you use the destroy call and if you are not a follower ou use the create call. Both of these use the POST method and authentication so we make sure to specify that in the TWRequest instance.

After this you perform the request and if there were no errors you simply change the label's text and your local boolean value, otherwise you show an alert view with the error's localized description.

Go ahead and run your project and test out the follow functionality, when you first select the Follow Us view controller from the tab bar it will check if you are already a follower and depending on this you will either become a follower or not when the Do It button is tapped.

Only the Tweet and Profile View Controllers remain, let's get working on those now.

Profile and tweet view controllers

The profile view controller will let you see the user's name, description, tweets, favorites, followers and following.

Open up `MainStoryboard.storyboard` and add some labels to the Profile View Controller in order to accommodate this info. Here's how I set up my Profile View Controller:



Next, connect the labels to outlets (either manually or via dragging to the assistant editor). When you're done, `ProfileViewController.m` should look like this:

```
#import "ProfileViewController.h"
#import "AppDelegate.h"
#import <Twitter/Twitter.h>

@interface ProfileViewController : UIViewController

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

```
@property (weak, nonatomic) IBOutlet UILabel *tweetsLabel;
@property (weak, nonatomic) IBOutlet UILabel *favoritesLabel;
@property (weak, nonatomic) IBOutlet UILabel *followingLabel;
@property (weak, nonatomic) IBOutlet UILabel *followersLabel;

@end
```

Next start adding the code to pull the profile from Twitter. Once again this code will be very familiar to you and only minor things will change (such as the parameters and requestURL):

```
- (void)loadProfile
{
    // Store The url for the profile call (we are getting the
    // results in JSON format)
    NSURL *feedURL = [NSURL
URLWithString:@"http://api.twitter.com/1/users/show.json"];

    // Get the shared instance of the app delegate
    AppDelegate *appDelegate =
    [[UIApplication sharedApplication] delegate];

    // Set the username label to the user's screen name
    self.usernameLabel.text = appDelegate.userAccount.username;

    // Create an NSDictionary for the twitter request parameters.
    // We specify we want to get 30 tweets though this can be changed to
    // what you like
    NSDictionary *parameters = @{@"screen_name" :
        appDelegate.userAccount.username};

    // Create a new TWRequest, use the GET request method, pass in
    // our parameters and the URL
    TWRequest *twitterFeed = [[TWRequest alloc]
        initWithURL:feedURL
        parameters:parameters
        requestMethod:TWRequestMethodGET];

    // Set the twitter request's user account to the one we
    // downloaded inside our app delegate
    //twitterFeed.account = appDelegate.userAccount;

    // Enable the network activity indicator to inform the user
    // we're downloading tweets
    UIApplication *sharedApplication =
    [UIApplication sharedApplication];
```

```
sharedApplication.networkActivityIndicatorVisible = YES;

// Perform the twitter request
[twitterFeed performRequestWithHandler:^(NSData *responseData,
NSHTTPURLResponse *urlResponse, NSError *error) {
    if (!error)
    {
        // If no errors were found then parse the JSON into a
foundation object
        NSError *jsonError = nil;

        id feedData = [NSJSONSerialization
            JSONObjectWithData:responseData
            options:0
            error:&jsonError];

        if (!jsonError)
        {
            NSDictionary *profileDictionary =
                (NSDictionary *)feedData;

            dispatch_sync(dispatch_get_main_queue(), ^{
                self.descriptionLabel.text =
profileDictionary[@"description"];
                self.favoritesLabel.text = [profileDictionary
[@@"favourites_count"] stringValue];
                self.followersLabel.text =
[profileDictionary[@@"followers_count"] stringValue];
                self.followingLabel.text =
[profileDictionary[@@"friends_count"] stringValue];
                self.tweetsLabel.text =
[profileDictionary[@@"statuses_count"] stringValue];
            });
        }
        else
        {
            dispatch_sync(dispatch_get_main_queue(), ^{
                // In case we had an error parsing JSON then
show the user an alert view with the error's description
                UIAlertView *alertView =
                    [[UIAlertView alloc] initWithTitle:@"Error"
message:[jsonError localizedDescription]
delegate:nil
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
            });
        }
    }
}];
```

```
                [alertView show];
            });
        }
    }
    else
    {
        dispatch_sync(dispatch_get_main_queue(), ^{
            // In case we couldn't perform the twitter request
            // successfully then show the user an alert view with the error's
            // description
            UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
            [alertView show];
        });
    }

    // Stop the network activity indicator since we're done
    // downloading data
    sharedApplication.networkActivityIndicatorVisible = NO;
};

}
```

Everything has been commented thoroughly so you can understand what each section does. You begin by creating an `NSURL` object with the link to a user's profile description and then getting the application's `AppDelegate` instance.

Because the `ACAccount` object stored in the app delegate contains the username of the user's Twitter account, you set the `usernameLabel` to that text instead of waiting for it to download from Twitter. After you set the `usernameLabel` you create an `NSDictionary` with the username of the Twitter profile you want to get the info for, the keys used for each call are specified in the Twitter documentation.

With your URL and parameters ready you create a `TWRequest` object, pass in the url, parameters and specify the GET HTTP method, you then show the network activity indicator so the user can have some feedback that the network is being accessed.

After performing the request you fill in the completion handler with a check for any errors in the request, once again if there are errors you show an alert view otherwise you proceed to parse the JSON results. The JSON parsing is checked for errors and if there are none then the labels are updated with the information you receive from the request.

Pretty simple, right? It's very similar to the code used for pulling a user's feed and mentions with only a change in the URL, parameters and how you use the information we receive. The last thing you need to do is call the `loadProfile` method when the view is about to appear by adding the following code to the `viewWillAppear:` method:

```
- (void)viewWillAppear:(BOOL)animated
{
    [self loadProfile];
}
```

With this, you are done with the `ProfileViewController`. Go ahead and run your project and check out your results:



Though the `ProfileViewController` shows just a few of the attributes downloaded from the API there are a lot of other things you can show and use here.

It's now time to move on to the final view controller, `TweetViewController`! This one is a bit different from what you've been doing so far because how you end up using `TWRequest` depends on whether you are simply uploading a text status or an image to Twitter.

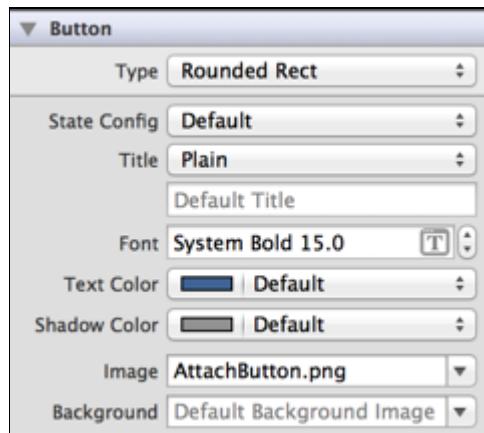
You might wonder why you're creating our own `TweetViewController` instead of simply using the built in `TWTweetComposeViewController` "Tweet Sheet" that was covered last chapter. Well, sometimes you might want to put your own user interface on top of sending a tweet, or send a tweet programmatically within your app, so it's good to know how to do this.

Before you go into the details of how this works in code, let's make some changes to the storyboard:



Try to mimic this layout the best you can - for the image, it's actually a custom button with an image as the background. In order to make the custom button add the **AttachButton.png** and **AttachButton@2x.png** images from the book's source code into your project. Alternatively you can use your own image for the button.

Now, in order to make the button have a custom image you have to change some of its properties in Interface Builder. With the button selected, open the Attributes Inspector and change the properties as shown below:



The button's type is now "Custom", the State Config is set to "Default" and for the Image property you use the name of the AttachButton.png image (iOS will

automatically use the higher resolution AttachButton@2x image on retina enabled devices).

There's also a text field for the tweet status, some labels for informative purposes and a Tweet button. This isn't the most polished of interfaces, primarily because, since it's beyond the scope of this chapter, you're not loading images from a user's photo library or checking for a status that's 140 characters or less.

Connect the "attached" label, the status label, and the text field to outlets, and the image button and tweet button to actions manually or with the assistant editor. When you're done **TweetViewController.m** should look like this:

```
#import "AppDelegate.h"
#import "TweetViewController.h"
#import <Twitter/Twitter.h>

@interface TweetViewController : UIViewController

@property (assign) BOOL isAttached;
@property (weak, nonatomic) IBOutlet UILabel *attachedLabel;
@property (weak, nonatomic) IBOutlet UITextField *statusTextField;
@property (weak, nonatomic) IBOutlet UILabel *successLabel;

- (IBAction)attachTapped:(id)sender;
- (IBAction)tweetTapped:(id)sender;

@end
```

Before moving on with your `TWRequest` let's explain how sending a tweet works. If you are uploading a text status to Twitter you use the following link:

<http://api.twitter.com/1/statuses/update.json>

Then you pass in the status as a key/value pair in your dictionary and execute the request as you've been doing all along.

However, if you are going to send an image with your tweet then you must send what's called multipart data. Using multipart data in your tweets is not complicated at all, you create a `TWRequest` as you normally do but for the parameters dictionary you just use nil. Then, when you want to add an image or some text to the request you use the following method:

```
[twitterFeed addMultiPartData:UIImagePNGRepresentation([UIImage
imageNamed:@"TweetImage.png"]) withName:@"media"
type:@"image/png"];
[twitterFeed addMultiPartData:[@"Some text for our tweet status!"
dataUsingEncoding:NSUTF8StringEncoding] withName:@"status"
type:@"text/plain"];
```

The first call to the `addMultiPartData:` method sends in an image as the data of your multipart POST body, then you specify the name of what you're sending and the type of the data you're sending. These parameters are available both in the Twitter documentation as well as in Xcode so be sure to check that your for all the possible values and supported image formats.

The second call to the `addMultiPartData:` method just sends some UTF8 encoded text, specifying that it's your status text and the type of `text/plain`.

Now that you know how to send a simple status or one with multipart data let's go ahead and implement the rest of your `TweetViewController`'s implementation.

Now let's add the code for the `dismissKeyboard` method:

```
- (void)dismissKeyboard
{
    // Dismiss the keyboard if visible when the user has tapped
    // the view.
    if (self.statusTextField.isFirstResponder)
    {
        [self.statusTextField resignFirstResponder];
    }
}
```

When this method is called it will check if the keyboard is currently being shown by asking the `statusTextField` if it's the first responder and hide it in case it's visible by resigning the first responder status on your `statusTextField`.

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

    // Initialize our _isAttached BOOL to NO.
    _isAttached = NO;

    // Create a tap gesture recognizer so he user can dismiss the
    // keyboard by tapping anywhere in the view.
    UITapGestureRecognizer *tapRecognizer =
    [[UITapGestureRecognizer alloc] initWithTarget:self
    action:@selector(dismissKeyboard)];
    tapRecognizer.numberOfTapsRequired = 1;
    tapRecognizer.cancelsTouchesInView = NO;

    [self.view addGestureRecognizer:tapRecognizer];
}
```

In the `viewDidLoad` method you initialize your `isAttached` boolean to NO (since the `TweetViewController` just loaded the user couldn't have chosen to attach an image

yet) and add a tap gesture recognizer to call the dismiss keyboard method, this will enable a user to hide the keyboard by simply tapping anywhere in the view.

Moving on to the actions you have the `attachTapped` button:

```
- (IBAction)attachTapped
{
    // If the image is already attached then un-attach it, empty
    // the label and change our boolean to NO, otherwise set the attached
    // label's text and our boolean to YES.
    if (_isAttached)
    {
        self.attachedLabel.text = @"";

        _isAttached = NO;
    }
    else
    {
        self.attachedLabel.text = @"Attached";

        _isAttached = YES;
    }
}
```

By toggling the `isAttached` boolean you can specify whether the user has chosen to add an image with their tweet or not, then you inform the user with your `attachedLabel`, nothing too long or complicated.

Finally you arrive to the heart and soul of your `TweetViewController`, the `tweetTapped` method, Here's the code for the method:

```
- (IBAction)tweetTapped
{
    self.successLabel.text = @"";

    // Store The url for the mentions call (we are getting the
    // results in JSON format)
    NSURL *feedURL;

    // If the user selected to attach the image then use the
    // update_with_media call, otherwise use the regular update call.
    if (_isAttached)
    {
        feedURL = [NSURL
URLWithString:@"https://upload.twitter.com/1/statuses/update_with_
media.json"];
    }
}
```

```
else
{
    feedURL = [NSURL
URLWithString:@"http://api.twitter.com/1/statuses/update.json"];
}

// Dictionary with the call's parameters
NSDictionary *parameters = [NSDictionary
dictionaryWithObjectsAndKeys:self.statusTextField.text, @"status",
@"true", @"wrap_links", nil];

// Create a new TWRequest, use the GET request method, pass in
our parameters and the URL
TWRequest *twitterFeed = [[TWRequest alloc]
initWithURL:feedURL
parameters:parameters
requestMethod:TWRequestMethodPOST];

// If the user has selected to attach an image then add it to
our request
if (_isAttached)
{
    [twitterFeed
addMultiPartData:UIImagePNGRepresentation([UIImage
 imageNamed:@"TweetImage.png"]) withName:@"media"
type:@"image/png"];
    [twitterFeed addMultiPartData:[self.statusTextField.text
dataUsingEncoding:NSUTF8StringEncoding] withName:@"status"
type:@"text/plain"];
}

// Get the shared instance of the app delegate
AppDelegate *appDelegate = [[UIApplication sharedApplication]
delegate];

// Set the twitter request's user account to the one we
downloaded inside our app delegate
twitterFeed.account = appDelegate.userAccount;

// Enable the network activity indicator to inform the user
we're downloading tweets
UIApplication *sharedApplication = [UIApplication
sharedApplication];
sharedApplication.networkActivityIndicatorVisible = YES;
```

```
// Set our _isAttached BOOL to NO and attachedLabel to an empty
string.
_isAttached = NO;

self.attachedLabel.text = @"";

// Perform the twitter request
[twitterFeed performRequestWithHandler:^(NSData *responseData,
NSHTTPURLResponse *urlResponse, NSError *error) {
    if (!error)
    {
        dispatch_sync(dispatch_get_main_queue(), ^{
            self.successLabel.text = @"Tweeted Successfully";

            self.statusTextField.text = @"";
        });
    }
    else
    {
        dispatch_sync(dispatch_get_main_queue(), ^{
            // In case we couldn't perform the twitter request
            // successfully then show the user an alert view with the error's
            // description
            UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
            [alertView show];
        });
    }
}

// Stop the network activity indicator since we're done
downloading data
sharedApplication.networkActivityIndicatorVisible = NO;
}];
```

The first thing you do is clear the `successLabel`'s text, then you create an `NSURL` object but instead of instantiating it you use an `if` statement to determine the URL to use. If the user elected to attach an image then you use the `update_with_media` URL, otherwise you just use the `update` URL.

With your URL ready you create an `NSDictionary` indicating to wrap any links found in the tweet and passing in your status text. After this you create a `TWRequest`

object with the appropriate URL, parameters and the POST method, if you use multipart data then the parameters will be ignored, otherwise you will use the status in your `NSDictionary` to send the tweet.

After your `TWRequest` is ready you use an if statement to check whether the user selected to upload the image with the tweet, if they did then you use multipart data (as you saw a bit earlier in the chapter) to send the image and status text. This method requires authentication so we get our application's app delegate and pass in its `userAccount` property to the twitter request.

Finally you show the network activity indicator, clear our `attachedLabel`'s text and perform the request. Inside the request's completion handler we check for errors and display them using an alert view, if no errors occurred then you clear the status text field and set the `successLabel`'s text to "Tweeted Successfully".

Go ahead and run your project and try sending a tweet and the attached image:



With that you are done with the `TweetViewController` and your application! ☺

Where to go from here?

The source code included with the book has the finished project with some visual tweaks and enhancements. I also wanted to point out another way to use the `TWRequest` class in case you already have Twitter implemented in your application and are using `NSURLConnection`. In order to use `NSURLConnection` with `TWRequest` you must create your request as usual but instead of calling

`performRequestWithHandler:` you must use `signURLRequest` to retrieve an OAuth compatible `NSURLRequest` object with the info from your request.

I hope you have enjoyed learning about Twitter as much as I did writing this chapter. There's a lot of things you can do with the Twitter API, from simple integration within your application to creating a full-fledged Twitter client.

The best part about all of this is that it's built right into iOS, you no longer have to use external APIs or libraries for OAuth authorization and then manually sending your requests to Twitter, it can now be done with Cocoa and Objective-C.

I cannot wait to see all the awesome Twitter applications and uses in your future apps!



Chapter 14: Beginning Newsstand

By Steve Baranski

One of the most interesting features that was introduced in iOS 5 was Newsstand, a special folder intended for newspapers, magazines, and other periodicals.



If you have installed any Newsstand apps on your devices, you may have noticed a few unique characteristics about Newsstand:

- While iBooks is a single app that presents a variety of types of static content (such as ePub, PDF, etc), the Newsstand folder contains multiple apps that manage the presentation of their own content. The notion of a folder as a container for multiple apps is consistent with the rest of iOS.
- Newsstand icons defy the “rounded rectangle” shape found in iOS. They can be vertically oriented rectangles (like a magazine) or horizontally oriented rectangles (like a newspaper).
- Like their real world metaphors, Newsstand icons change with the availability of new “issues.” This is a powerful visual cue – it informs users that new content is available, and keeps users engaged with your app.

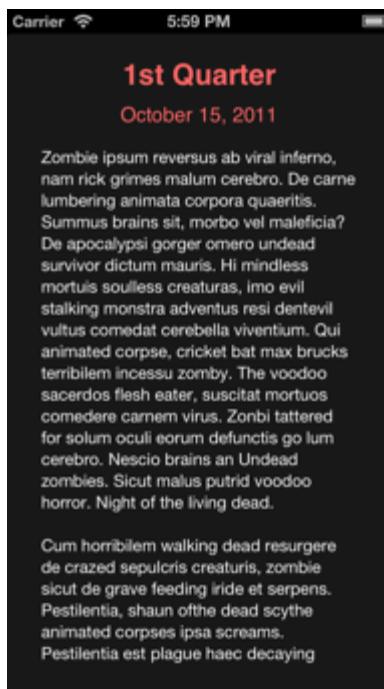
- The folder contains a “Store” link to purchase additional Newsstand apps. This can be a good way to get additional exposure for your app.

In these two chapters, you will work on a rudimentary app for retrieving blog content. You’ll make the app visible in Newsstand, configure it to receive push notifications when new content is available, modify it to retrieve the new content, and let the user know that the new content is available.

Getting started

Now that you know a bit about what Newsstand is, let’s take a look at the project that you will be converted into a Newsstand app!

In the resources for this chapter you will find a **Zombie Quarterly** starter project. Open it in Xcode and run it in the simulator, and you’ll see the following:



As you can see, this is a fairly straightforward iPhone app designed to retrieve RSS content from <http://zombiequarterly.tumblr.com>, an emerging blog for the undead.

Feel free to look through the code to get an idea of how the app works. It’s a fairly simple implementation – you’ll see the view controller simply consists of a few labels and a text view, currently configured to display the oldest post.

Note: This example is somewhat contrived. Using RSS to deliver static content is admittedly much simpler than the complex content delivery workflows of

major newspapers and other world class periodicals. Fortunately for us, zombies don't have a lot to say.

In this chapter, you'll focus on configuring Zombie Quarterly for inclusion in the Newsstand folder, and set the app up to be able to receive a new type of push notification indicating the availability of updated content.

Appearing in Newsstand

To characterize your app as a Newsstand app, you must add an entry to your project's Info.plist.

Open **Supporting Files\Zombie Quarterly-Info.plist**, select the bottom row, right-click, and select **Add Row**. Select the option entitled **Application presents content in Newsstand**, then choose **YES** as the value.

Your Info.plist should now look like the following:

Key	Value
▼ Information Property List	(17 items)
Localization native development region	en
Bundle display name	Zombie Q
Executable file	\$(EXECUTABLE_NAME)
► Icon files (iOS 5)	(1 item)
Bundle identifier	com.razeware.zombies
InfoDictionary version	6.0
Bundle name	\$(PRODUCT_NAME)
Bundle OS Type code	APPL
Bundle versions string, short	1.0
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone environment	YES
Main storyboard file base name	MainStoryboard
Icon already includes gloss effects	YES
► Required device capabilities	Related Files (1 item)
► Supported interface orientations	(3 items)
Application presents content in Newsstand	YES

For those that prefer the "raw" key-value pairs (which you can see by right clicking the file and selecting Open As\Source Code), the entry looks like this:

```
<key>UINewsstandApp</key>
<true/>
```

Let's try it out! Delete the app from the Simulator if it's installed already, then build and run to re-install it on the Simulator. After the app launches, hit Shift-Cmd-H to return to the home screen, and tap on the Newsstand folder. You should see your app's icon front and center!



While it's nice to see your app's icon in the Newsstand chapter, it appears that it's actually regressed a bit – your icon has lost its rounded corners! Overall, it looks pretty bland.

Fortunately, Newsstand allows you to tailor the appearance of an icon to more closely resemble a magazine or book. To do so, you need to edit the Info.plist further.

Back in **Zombie Quarterly-Info.plist**, select the **Icon files (iOS 5)** entry and click the **+** button. It should automatically create a new entry for Newsstand Icon for you by default. Click the down arrow to expand it, then also expand the Icon files array. Double click the entry for **Item 0**, and enter **issue_01.png**.

At this point, your file should look like this:

Key	Type	Value
Information Property List	Dictionary	(17 items)
Localization native development reg	String	en
Bundle display name	String	Zombie Q
Executable file	String	\$(EXECUTABLE_NAME)
Icon files (iOS 5)	Dictionary	(2 items)
Newsstand Icon	Dictionary	(3 items)
Icon files	Array	(1 item)
Item 0	String	issue_01.png
Binding type	String	Magazine
Binding edge	String	Left
Primary Icon	Dictionary	(2 items)

Again, for those of you who prefer the “raw” key-value pairs, here is what those look like:

```
<key>CFBundleIcons</key>
<dict>
    <key>UINewsstandIcon</key>
    <dict>
        <key>CFBundleIconFiles</key>
        <array>
            <string>issue_01.png</string>
        </array>
        <key>UINewsstandBindingType</key>
        <string>UINewsstandBindingTypeMagazine</string>
        <key>UINewsstandBindingEdge</key>
        <string>UINewsstandBindingEdgeLeft</string>
    </dict>
    <key>CFBundlePrimaryIcon</key>
    <dict>
        <key>CFBundleIconFiles</key>
        <array>
            <string>Icon.png</string>
            <string>Icon@2x.png</string>
        </array>
        <key>UIPrerenderedIcon</key>
        <true/>
    </dict>
</dict>
```

Here you've configured several aspects related to the Newsstand icon:

- You set the icon itself to issue_01.png, which is bundled with the project.
- You set the icon's "binding type", which allows you to choose whether you want your icon styled like a newspaper or magazine. In the case of a newspaper, the OS applies a fold shadow and stack effect to your icon. In the case of a magazine, it simulates pages on one side and staples on the opposite side. For *Zombie Quarterly*, you want magazine style.



- You set the icon's binding edge, which controls the placement of the staples (in the case of a magazine) or the fold (for a newspaper). You can choose between left, right, or bottom. For Zombie Quarterly, you want the left.

It's important to note that this specialized icon appears in both Newsstand and the app switcher "drawer". The standard icons, however, are still necessary – they are presented in Notifications, Settings, and Spotlight (Search).

Build and run the app to see how it looks! After the app launches, use Cmd-Shift-H to get to the home screen, and expand the Newsstand folder. You should be greeted by a drooling zombie - eek!



It's worth noting that the icon sizes are fairly flexible; you can create them to match the aspect ratio of your publication. For Zombie Quarterly, you are using icons that are 180x252. Apple's [Technical Note TN2280](#) (Newsstand FAQ) provides the following guidance:

On iPhone, the length of the longest edge of your Newsstand app icon should be at least 90 pixels. On iPhone 4 retina display your icon dimensions should be doubled, meaning that the longest edge should be at least 180 pixels. On iPad the longest edge of your Newsstand app icon should be at least 126 pixels.

In all cases, the longest edge may be a horizontal or vertical edge. If the longest edge is horizontal, the aspect ratio should not exceed 2:1. If the longest edge is vertical, the aspect ratio should not be smaller than 1:2.

Background behavior

Newsstand alerts your app of new content via push notifications. Unless your reader is actively using your app when new content arrives, your app will likely be in the background.

In order for your app to be able to process new content when these push notifications arrive (to do things such as downloading the new issues), you need to set a background mode flag in your Info.plist.

This means you have to make one more revision to **Zombie Quarterly-Info.plist**. Select the bottom row, right-click, and select **Add Row**. Select the option entitled **Required background modes**, click the arrow to expand the row, and for **Item 0** select **App processes Newsstand Kit downloads** from the dropdown.

Here's what it looks like:

▶ Supported interface orientations	⊕ ⊖ (3 items)
Application presents content in Newsstand	YES
▼ Required background modes	(1 item)
Item 0	⊕ ⊖ App processes Newsstand Kit downloads

Again, for those of you who prefer the "raw" key-value pairs, here is what those look like:

```
<key>UIBackgroundModes</key>
<array>
    <string>newsstand-content</string>
</array>
```

Apple states that the arrival of this notification only provides you with a few precious seconds to respond to this notification. If your app needs more time to respond, you can secure up to 10 minutes via a call to `[UIApplication beginTaskWithExpirationHandler:]`.

Where to go from here?

So far you have the app displaying properly in the Newsstand app, but you still have a long way to go!

In the next chapter, you'll focus on notifying the app of new content via push notifications. You'll also learn more about the NewsstandKit framework itself, and leverage the components of the framework to manage content retrieval, content organization, and new content alerts.

Chapter 15: Intermediate Newsstand

By Steve Baranski

In the previous chapter, you learned how to configure an app to show up in Newsstand by simply adding some properties to your Info.plist.

That's a good start, but you have a lot more work to do to make it a fully functional Newsstand app!

The first thing you need to do is notify your app when new content becomes available. To do so, you'll employ a familiar iOS mechanism – push notifications. It's worth noting, however, that push notifications typically occur as alerts visible to the user; Newsstand more subtly delivers notifications, updating app content and icons behind the scenes.

Because this chapter is focused on Newsstand and not push notifications, you are going to set up push notifications via Urban Airship, a third party provider of infrastructure for push notifications and other tools. They currently offer a free "Basic" plan that will suffice for this chapter.

Configuring your app

Before you set things up at Urban Airship, you must configure a new app in the iOS Provisioning Portal. Here are the steps:

1. From your computer, sign in to the **iOS Developer Center** at <http://developer.apple.com/ios> and click **iOS Provisioning Portal** on the right side of the screen.

The screenshot shows the 'Development Resources' section of the iOS Dev Center. On the left, there's a sidebar with links like 'Downloads', 'Getting Started Videos', 'Getting Started Documents', and 'iOS Developer Library'. In the center, there's a 'Featured Content' section with links such as 'Create Apps for iOS 6', 'What's New in iOS 6', 'Passbook for Developers', etc. On the right, there's a 'iOS Developer Program' sidebar with links like 'iTunes Connect', 'Apple Developer Forums', 'Developer Support Center', 'App Store Resource Center', 'Prepare for App Submission', and 'App Store Approval Process'. The 'iOS Provisioning Portal' link is highlighted with a red circle.

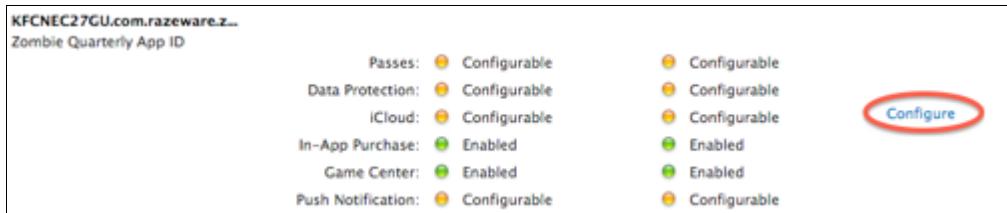
2. Click **App IDs** on the left sidebar, then **New App ID** on the upper right.

The screenshot shows the 'Provisioning Portal : Razeware LLC' interface. The left sidebar has links for 'Home', 'Certificates', 'Devices', and 'App IDs' (which is highlighted with a red circle). The main area shows a 'Manage' tab and a 'How To' section. Below that is a 'App IDs' section with a sub-section for 'App IDs'. A large button labeled 'New App ID' is circled in red.

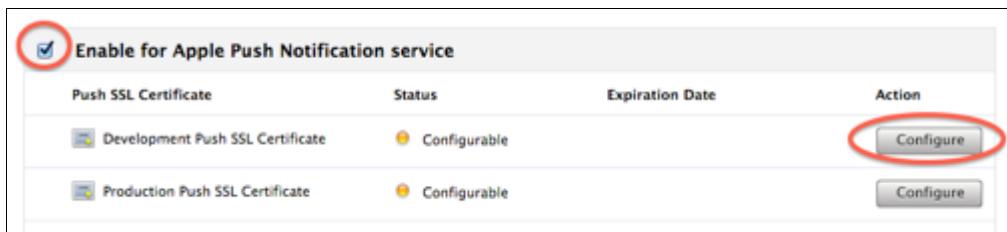
3. Enter **Zombie Quarterly App ID** for the Description and select **Use Team ID** for the Bundle Seed ID. Enter **com.razeware.zombiequarterly** for the Bundle Identifier, except substitute your own name/company name for razeware. This value must match your Bundle identifier in Zombie Quarterly-Info.plist – later on, you'll fix that up to match. For now you're done, so click **Submit**.

The screenshot shows the 'Create App ID' form. It has three main sections: 'Description' (with placeholder text about alphanumeric characters), 'Bundle Seed ID (App ID Prefix)' (with a dropdown for 'Use Team ID' and a note about sharing keychain access), and 'Bundle Identifier (App ID Suffix)' (with a text input field containing 'com.razeware.zombiequarterly' and an example placeholder). At the bottom are 'Cancel' and 'Submit' buttons.

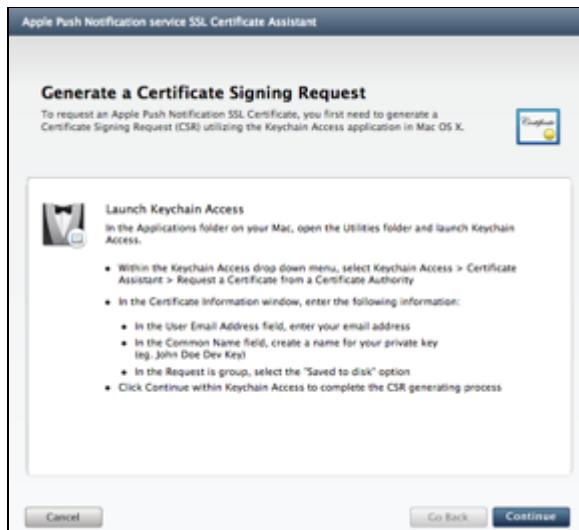
4. Find your new App ID in the list of App IDs, and click the **Configure** button to the right.



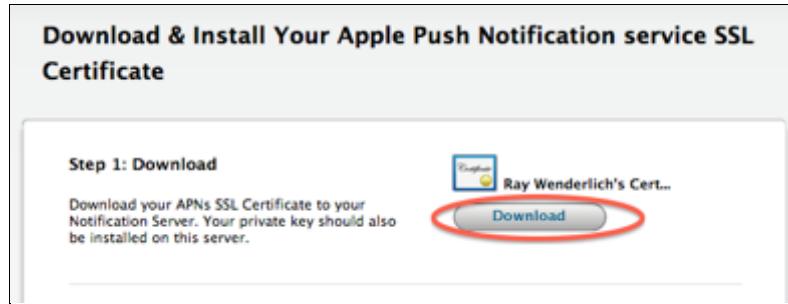
5. Click the checkbox next to **Enable for Apple Push Notification service**, then click **Configure** next to Development Push SSL Certificate.



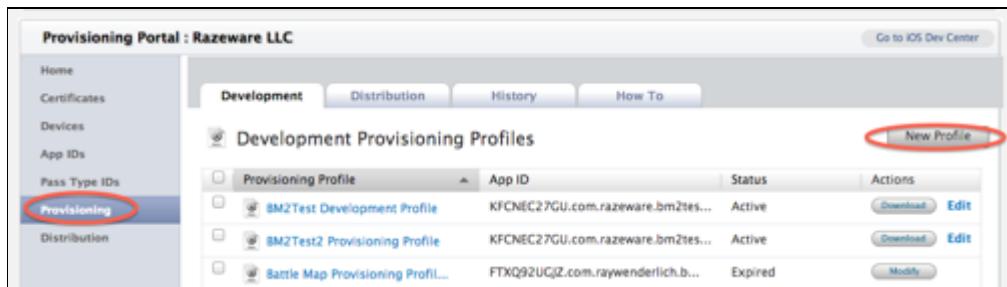
6. The Apple Push Notification service SSL Certificate Assistant will then appear, and guide you through the process of creating the certificate required for push notifications. Follow the steps to create a certificate.



7. Once you've finished with the assistant, click the **Download** button to fetch your newly created certificate. After the download completes, double click the *.cer file to import it into the Keychain Access application on your Mac.



8. You'll also need to create a Provisioning Profile that reflects the fact that your app has been configured to use push notifications. To do so, click on **Provisioning** in the sidebar, and then **New Profile**.



9. Enter **Zombie Quarterly Dev Profile** for the Profile Name, check your certificate, choose **Zombie Quarterly App ID** for the App ID, and select all of your devices. Finally, click **Submit**.

Profile Name	Zombie Quarterly Dev Profile
Certificates	<input checked="" type="checkbox"/> Ray Wenderlich
App ID	Zombie Quarterly App ID
Devices	<input type="checkbox"/> Deselect All <input checked="" type="checkbox"/> [Device Icon] <input checked="" type="checkbox"/> [Device Icon] <input checked="" type="checkbox"/> [Device Icon] <input checked="" type="checkbox"/> [Device Icon]

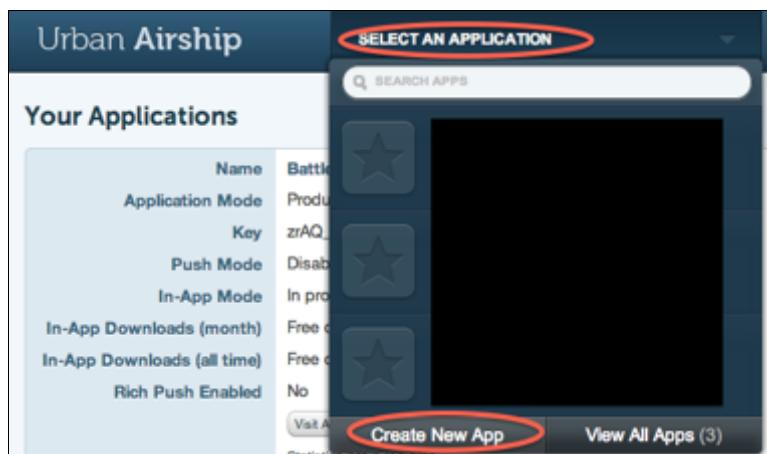
10. Refresh the screen until the Download button appears, then download the provisioning profile and double click it to add it into Xcode. This should be familiar to you if you've submitted apps to the App Store in the past.

Now that you've configured your app in the iOS Provisioning Portal, you are ready to configure it at Urban Airship for push notifications!

Configuring Urban Airship

To enable the ability to send push notifications to your app through Urban Airship, complete the following steps:

1. If you do not already have an account at Urban Airship, create one at <https://go.urbanairship.com/accounts/register/>.
2. Sign in to your account.
3. Click the **Select an Application** box at the top of the screen, and choose **Create New App**.



4. Create a new app according to the following guidelines:
 - For **application name**, enter Zombie Quarterly App.
 - For **application mode**, select Development.
 - For **category**, select Newsstand.
 - Click the **Push Enabled** checkbox.
 - For the **Apple push certificate**, recall that you created it in the Provisioning Portal and imported it into Keychain Access. To share it with Urban Airship, you need to export it from Keychain Access and import it here. To do so, open Keychain Access and select login from the Keychains pane in the upper left-hand corner of the screen. In the primary pane, you should see an entry that reads something like "Apple Development iOS Push Services: com.razeware.zombiequarterly" – right click that and choose Export. Save the Certificates.p12 to disk (with no password) and close Keychain Access.
 - Click the **Push debug mode** checkbox.
 - Finally, click **Create your application**.

Create an application

Application name	Zombie Quarterly App
Application Icon	Choose File No file chosen
Application Mode	Development - connecting to testing servers.
Category	Newsstand
Push Enabled	<input checked="" type="checkbox"/>
Apple push certificate	Choose File Certificates.p12
Certificate password	
Push debug mode	<input checked="" type="checkbox"/> What's this?

- Once the app is created, you'll see a new screen that displays the app details. You'll need the **Application Key**, **Application Secret**, and **Application Master Secret** values later on (use the Click to show buttons to reveal as necessary), so record them somewhere handy.

Key	Value
Information Property List (18 items)	
Localization native development reg	en
Bundle display name	Zombie Q
Executable file	\$(EXECUTABLE_NAME)
Icon files (iOS 5) (2 items)	
Bundle identifier	com.razeware.zombiequarterly

- Next, it's time to set up your Xcode project. Open **Zombie Quarterly-Info.plist**, and set your **Bundle identifier** equal to what you set when you created your App ID:

Key	Value
Information Property List (18 items)	
Localization native development reg	en
Bundle display name	Zombie Q
Executable file	\$(EXECUTABLE_NAME)
Icon files (iOS 5) (2 items)	
Bundle identifier	com.razeware.zombiequarterly

- Zombie Quarterly already includes the Urban Airship iOS library, so you just need to enter a few configuration settings. Open **AirshipConfig.plist**, and replace the values of **DEVELOPMENT_APP_KEY** and **DEVELOPMENT_APP_SECRET** with the **Application Key** and **Application Secret** values you got earlier.

8. Finally, give this a quick test! Run the app on your device (not the simulator), and click OK if a popup about Push Notifications appears. This should automatically send a unique ID for your device to Urban Airship.
9. Tap the home button to bring the app to the background – this way when you send a push notification later you'll see a popup (but you won't if the app is in the foreground).
10. Back in Urban Airship, go to Push\Device Tokens, and check to see if you have a new entry. If you do not, check your console log to look for any errors, and double check that your Bundle Identifier and AirshipConfig.plist is set up properly.

Urban Airship

ZOMBIE QUARTERLY APP

Tools

- Push Composer
- Rich Push Composer
- Segments
- Reports

Zombie Quarterly App

Details

Edit

Statistics

Push

Test Push Notifications

Send Broadcast

Device Tokens

APIs

Feeds

DEVICE TOKENS

Device Token
1D1D20A23BE10A3CACA300234EB58FCC9C31A8CDE62DAE02A60810D795C49AD1

Alias
None

Status
Active

Last Registration
0 minutes ago

Tags
None

More

It can be helpful during development to clear out all of your device tokens, especially if you API. Would you like to do that?

Yes, delete all device tokens

11. Copy that device token, and go to **Push\Test Push Notifications**. Paste the Device token in, enter **Hello from iOS 5 by Tutorials!** as the Alert, then click **Send it!**

The screenshot shows the Urban Airship dashboard interface. On the left, there's a sidebar with 'Tools' (Push Composer, Rich Push Composer, Segments, Reports) and 'Zombie Quarterly App' (Details, Edit, Statistics, Push). Under 'Push', options include Test Push Notifications, Send Broadcast, Device Tokens, APIDs, and Feeds. The main area is titled 'TEST PUSH NOTIFICATIONS' with a 'PROTIP: Something not working right? Check the error console.' message. It has tabs for iOS, Android, BlackBerry, and Helium for Windows. For iOS, fields include 'Device token' (1D1D20A23BE10A3CAC300234EB58FCC9C31A8CDE62C), 'Alias' (empty), 'Badge' (empty), 'Alert' ('Hello from iOS 5 by Tutorials!'), 'Sound' (empty), and 'Payload' ({"aps": {"alert": "Hello from iOS 5 by Tutorials!"}, "device_tokens": ["1D1D20A23BE10A3CAC300234EB58FCC9C31A8CD"]}). A 'Send it!' button is at the bottom.

12. Back on your device, you should see a popup appear with the message you just sent!



Congrats, you now know everything is working OK with Urban Airship with normal push notifications – it's time to transition to newsstand push notifications instead!

Handling newsstand notifications

These chapters have been somewhat unique in that you've written very little code so far. That is about to change!

Back in your project, open **ZQAppDelegate.m**, and add the following line to the top of `application:didFinishLaunchingWithOptions:`:

```
[ [NSUserDefaults standardUserDefaults] setBool: YES  
    forKey:@ "NKDontThrottleNewsstandContentNotifications" ];
```

I mentioned earlier that the Newsstand notification is special – it's intended to deliver content at occasional intervals (as opposed to frequent "breaking news"). Perhaps because it's special, iOS limits the frequency of this notification to once per day. The preceding line allows you to circumvent that limit while testing the app.

In the same method, find the call to `registerForRemoteNotificationTypes:` and replace it with the following:

```
[ [UAPush shared] registerForRemoteNotificationTypes:  
    UIRemoteNotificationTypeNewsstandContentAvailability];
```

Previously the project had some text code that registered to receive standard push notifications – this app doesn't need those anymore, it only cares about newsstand notifications.

Next add this new method to the file:

```
- (void)application:(UIApplication *)application  
didReceiveRemoteNotification:(NSDictionary *)userInfo {  
    NSLog(@"application:didReceiveRemoteNotification: %@",  
        userInfo);  
  
    [[NSNotificationCenter defaultCenter] postNotificationName:  
        @"com.razeware.zombies.newsstand.notificationReceived"  
        object:self];  
}
```

When the app receives a notification, this method will be called. All it does is log out some info to the console, and send a notification which you'll listen for later.

Let's try if this works! Run the app on your device, and tap OK if a popup appears.

Then back in Urban Airship go to **Push\Test Push Notifications** as before. But this time, modify the Payload to include this small tweak right before the "alert":

```
"content-available": 1,
```

So it should look like this:

The screenshot shows the Urban Airship dashboard with the 'ZOMBIE QUARTERLY APP' selected. In the left sidebar under 'Push', 'Test Push Notifications' is selected. The main area shows a form for sending a push notification to an iOS device. The 'Device token' field contains '1D1D20A23BE10A3CACA300234EB58FCC9C31A8CDE62C'. The 'Alert' field contains 'Hello from iOS 5 by Tutorials!'. The 'Payload' field contains the JSON object: `{"aps": {"content-available": 1, "alert": "Hello from iOS 5 by Tutorials!"}, "device_tokens": ["1D1D20A23BE10A3CACA300234EB58FCC9C31A8CDE62C"]}`. A 'Send It!' button is at the bottom.

Click **Send it!**, then back in Xcode look to see if you got a message like this in the console:

```
2012-10-04 20:07:42.521 Zombie Quarterly[4910:907]
application:didReceiveRemoteNotification: {
    "_" = "rvBF2g6AAEeK6GBT+tdMgpg";
    aps = {
        alert = "Hello from iOS 5 by Tutorials!";
        "content-available" = 1;
    };
}
```

If you did, it means your Newsstand notification is working correctly! Now it's time to do something interesting when this occurs.

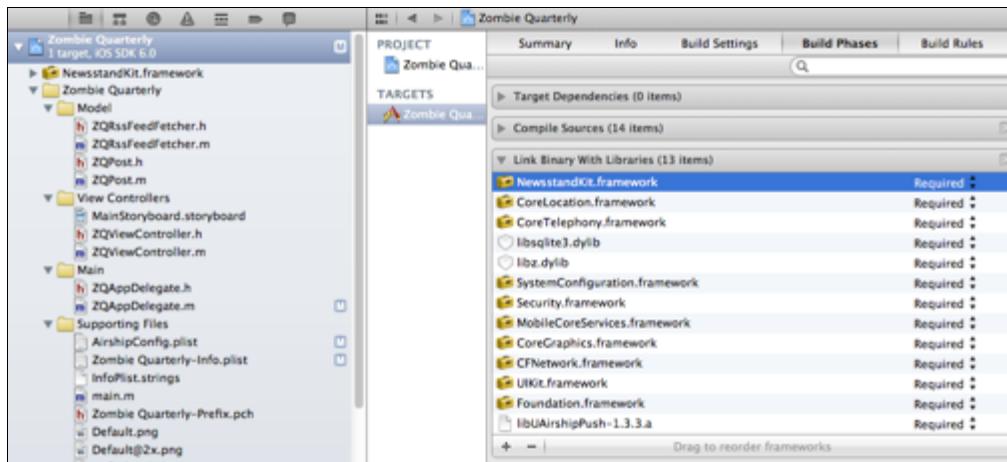
Integrating Newsstand Kit

In addition to adding your app to the special Newsstand folder and giving you a special notification about new content, Newsstand provides your app with a set of APIs to manage information about the content you've downloaded – such as what issues are available, which one you're currently reading, and so on.

Of course, in order to use these APIs you need to add a framework to your Xcode project – the NewsstandKit framework.

To do so, select the project in the Project Navigator and select the **Zombie Quarterly** target. Select the **Build Phases** tab, and expand the **Link Binary with**

Libraries section. Click the **+** button to add a new framework, select **NewsstandKit.framework**, and click **Add**.



Now time for some code! The goal is to synchronize the local list of content (blog posts retrieved from the RSS feed) with the list of “issues” Newsstand knows about, tell it what issue you’re currently reading, and manage icon changes.

Let’s start with **ZQRssFeedFetcher.m**. First import the NewsstandKit header at the top of the file:

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

Then add a new private instance variable to keep track of whether or not you received a Newsstand notification:

```
BOOL _notified;
```

And add the following inside the if statement in `initWithContentURL:delegate::`

```
[ [NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(handleNotification:)
    name:@"com.razeware.zombies.newsstand.notificationReceived"
    object:nil];
```

Remember that earlier you added code to send this notification when a newsstand push notification arrives. Here you listen for this notification, and when it occurs the `handleNotification:` method will be called.

So add the implementation of that method next:

```
- (void)handleNotification:(NSNotification *)notification {
    _notified = YES;
    [self fetch];
```

```
}
```

This simply sets the `_notified` flag to `YES`, and starts fetching the feed.

But what's that flag for? After successfully parsing the RSS feed, you'll check for this flag, and if it is set you'll do some newsstand processing. To do this, add the following code in `fetch`, right before the line `[_delegate fetchSuccess:_posts]`:

```
if (_notified) {
    [self updateNewsstand];
    _notified = NO;
}
```

And for the final piece of the puzzle, implement `updateNewsstand` as follows:

```
- (void)updateNewsstand {
    NKLibrary * library = [NKLibrary sharedLibrary];
    for (ZQPost * post in _posts) {
        NKIssue * issue = [library issueWithName:post.header];
        if (!issue) {
            [library addIssueWithName:post.header
                date:post.date];
        }
    }
}
```

Newsstand maintains a repository of issues for bookkeeping purposes. The central part of the framework is `NKLibrary`, which is accessed via the singleton method shown above.

The code then loops through the posts received from the RSS feed, and syncs up this local content with the library. Zombie Quarterly treats each blog post as an "issue" of content with respect to Newsstand, so each post has its own `NKIssue` representing it. Of course, what determines an "issue" of content is completely up to your own app.

This code checks to see if there's an issue already with the blog post's title, and if not creates a new one and adds it to the library.

That's it for `ZQRssFeedFetcher` – just need to make a few changes to `ZQViewController`. Open **ZQViewController.m** and import the `NewsstandKit` header at the top of the file:

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

Then modify `fetchSuccess:` as follows:

```
- (void)fetchSuccess:(NSArray *)posts {
```

```
    ZQPost * post;
    NKLibrary * library = [NKLibrary sharedLibrary];
    if (library.issues.count > 0) {

        // Get most recent issue
        post = posts[library.issues.count - 1];
        NKIssue * currentIssue = [library
            issueWithName:post.header];

        // Set issue as being read
        if (currentIssue) {
            [library setCurrentlyReadingIssue:currentIssue];
        }

        // TODO: Update icon
        // TODO: Add +1 to badge

    } else {
        post = posts[0];
    }

    self.titleLabel.text = post.header;
    self.subtitleLabel.text = [_dateFormatter
        stringFromDate:post.date];
    self.bodyTextView.text = post.body;

}
```

This code checks to see if there are any issues in the Newsstand library. If there are, then you know that the `updateNewsstand` method you wrote earlier has been called at least once in response to a Newsstand notification. So when this occurs, instead of getting the oldest post it gets the most recent post. It also uses NewsstandKit's API to mark the current issue that is being read.

Finally, replace the two TODOs above with the following code:

```
// Update icon
NSURL * imageURL = [NSURL URLWithString:post.iconUrl];
UIImage * currentIcon = [UIImage imageWithData:[NSData
    dataWithContentsOfURL:imageURL]];
[[UIApplication sharedApplication]
    setNewsstandIconImage:currentIcon];

// Add +1 to badge
NSInteger unreadCount = [[UIApplication sharedApplication]
```

```
applicationIconBadgeNumber];
[[UIApplication sharedApplication]
setApplicationIconBadgeNumber:unreadCount + 1];
```

The RSS posts have a category attribute that includes a URL to the image for the new magazine cover. Here you use this attribute to download the image and designate it as the new Newsstand icon for the app.

After that, you add the “New” sash by incrementing the application icon badge number for the app. As with any other app, the badge (sash) will display for any non-zero value.

As a final step, open **ZQAppDelegate.m** and add this line into `applicationWillEnterForeground:` to clear the badge when the app launches:

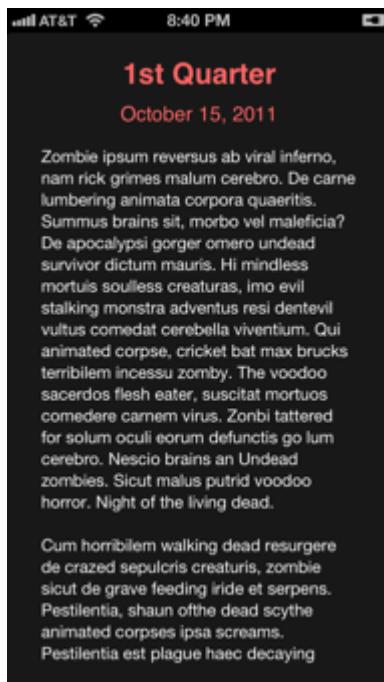
```
[[UIApplication sharedApplication]
setApplicationIconBadgeNumber:0];
```

And that’s it – finally it’s time to see it all come together!

Putting it all together

Again, you have to perform this test on your device, so make sure your iPhone or iPad are connected. Then complete the following steps:

1. Delete Zombie Quarterly from Newsstand.
2. Run the app. You should see the “1st Quarter” entry as before.



3. Now tap the Home button to enter the background. Leave Xcode connected to the suspended instance of your app.
4. Open a web browser and return to Urban Airship. Go to Push\Test Push Notifications and enter the a test payload with the content-available flag set, just like you did before:

The screenshot shows the Urban Airship dashboard for the "ZOMBIE QUARTERLY APP". The left sidebar has sections for Tools (Push Composer, Rich Push Composer, Segments, Reports), Zombie Quarterly App (Details, Edit, Statistics), and Push (Test Push Notifications, Send Broadcast, Device Tokens, APIDs, Feeds). The main area is titled "Test Push Notifications" with a "PROTIP: Something not working right? Check the error console." message. It has tabs for iOS, Android, BlackBerry, and Helium for Windows. Under the iOS tab, there are fields for Device token (1D1D20A23BE10A3CACA300234EB58FCC9C31A8CDE62C), Alias, Badge, Alert (Hello from iOS 5 by Tutorials!), Sound, and Payload. The Payload field contains the JSON: {"aps": {"content-available": 1, "alert": "Hello from iOS 5 by Tutorials!"}, "device_tokens": ["1D1D20A23BE10A3CACA300234EB58FCC9C31A8CDE62C"]}. A "Send it!" button is at the bottom.

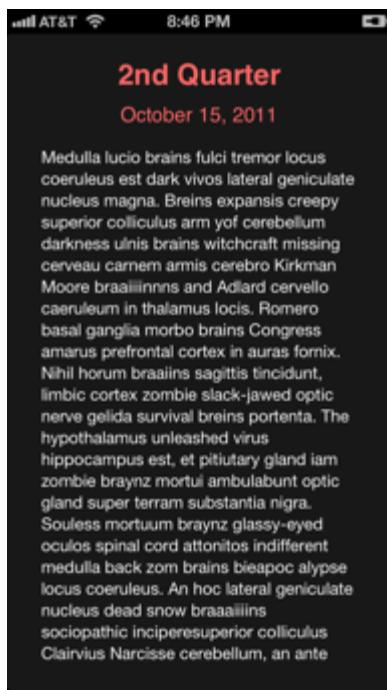
5. Upon receipt of the push notification, the Newsstand folder will update to reflect a badge number:



6. Open the folder, and you'll see a new magazine cover with a happy zombie styling a "New" sash:



7. And perhaps most importantly, if you open the app, you'll see the latest post, entitled "2nd Quarter":



Great success!

Where to go from here?

That wraps up our coverage of Newsstand. Because our content delivery scheme was fairly simplistic, this implementation will likely differ from that of a major publication. For example, we haven't covered `NKAssetDownload`, a class which provides support for managing more complex content payloads. Still, now you have a solid basis with Newsstand and can continue your investigations from here.

As you begin to explore Newsstand yourself, here are some other references that may prove useful:

- Apple has a page titled "Newsstand for Developers" – it servers as a "one stop shop" for resources on the topic:
<http://developer.apple.com/devcenter/ios/newsstand/>
- If you want to use Urban Airship for your project, Urban Airship has some Newsstand helper classes that may be useful. Check this guide as a starting point: <https://support.urbanairship.com/customer/portal/articles/140033-adding-newsstand-support-to-your-app>
- If you would prefer to use an approach other than Urban Airship for push notifications, check out this tutorial for a starting point:
<http://www.raywenderlich.com/3443/apple-push-notification-services-tutorial-part-12>

Best of luck with your continued adventures into Newsstand, and be sure to let me know if you end up creating anything cool! ☺

Chapter 16: Beginning UIPageViewController

By Felipe Laso Marsetti

Back in January 2010, when Apple announced the iPad they also unveiled iBooks, an eBook reading app that makes full use of the iPad's larger screen and form factor. By now all of you have at least seen iBooks in action if not used it on your own.

Apple maintains that whenever possible, developers should attempt to make their apps feel just as a real physical object for users to interact with. That theme can be seen throughout iOS with apps like Calendar, Address Book, Photos and more.

iBooks allows users to flip through pages by dragging their fingers across the screen while getting beautiful visual feedback of a page folding and flipping over, just as if reading a printed book.

Surely I'm not alone when I say that ever since iBooks and iOS 4 were announced, developers have hoped for similar functionality to be incorporated as part of the iOS SDK with every update. Others have attempted to build their own custom implementation for page flipping and other iBooks goodies.

Luckily for you, the wait is over with the release of iOS 5! Apple now provides the `UIPageViewController` that allows you to create your own iBooks-like interfaces and apps with many different styles and customization options.

This chapter will cover everything about `UIPageViewController`: its methods, data source, delegate, and the template provided for you in Xcode. In the next chapter you'll dig into even more detail, and build a project that uses the `UIPageViewController` from the ground up!

So flip the page and let's get started!

Understanding how UIPageViewController works

Before you delve into any code or specifics about the `UIPageViewController`, let's talk about View Controller Containment. As part of iOS 5, Apple now allows developers to use `UIViewController`s as containers for other view controllers, thus enabling the creation of rich and complex user interfaces or perhaps an app that wasn't very easy (or possible at all) without this new functionality.

Some examples of containers provided by Apple are:

- `UINavigationController`
- `UITabBarController`
- `UISplitViewController`

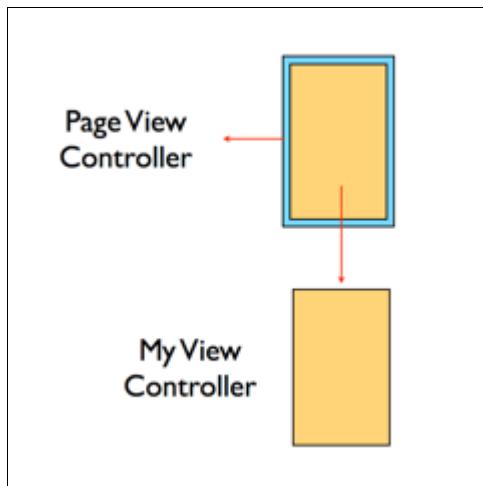
All of these are containers that have one or more children view controllers. If you think about it, a navigation controller is a container that has a stack of view controllers and takes care of pushing and popping them according to user input. Similarly with a tab bar controller or a split view controller, you have containers of many view controllers each with its own look and functionality.

If you want to know more about view controller containment and how to implement this new functionality in your apps then refer to Chapter 22, "UIViewController Containment".

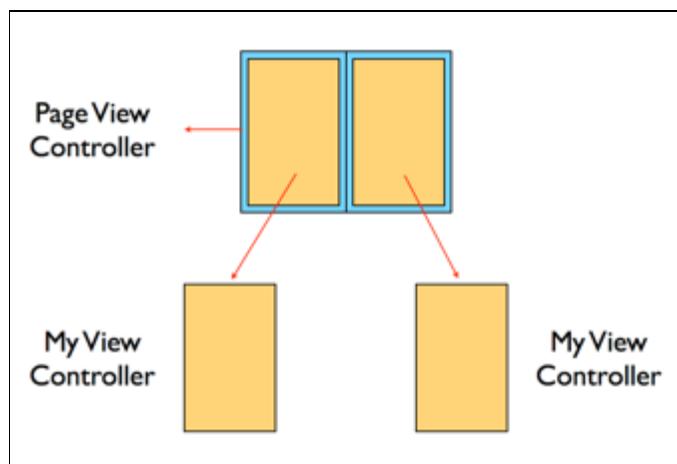
As you might have guessed, the new `UIPageViewController` is a container that will allow you to replicate the look and functionality of the iBooks app while showing your own custom content.

There are many ways in which you can customize your page view controller, from using vertical or horizontal page flipping to showing one or two pages side by side - it's all possible in iOS 5.

The way `UIPageViewController` handles containment depends on whether you are displaying one or two pages of content on your device. Take a look at the images below so you get a better understanding of how things work:



The page view controller is only displaying a single child view controller.



The page view controller is displaying two view controllers side by side (like an open book).

So the page view controller is the container for your view controllers, simple enough right?

That's pretty much half the battle in order to understand the `UIPageViewController`, the other part relates to the spine location and navigation orientation when flipping pages. The spine location is where each "page" is anchored when the user flips it and the navigation orientation specifies whether the user is flipping pages vertically or horizontally.

The spine can be located in the following places:

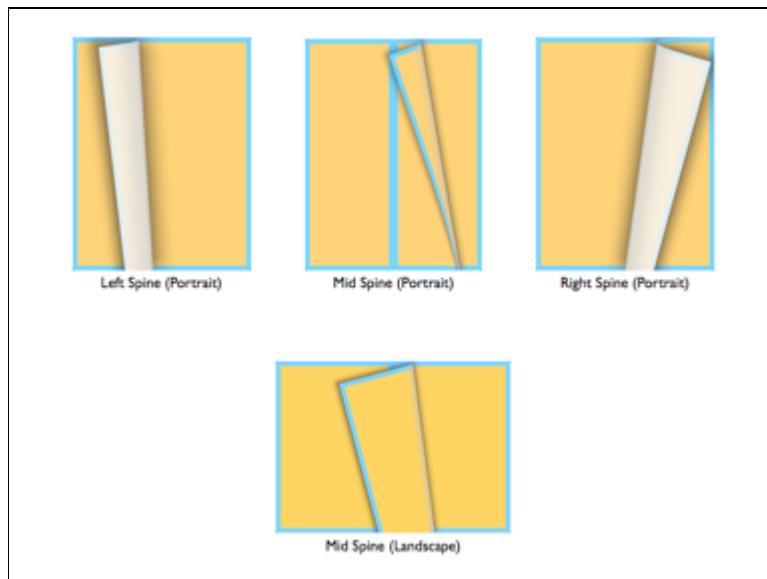
- Left
- Right
- Top
- Bottom

- Middle

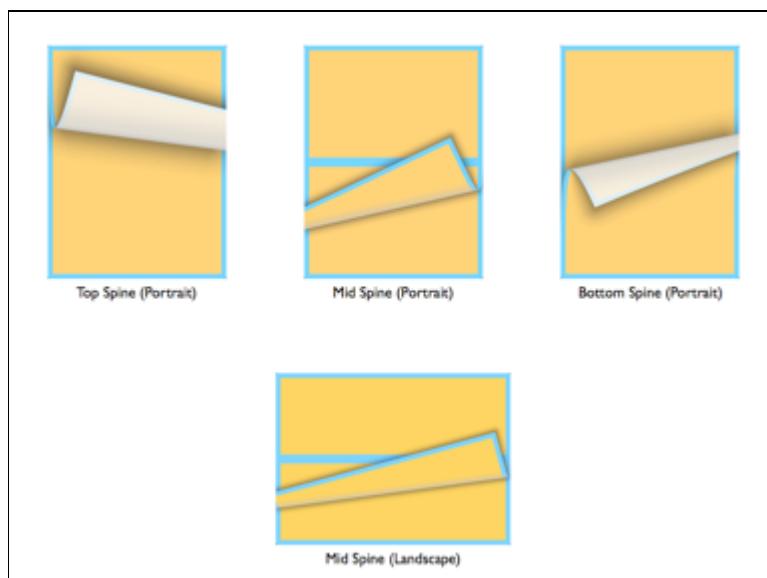
You can even change the spine location depending on the interface orientation of the device, allowing you, for example, to show a single page with the spine on the left when the user is holding the device in portrait mode and the spine on the middle when the device is in landscape, just like iBooks does.

Here are some of the possible combinations of spine locations and navigation orientations:

Horizontal Navigation:



Vertical Navigation:

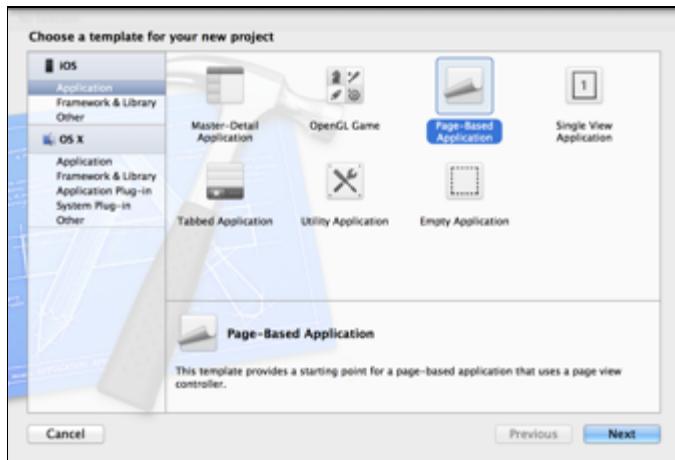


Awesome! You have pretty much gone over all of the theory behind the page view controller, one small thing to keep in mind is that the API doesn't actually use "Top", "Bottom", "Left" or "Right" as keywords for the spine location, but as you will see in a moment it's very simple and easy to learn.

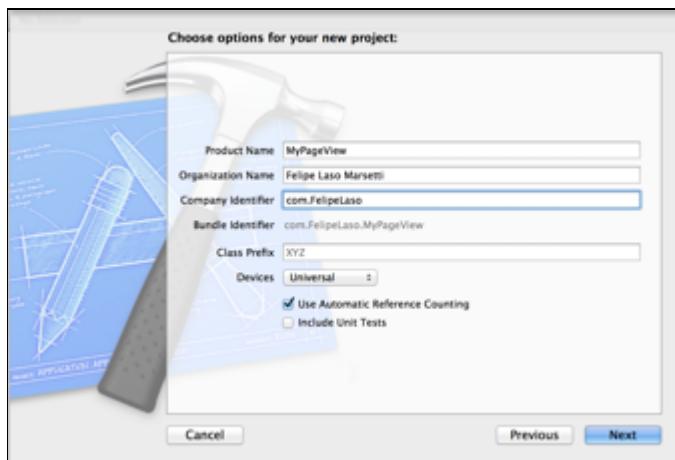
Creating a UIPageViewController project

Now that you've covered the concepts and theory behind view controller containers and the page view controller, it's time to get your hands dirty and work on some actual code.

Fire up Xcode and go to **File/New/New Project**, alternatively you can use the keyboard shortcut of Shift + Command + N. From the menu select **iOS/Application** on the left side and choose the **Page-Based Application template**, go ahead and click Next.

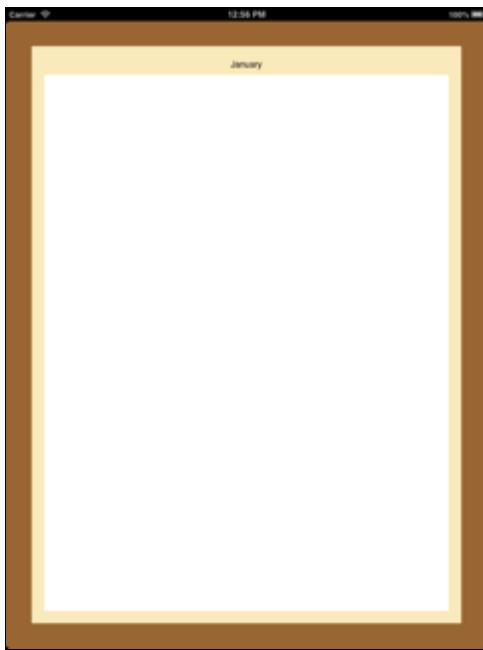


On the next screen name your project **MyPageView**, give it a company identifier in case you want to run your app on a device, make sure **Universal** is set for Device Family and select **Use Automatic Reference Counting**.



Go ahead and click **Next**, save your project and create a local Git repository if you'd like to (I highly recommend you develop a habit of making a git repository for everything you code).

Run the project so you can see what the template has provided and so you can play around a bit with the page flipping and such.



Leave the project settings as they are. The device family has to be iOS 5.0 and above since `UIPageViewController` is not available in earlier versions, and the supported orientations are fine for what you will do.

In the Project Navigator you can see that Xcode has created several classes as well as two storyboard files for you interface, one for iPhone and one for iPad.

Let's go over each of the files and see what's been setup for you. Open up **AppDelegate.h** and the only thing you will see here is the standard `UIWindow` property, notice how it's declared as strong instead of retain, this is part of ARC so be sure to read up about it in the Beginning and Intermediate ARC chapters in this book.

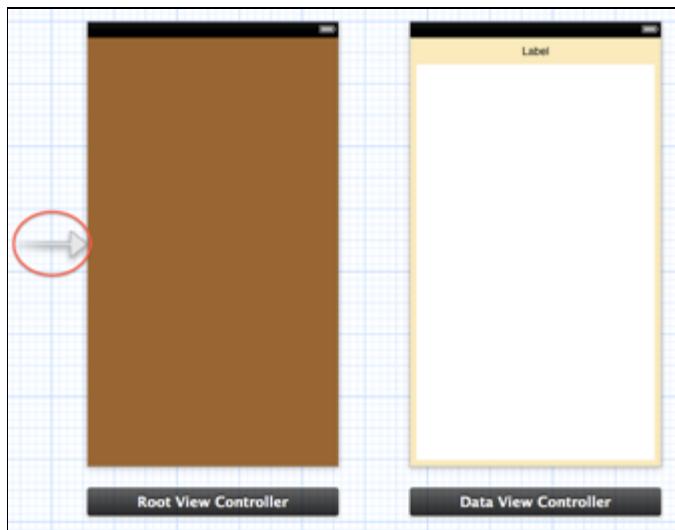
One thing to notice is that the `window` property is not declared as an outlet so it's not connected to anything in your storyboards, this is automatically setup for you.

Jump over to **AppDelegate.m** file notice the `applicationDidFinishLaunchingWithOptions:` method no longer has all the setup of adding a subview and calling `makeKeyAndVisible`. As you may have read in the Beginning and Intermediate Storyboards chapters, what Apple is trying to do with storyboards is reduce the amount of code necessary to achieve some trivial and common things, this is one of them.

Before checking out other files let's see how storyboards for this particular project are set up. Start with the iPhone storyboard - open up `MainStoryboard_iPhone.storyboard` and take a look!

The first thing you'll notice is the two view controllers: one is Root View Controller and the other is Data View Controller. In order to know which is the initial view controller you can see which one has the arrow pointing to it, in your case it's Root View Controller (hence the name).

Here is an image of your iPhone storyboard file:



Notice how, compared to other projects using storyboards, there is no relationship or segue connecting the `RootViewController` with the `DataViewController`. Just because you have storyboards doesn't mean all view controllers have to be connected in some way.

Select the Data View Controller, open up the Identity Inspector and notice that it has "DataViewController" set as the Storyboard ID. With storyboards you can give scenes a unique identifier so you may programmatically instantiate them, just like you would have done with NIBs by calling `initWithNibName:bundle:`.

This is how the project template is set up as far as the interface goes. When the root view controller is loaded, it will create a page view controller, and add data view controller objects to it by using the identifier setup in Interface Builder.

To view this in action open `RootViewController.h` and see what you've got:

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

@interface RootViewController : UIViewController
<UIPageViewControllerDelegate>

@property (strong, nonatomic) UIPageViewController
*pageViewController;
```

```
@end
```

As you can see there's a `UIPageViewController` property with the `strong` attribute, the equivalent of `retain` using ARC, and you add the `UIPageViewControllerDelegate` protocol to the `RootViewController` class.

Note that under the modern runtime, you no longer have to declare your instance variables within the class block and then a property for them, you can simply declare a property and the runtime will take care of synthesizing and creating the property's instance variable for you.

Head over to `RootViewController.m` next to continue you investigation. First you have some imports necessary for the classes you are going to use and communicate with, next up you have an unnamed category for your `RootViewController` that adds a property of type `ModelController`:

```
@interface RootViewController ()  
  
@property (readonly, strong, nonatomic) ModelController  
*modelController;  
  
@end
```

A category is a part of the Objective-C language, nothing complicated if you want to read up about it. All this is doing is adding a private property for the model controller, so don't be scared by the syntax if this is new to you!

The `ModelController` is a class provided to you by the template that is the page view's data source and feeds it with the necessary info as the user navigates. In the implementation the properties are synthesized as usual.

Scrolling down a bit you will find a custom implementation of the `modelController` property getter which verifies if you have already instantiated a model controller, otherwise it allocates one and returns it.

Let's see what's going on in the `viewDidLoad` method:

```
- (void)viewDidLoad  
{  
    // 1  
    [super viewDidLoad];  
  
    // 2  
    self.pageViewController = [[UIPageViewController alloc]  
initWithTransitionStyle:UIPageViewControllerTransitionStylePageCur  
1
```

```
navigationOrientation:UIPageViewControllerNavigationOrientationHorizontal
options:nil];
self.pageViewController.delegate = self;

// 3
DataViewController *startingViewController =
[self.modelController
viewControllerAtIndex:0
storyboard:self.storyboard];
NSArray *viewControllers = @*[startingViewController];
[self.pageViewController setViewControllers:viewControllers
direction:UIPageViewControllerNavigationDirectionForward
animated:NO
completion:NULL];

// 4
self.pageViewController.dataSource = self.modelController;
[self addChildViewController:self.pageViewController];
[self.view addSubview:self.pageViewController.view];

// 5
CGRect pageViewRect = self.view.bounds;
if ([[UIDevice currentDevice] userInterfaceIdiom] ==
UIUserInterfaceIdiomPad) {
    pageViewRect = CGRectInset(pageViewRect, 40.0, 40.0);
}
self.pageViewController.view.frame = pageViewRect;
[self.pageViewController didMoveToParentViewController:self];

// 6
self.view.gestureRecognizers =
self.pageViewController.gestureRecognizers;
}
```

There's a lot of code here, so let's go over it step by step so you understand how each part works.

1. This is the standard call to the super class's `viewDidLoad` method.
2. You create an instance of `UIPageViewController` and initialize it with some parameters. The transition style is, as the name implies, the way the page view controller will transition between view controllers. iOS 5 only has the page curl transition style, but perhaps Apple will add more in the future. The navigation orientation sets how the user flips through pages, either horizontally (like a book) or vertically (like a calendar). For the options you can pass in an `NSDictionary` indicating where the spine is located, you'll take a look at that bit later when you

create your own page view controller from the ground up. Finally you set the root view controller as the delegate of the page view controller.

3. A `DataViewController` instance is created using a method in the Model Controller. For now just be aware that this is creating a new data view controller for you.

After this you put the newly created data view controller into an array and call the page view controller's `setViewControllers:direction:animation:completion` method. This is the method to use when you want to change the view controllers being shown on your page view controller. Depending on whether you are showing two pages of content or just one, you pass an array with one or two view controllers. The direction simply states if the user is navigating forward or backward, this actually doesn't change anything as far interacting with the interface goes, but for the model this allows you to implement books or content in languages where you read from back to front or right to left. You can animate the setting of the view controllers and use a custom completion handler block if necessary.

4. Here you set the `modelController` property as the data source of the page view controller and add the page view controller as a child view controller, this is something new to iOS 5 and part of view controller containment. Finally you add the page view controller's view as a subview of the root view controller otherwise you would not see your page view controller on screen, just the root view controller's view.

5. Because you are adding the page view controller as a child of root view controller, you must set its frame so that you can define a width, height and position within the root view controller's view. If you don't use this you will probably see the page view controller when you run the app, but it might be offset or positioned incorrectly. The template includes a simple if statement to shrink the page view controller a little bit in case you are using an iPad. This is useful if you want to build a Twitter client, for example. You could divide the screen in half and show a table view with all of the tweets on top, and a page view controller on the bottom to show the selected tweet or user. If you just want to see the page view controller without making it smaller or seeing its super view you may remove this line. There's also a call to tell the page view controller it has moved to a new parent view controller. This is necessary when implementing a container view controller - for more information, see Chapter 22.

6. Before `viewDidLoad` finishes all of this setup, you add the page view controller's gesture recognizers to the root view controller's gesture recognizers.

The explanation for this is very simple, if you don't do this step then you will see your page view controller once your app loads but no interaction will occur when we tap or drag on the screen. This happens because you would be calling the root view controller's gesture recognizers (which at this point aren't configured to do anything) instead of the page view controller's gesture recognizers.

And that wraps up `viewDidLoad`, phew! Believe it or not you have covered a lot of things here, especially some crucial inner workings of the page view controller.

Moving on you can see that the `shouldRotateToInterfaceOrientation` method is allowing rotation on all orientations except portrait upside down for iPhone, and all orientations for iPad.

You're almost done with the `RootViewController` implementation, all you need to cover are the `UIPageViewControllerDelegate` methods, yay!

You might see a method commented out that begins with `didFinishAnimating...`. This method gets called after a gesture transition occurs, meaning when the user taps or drags across the screen and you show a new view controller (or pair of view controllers).

This method might come in handy if you want to do some coding after the user switches to a new page. The template doesn't use it but just know it's there as part of the API.

The method that gets implemented (and you should too on your own projects) is the `spineLocationForInterfaceOrientation`. There are four possible return values you can use here:

- `UIPageViewControllerSpineLocationNone`
- `UIPageViewControllerSpineLocationMin`
- `UIPageViewControllerSpineLocationMid`
- `UIPageViewControllerSpineLocationMax`

Remember you saw some graphics earlier about the possible combination of spine locations (left, right, top, bottom or middle) with navigation orientations (vertical or horizontal)?

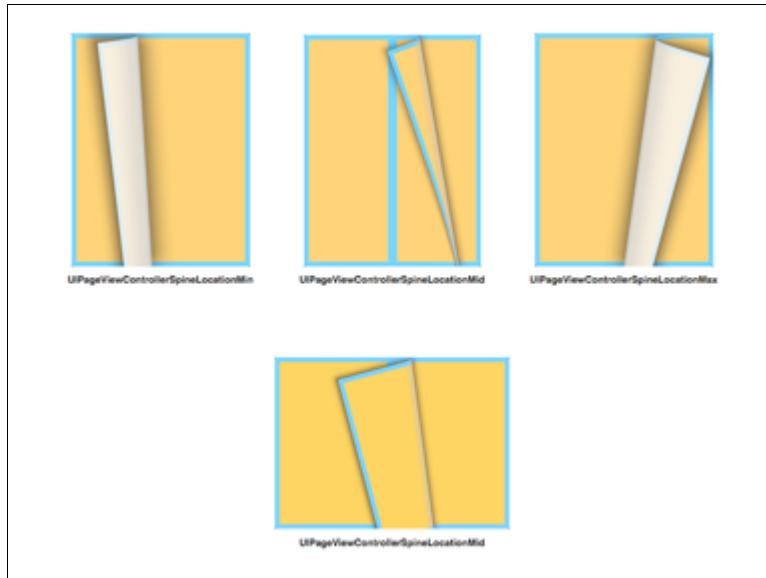
The API doesn't really use left, right, top or bottom but it's very easy to understand.

When navigating horizontally, `UIPageViewControllerSpineLocationMin` will position the spine on the left, `UIPageViewControllerSpineLocationMid` in the middle and `UIPageViewControllerSpineLocationMax` on the right.

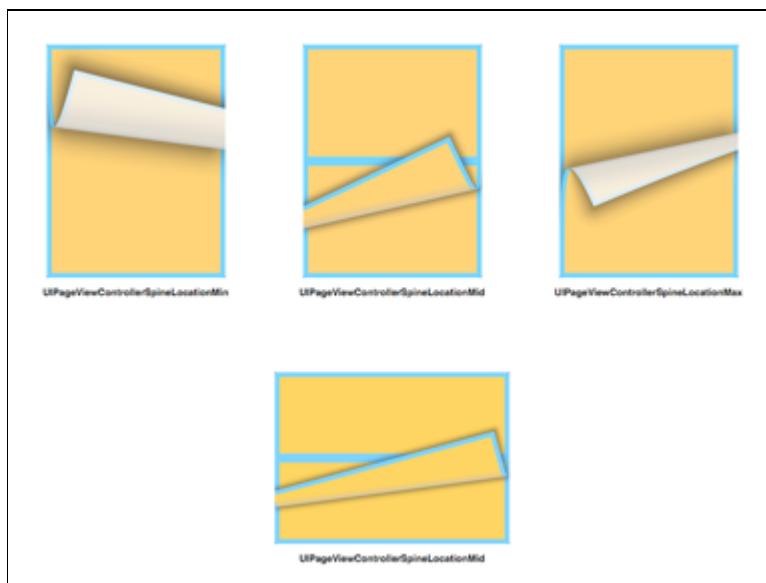
Similarly when navigating vertically `UIPageViewControllerSpineLocationMin` will position the spine at the top, `UIPageViewControllerSpineLocationMid` in the middle and `UIPageViewControllerSpineLocationMax` at the bottom.

Here are some images to show how this works:

Horizontal Navigation:



Vertical Navigation:



The chapter will not cover the code inside the `spineLocationForInterfaceOrientation` method because it just changes the spine location the left hand side when in portrait to middle when in landscape.

Some calculations are made to see which view controllers should be set (it could vary if the user is at the first view controller, last view controller or in between).

One interesting thing to keep in mind is the `doubleSided` property of the page view controller. This property is standard to all `UIPageViewController` objects and it just indicates whether you want content to appear on the backside of a page or not.

For the vertical orientation (like in iBooks for example) you don't want it to be double sided in order to get the semi-transparent effect you see when flipping a

page. When you are in landscape orientation with two view controllers being shown simultaneously then you set this to YES.

Keep in mind that this property is set to YES by default, so you should only use it when you don't want a page to be double sided (check out the method code and you'll see how it only sets this to NO when moving to a portrait orientation with a min spine location).

Woohoo! You are done with the `RootViewController` class, let's move on to the `ModelController` class to see how you are displaying some data using a page view controller.

Open up **ModelController.h** and let's see what you have:

```
@class DataViewController;

@interface ModelController : NSObject
<UIPageViewControllerDataSource>

- (DataViewController *)viewControllerAtIndex:(NSUInteger)index
Storyboard:(UIStoryboard *)storyboard;
- (NSUInteger)indexOfViewController:(DataViewController *)
viewController;

@end
```

First up there's a forward declaration for the `DataViewController` class (which you will learn about in a bit), next you will notice that `ModelController` implements the `UIPageViewControllerDataSource` protocol and it declares two public instance methods.

The `viewControllerAtIndex:` method will return a new `DataViewController` instance with the appropriate information for the given index, `indexOfViewController:` will return an `NSUInteger` with the index of the `DataViewController` object you pass into the method.

Let's go over to the implementation file to see how these two methods work as well as which methods you have to implement as the page view controller's data source.

Open up **ModelController.m**. At the top, you'll see a category to declare a private, strong, read only `NSArray` property named `pageData`, it will only be used internally which is why it's been made private.

In the `init` method all that's happening is you are getting the name of the twelve months of the year and saving each into your `pageData` array.

Switch over to the `viewControllerAtIndex:storyboard:` method. This is quite an interesting method because you'll see an example of how to programmatically instantiate view controllers that you have created in your storyboard files (as you

may have also seen in Chapters 4 and 5, "Beginning and Intermediate Storyboards").

```
- (DataViewController *)viewControllerAtIndex:(NSUInteger)index
storyboard:(UIStoryboard *)storyboard
{
    // Return the data view controller for the given index.
    if ([self.pageData count] == 0) || (index >= [self.pageData
count])) {
        return nil;
    }

    // Create a new view controller and pass suitable data.
    DataViewController *dataViewController = [storyboard
instantiateViewControllerWithIdentifier:@"DataViewController"];
    dataViewController.dataObject = self.pageData[index];
    return dataViewController;
}
```

The if statement is included to check for a valid index for your view controller, in case you can't create one you return nil. Outside of the if statement, a `DataViewController` object is created by calling `instantiateViewControllerWithIdentifier:` on your local storyboard property.

Because the model controller class doesn't know about your storyboard, you pass it in from the root view controller and use it to create your data view controller instance.

`indexOfViewController:` simply receives a `DataViewController` object and looks up the index of its local `dataObject`, that way you know where you should navigate to, and where your pages begin and end.

Next up are the `UIPageViewControllerDataSource` methods, there are two of them; `viewControllerBeforeViewController:` and `viewControllerAfterViewController:`.

These methods receive a `UIViewController` object and you use the `indexOfViewController:` method to know where in your application you are and what data you must put into the returning view controller.

It's quite simple, though for production apps you might want to validate the class of the view controller passed into these methods. Otherwise when you try to cast them you might be calling a method that is unavailable to that particular view controller.

Believe it or not that's all you must cover inside the Model Controller! So let's wrap up by taking a look at the final class - `DataViewController`. Open up `DataViewController.h` and take a look inside:

```
@interface DataViewController : UIViewController
```

```
@property (strong, nonatomic) IBOutlet UILabel *dataLabel;  
@property (strong, nonatomic) id dataObject;  
  
@end
```

This is a standard `UIViewController` subclass with two strong properties declared, one is an `IBOutlet` of type `UILabel` that will connect with the label on your storyboard and the other is an object that will hold the month information that will be set on the label.

Quite simple right? It gets even better, open up the `DataViewController.m` file and look at what's inside. The `dataLabel` is set to the `dataObject`'s `description` (which returns the name of the month) inside of `viewWillAppear:`.

Where to go from here?

That's it! You've completed the tour of the Page View Controller template, and you should now understand how it works well enough to start modifying it for use in your own projects.

But if you feel like you need a bit more information before diving in, keep reading. In the next chapter, you'll learn how to use a `UIPageViewController` in your project from scratch, and use it to build a fun photo-viewing app!

Chapter 17: Intermediate UIPageViewController

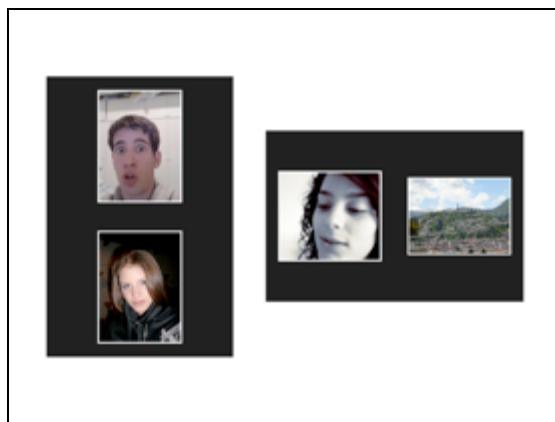
By Felipe Lasso Marsetti

In this chapter, you are going to create a `UIPageViewController` project from scratch, without using the template. You might be thinking it's very redundant to pretty much repeat what the template does, but rest assured this is a worthwhile and fun learning experience!

If you ever want to make a highly customized app, perhaps by using view controller containers, setting up a navigation controller inside a tab bar or so on, then knowing how to start a project from scratch is fundamental.

While recreating the template, not only will you understand how everything works, you are going to make a very cool photo album in the process so that you may show your pictures or content in a beautiful, elegant and intuitive way.

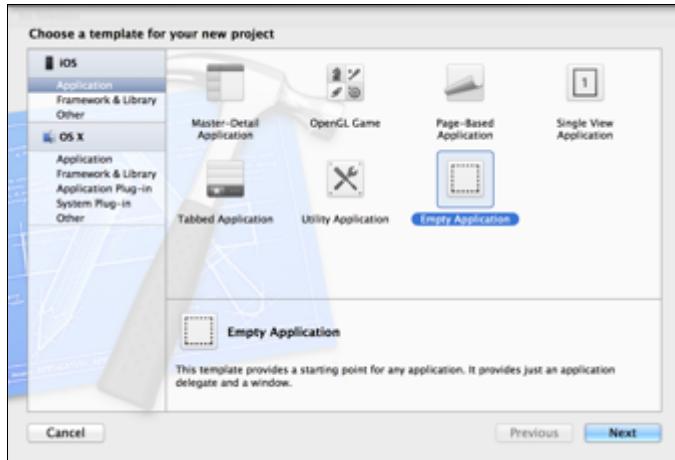
Here's a preview of the app you're going to build:



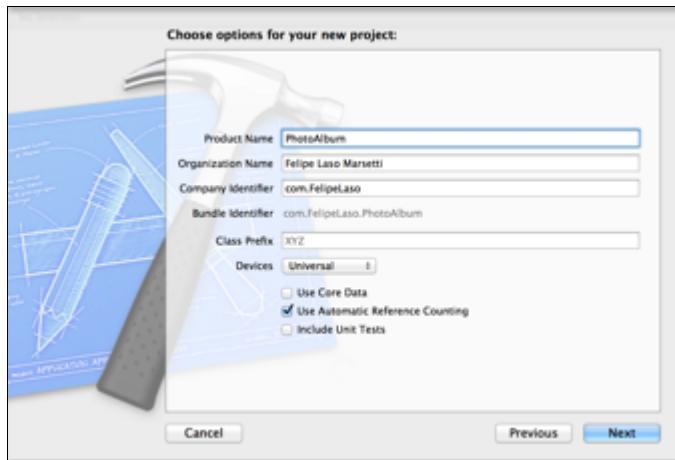
Getting Started

Right, the time has come for you to get your fingers typing and building a beautiful, interactive photo album.

Open up Xcode and create a new project by going to **File\New\New Project**. From the template window select the **iOS\Application\Empty Application** project template.



Go ahead and click **Next**, for the Product Name use **PhotoAlbum**, use your own identifier so you may run the app on your devices, leave the class prefix field empty, select **Universal** for the Devices and make sure you select **Use Automatic Reference Counting**.



Click **Next** one more time, save your project wherever you like and, as a personal suggestion, create a git repository for your project to take advantage of Xcode's built in version control support.

You now have a blank project with just an application delegate that you can customize and start from scratch. If you see the settings for your project everything is fine for now, you have the orientations, iOS version and everything else setup.

Note: Notice in the project settings you have an empty field where you can select either a storyboard or main interface file for iPhone and iPad, for now leave those blank as you will be creating your storyboards in a bit.

The first thing you are going to do is create a view controller that will be the root of your view hierarchy and also instantiate a `UIPageViewController`.

Go ahead and right-click the **PhotoAlbum** folder and select **New File**, from the new window go ahead and choose **iOS/Cocoa Touch/Objective-C class**. Name your class **RootViewController**, make sure in Subclass of you have **UIViewController** selected and uncheck both Target For iPad and With XIB for user interface. Click **Next** and save your file.

In `RootViewController.m` the template automatically added a `initWithNibName:bundle:` method, you don't need it for your project so go ahead and delete it.

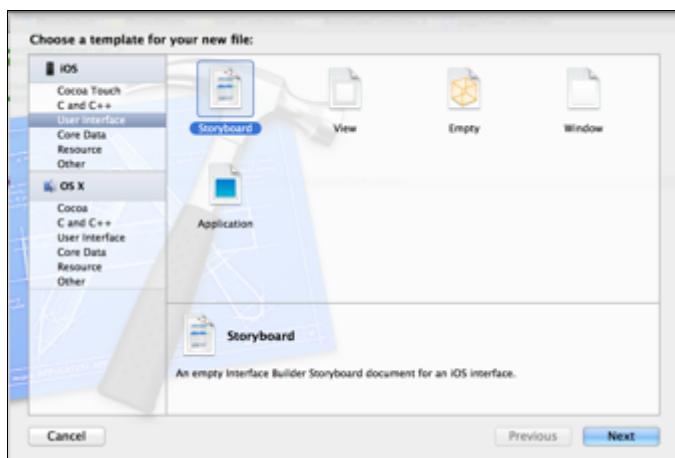
Next open `RootViewController.h` and add a property for a `UIPageViewController` right after the `@interface` declaration:

```
@property (strong, nonatomic) UIPageViewController  
*pageViewController;
```

Awesome! You have a page view controller property that you can now use throughout the project.

Before moving on, create your storyboards so that you can instantiate a root view controller and have it act as the container (parent) of your `UIPageViewController`. You will have one storyboard for iPad and one for iPhone/iPod Touch devices.

Control-click on the **PhotoAlbum** folder (or if you prefer create subfolders and organize your files as you like) and select **New File**. Choose the **iOS/User Interface/Storyboard** template, and click **Next**.



Start with **iPad** for Device Family so make sure to have that selected, click **Next** again, name your storyboard **Storyboard_iPad.storyboard** and click **Create**. You should now have a storyboard file in your Project Navigator.

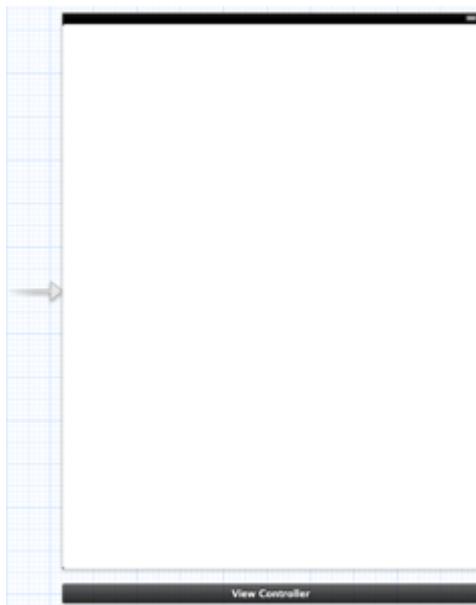
Now you need to repeat the same steps for the iPhone, so right-click your **PhotoAlbum** folder, select **New File**, choose the **iOS/User Interface/Storyboard** template, and click **Next**. Under Device Family make sure you choose **iPhone**, click **Next** and name your file **Storyboard_iPhone.storyboard**. Finally click **Create** and you should see your iPhone storyboard in the Project Navigator.

Great! You now have both of your storyboards ready to be set up. Because both the iPhone and iPad versions of your interface will have the same layout and contents you'll have to repeat the same process for both. I know it's a bit tedious but luckily you only have to do this once and it will keep you from having to write a lot of code.

Note: You can show or hide the Utilities by clicking this button (see below) on the top right corner of the Xcode window. Alternatively, you can use the keyboard shortcut of CONTROL + OPTION + COMMAND + 3.



Open up **Storyboard_iPad.storyboard**, and drag a `UIViewController` object from the library and drop it onto the canvas, notice how Interface Builder automatically added an arrow on the left side of your view controller, this indicates it's the initial view controller.



Select your newly created view controller and open the Identity Inspector (the keyboard shortcut for this is OPTION + COMMAND + 3). In the Storyboard ID field write `RootViewController` and for the class write `RootViewController`. After you've done this, go to the Attributes Inspector and set the Status Bar to None under the Simulated Metrics section.

For your case you don't really need to use the Identifier but it's good practice when using storyboards.

Drag another view controller from the Object Library (this will be the actual pages of your photo album), set the simulated status bar to None, and set the Storyboard Identifier to `AlbumPageViewController`.

If you have read chapters 4 & 5 on Storyboarding, you will notice you don't have a segue or relationship between the view controllers. Don't worry about this because you can programmatically instantiate them using the identifier you just set. Adding the view controllers into a storyboard like this just helps you write less code and design your interface within a single file.

You will not be setting the class for your new view controller because you haven't created one yet.

Drag a new view inside the newly created view controller and size it so that it fits within the blue interface guidelines, in the Attributes Inspector change the background color to something other than white (so you may differentiate it with the view controller's view).

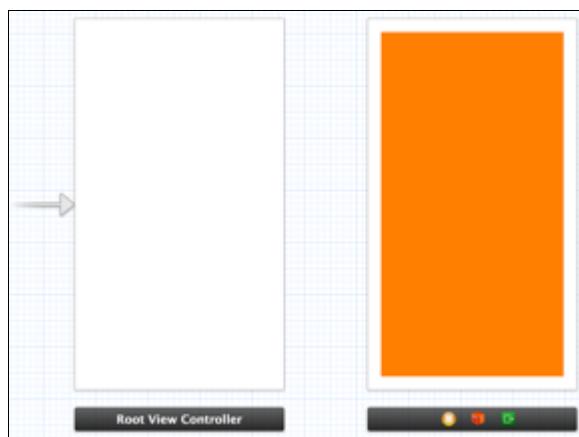
Don't worry if this doesn't make much sense right now, this is to show you why it's necessary to set the page view controller's frame when instantiating it. Just put a sticky on your mind to remind yourselves of this for now!

Now repeat the same steps within `Storyboard_iPhone.storyboard`. Drag a view controller from the Object Library, remove the simulated status bar, set the Storyboard ID to `RootViewController` and then inside the Identity Inspector change the class to `RootViewController`.

Then drag another view controller, remove the simulated status bar, set its Storyboard ID to `AlbumPageViewController`, add a view to the view controller you just created and change its background color.

Once thing you need to remember to do for this project is to turn off Autolayout for both storyboards. To do so click anywhere in your storyboard's canvas and in the File Inspector deselect the box that says Use Autolayout.

Awesome, your storyboard files should now look as follows:



Don't worry if the colors aren't the same, the important thing is that your view controllers are setup correctly with the appropriate identifiers.

If you recall, earlier when you first created the project, you didn't have a storyboard or interface file setup in the project settings. Let's go ahead set up your project now to use the storyboards you just created.

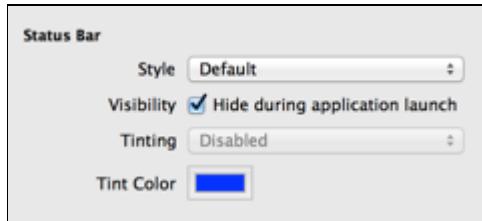
In the Project Navigator select the PhotoAlbum Xcode project, select the PhotoAlbum target and make sure you are in the Summary tab. Inside there you should see a field that says Main Storyboard for both the iPhone/iPod Touch and iPad builds. Use the drop down to select the corresponding storyboard file or just type the name in the field.

This is what your project settings should look like for now:



Yay! Your storyboard files are now connected to the project and ready to be used. There are just a few small things you have to do before building and running the project.

In this window, also select the checkbox under the Status Bar section that says Hide during application launch.



Now jump over to the `AppDelegate.m` implementation file and just return `YES` inside the `applicationDidFinishLaunchingWithOptions:` method as follows:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    return YES;
}
```

Go ahead and run your project using either the iPad or iPhone simulator. For now all you'll see is an empty white window (without a status bar, yay!) but believe it or not you are actually using the storyboards. If you want to verify this just add a label or interface element to the root view controller inside your storyboards so you can see this.

Great, the project is now set up and ready to use a `UIPageViewController`.

Open RootViewController.m, delete all the methods and move on to `viewDidLoad` where you will create your `pageViewController`. Inside `viewDidLoad` add this code:

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

    NSDictionary *pageViewOptions = @{@"NSNumber
numberWithInteger:UIPageViewControllerSpineLocationMin" :
UIPageViewControllerOptionSpineLocationKey};

    self.pageViewController = [[UIPageViewController alloc]
initWithTransitionStyle:UIPageViewControllerTransitionStylePageCur
1

navigationOrientation:UIPageViewControllerNavigationOrientationHor
izontal

options:pageViewOptions];

    UIViewController *albumPageViewController = [self.storyboard
instantiateViewControllerWithIdentifier:@"AlbumPageViewController"];
    [self.pageViewController setViewControllers:[NSArray
arrayWithObject:albumPageViewController]

direction:UIPageViewControllerNavigationDirectionForward
            animated:NO
            completion:nil];
    [self addChildViewController:self.pageViewController];
    [self.view addSubview:self.pageViewController.view];
    [self.pageViewController didMoveToParentViewController:self];
}
```

This code probably looks simpler than what the page based template provided. It's still missing a few optional elements here and there but it should be familiar since you covered a lot of this back in the theory section.

Remember I told you you'd see how to pass in options to the `pageViewController`? Here it is, for now the API only supports the spine location option but in the future more could be added.

Because the value for the spine location is from an `enum`, and you can only put objects inside a dictionary, you must wrap it within an `NSNumber`.

Next up you create our page view controller and save that instance in the `pageViewController` property. In the initializer you pass in the transition style

(which in your case is the page curl), a navigation orientation of forward (because you read your books from left to right) and the options dictionary (or nil in case you don't want to set any).

After this you need to pass the initial view controllers for your page view controller, in order to do this you create a `UIViewController` instance using the `instantiateViewControllerWithIdentifier` method provided to you by the `UIStoryboard` class. Notice how you used the `AlbumPageViewController` identifier that was setup in Interface Builder.

This line might seem new and interesting:

```
self.storyboard
```

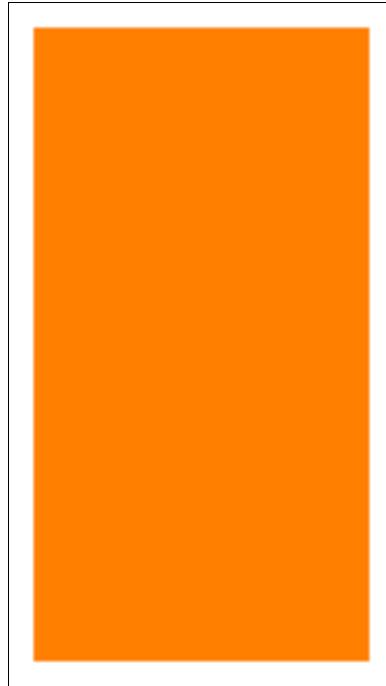
This property is available to all `UIViewController` instances, it's set to the storyboard where this view controller was loaded from. If you create a view controller programmatically (or using a NIB) then this property is simply nil.

After you create the view controller you proceed to set the view controllers for your page view controller. Inside this method you pass in an array of view controllers, the navigation direction, whether you want this action to be animated and a completion block.

Because you are creating the page view controller right after the app launches you don't want any animation to occur, when the user switches the spine location after changing the device orientation (meaning that you display two view controllers at a time) animating the transition might be a good idea.

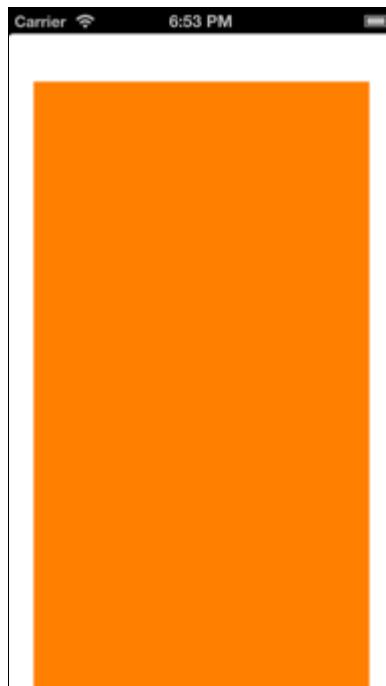
Finally, you add the `pageViewController` as a child of the root view controller and add the page view controller's view as a sub-view of the root view controller's view (phew, used the word `view` a lot in that paragraph!).

Go ahead and run your project now and you'll see something like this:



Yay! This is exactly what you desire. Now, take a moment to see one interesting aspect of view controller containment.

Remember how I told you to put a colored view within the view controller in your storyboard file? Things work fine right now, but if you wanted to use a status bar, you'd see something like this:



Oh no, your view controller is not centered properly! Fear not, with just one line you can fix the problem:

```
self.pageViewController.view.frame = self.view.bounds;
```

This little line just sets the page view controller's frame to be the same size as the root view controller's bounds. Keep this in mind for future projects (in case this happens to you) or when your status bar is being shown.

Expanding your app and the UIPageViewController

You want to create a custom subclass of `UIViewController` so you can handle laying out the album pages.

Control-click the **PhotoAlbum** folder, select **New File**, select the **iOS/Cocoa Touch/ Objective-C class** template, and click **Next**. Enter **AlbumPageViewController** for the Class, **UIViewController** for the Subclass, make sure Targeted for iPad and With XIB for user interface are both unchecked, and finish creating the file.

Open **AlbumPageViewController.m** and remove the `initWithNibName:` method, since it will not be used.

Then open **Storyboard_iPad.storyboard**, select the view controller and in the Identity Inspector change the class from a regular `UIViewController` to our `AlbumPageViewController` subclass. Repeat the same process for **Storyboard_iPhone.storyboard**.

Now go back to **RootViewController.m**. You need to change the regular `UIViewController` instance that you set as the page view controller's initial view controller to an instance of `AlbumPageViewController`.

Add the following import before the `RootViewController` implementation block:

```
#import "AlbumPageViewController.h"
```

And instead of making your `albumPageViewController` variable an instance of `UIViewController`, change it to the following:

```
AlbumPageViewController *albumPageViewController =
[self.storyboard
 instantiateViewControllerWithIdentifier:@"AlbumPageViewController"
];
```

Build and run the app to make sure that everything works fine, you get the same result as before but are now using the `AlbumViewController` class as the page view controller's child view controllers.

If you notice at the moment there is no interaction going on within the app, you can't swipe to flip pages or even rotate the device. Let's add that functionality now!

You are going to make the `RootViewController` both the page view controller's data source and delegate, so go ahead and add the protocol declaration within **RootViewController.h** as follows:

```
@interface RootViewController : UIViewController  
<UIPageViewControllerDelegate, UIPageViewControllerDataSource>
```

Next switch back to **RootViewController.m**, and implement the delegate and data source methods.

You'll begin with the only required delegate method, so copy this within the `rootViewController` block. You can place it anywhere you want since it's a delegate method and it will be called regardless of where it's located in the implementation file:

```
-  
(UIPageViewControllerSpineLocation)pageViewController:(UIPageViewController *)pageViewController  
spineLocationForInterfaceOrientation:(UIInterfaceOrientation)orientation  
{  
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad &&  
    UIInterfaceOrientationIsLandscape(orientation))  
    {  
        AlbumViewController *pageOne = [self.storyboard  
        instantiateViewControllerWithIdentifier:@"AlbumViewController"];  
        AlbumViewController *pageTwo = [self.storyboard  
        instantiateViewControllerWithIdentifier:@"AlbumViewController"];  
  
        NSArray *viewControllers = @[pageOne, pageTwo];  
  
        [self.pageViewController  
        setViewControllers:viewControllers  
        direction:UIPageViewControllerNavigationDirectionForward  
        animated:YES  
        completion:nil];  
    }  
}
```

```
    return UIPageViewControllerSpineLocationMid;
}

return UIPageViewControllerSpineLocationMin;
}
```

All this does is return the mid spine location if you are in landscape mode on an iPad, otherwise you return the min location. Because the iPhone's screen is much smaller than the iPad you don't want to divide the page view controller into two pages, so you will always have the spine on the left side.

If you are on the iPad and are switching to landscape orientation you will display two view controllers (two pages side by side) so you need to set the page view controller's view controllers again (because it now needs two instead of just one).

You instantiate two `AlbumPageViewController` objects, store them in an array and call the method provided to you by `UIPageViewController`, you keep the navigation direction to forward and animate the transition.

Right now if you run this nothing will change, take a look at the `shouldAutorotateToInterfaceOrientation:` method and you'll see you only support the portrait orientation at the moment. Change the method as follows:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation)interfaceOrientation
{
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone)
    {
        return interfaceOrientation !=
            UIInterfaceOrientationPortraitUpsideDown;
    }

    return YES;
}
```

This will allow the iPad to support all interface orientations and for the iPhone all but the portrait upside down orientation. Paste this method in the **AlbumPageViewController.m** file so you support these orientations within it as well.

Go ahead and run your project now, you should see it rotates properly, another yay for you!

Now for the data source methods paste the following methods inside **RootViewController.m** (before `splineLocationForInterfaceOrientation`):

```
- (UIViewController *)pageViewController:
```

```
(UIPageViewController *)pageViewController:  
viewControllerBeforeViewController:  
    (UIViewController *) viewController  
{  
    return [self.storyboard  
        instantiateViewControllerWithIdentifier:  
            @"AlbumPageViewController"];  
}  
  
- (UIViewController *)pageViewController:  
    (UIPageViewController *)pageViewController  
viewControllerAfterViewController:  
    (UIViewController *)viewController  
{  
    return [self.storyboard  
        instantiateViewControllerWithIdentifier:  
            @"AlbumPageViewController"];  
}
```

These methods just return an instance of the `AlbumPageViewController` class. This isn't the final implementation of these methods but for now you will use them to test that your page view controller is properly setup with the correct navigation and spine locations.

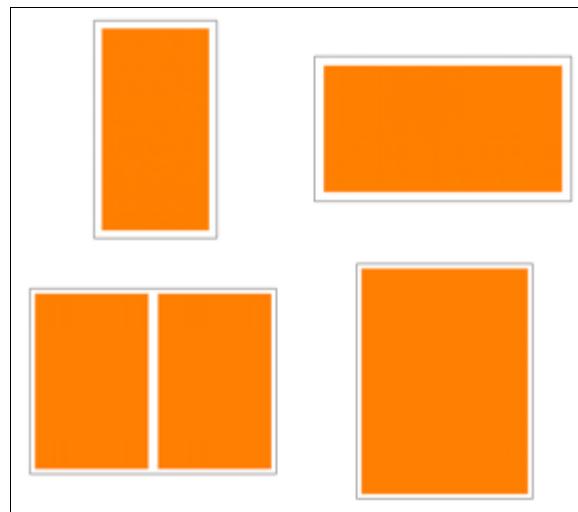
Before you go ahead and run the project you have to add the page view controller's gesture recognizers to the root view controller's gesture recognizers. Add this line inside the `viewDidLoad` method right after adding the page view controller's view as a sub-view:

```
self.view.gestureRecognizers =  
    self.pageViewController.view.gestureRecognizers;
```

Finally you need to set yourself as the delegate and data source of the page view controller. Also inside `viewDidLoad` add this line right after you create an instance of `AlbumPageViewController`:

```
self.pageViewController.delegate = self;  
self.pageViewController.dataSource = self;
```

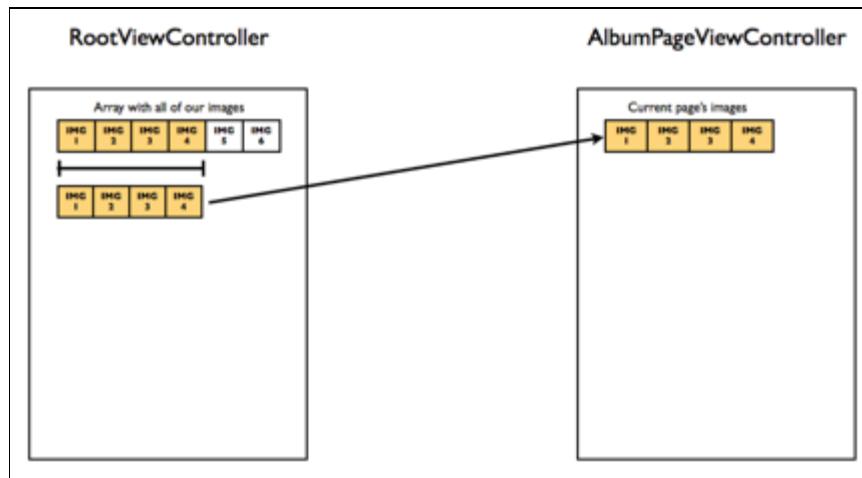
And with that build and run your app, these are the results:



Just as wanted, you have the spine located on the left when you are using the iPhone, regardless of its orientation, and on the iPad you have a spine on the left when in portrait orientation or in the middle when in landscape orientation.

Go ahead and try swiping or tapping to flip to the next or previous page. Right now you aren't using any actual logic to drive your pages so you can infinitely flip through them.

You should work on that soon, but before doing so you need an actual model to drive the information being shown on the pages and so you can set a limit to the number of pages to display.



Don't laugh, I know it's simple! Basically your root view controller will contain an array with all of the images for the application, whenever the user rotates the device or flips to another page you will grab a set of pictures and pass them to a local array in the album page view controller object.

Because of the iPad's larger screen you will show 4 pictures at a time, on the iPhone you will only show 2.

You have to create the logic to determine how many images need to be passed along and which ones to choose from.

Add an `NSArray` property inside **RootViewController.h**:

```
@property (strong, nonatomic) NSArray *picturesArray;
```

Do the exact same thing for the `AlbumPageViewController` class, inside **AlbumPageViewController.h** add the same property.

The `AlbumPageViewController` class will also have an integer to store its index in the view hierarchy, this allows you to control the hierarchy without having to write the methods that the template uses.

You are just doing things a bit different so you learn more in the process, both ways are fine and it all depends on how your app is built and structured.

In **AlbumPageViewController.h** add this property:

```
@property (nonatomic) NSUInteger index;
```

Great! You are now ready to make a few changes to the data source and delegate methods so that the number of pages presented matches the number of images available.

Go to **RootViewController.m** and update the `spineLocationForInterfaceOrientation:` delegate method to look like the following:

```
- (UIPageViewControllerSpineLocation)pageViewController:  
    (UIPageViewController *)pageViewController  
    spineLocationForInterfaceOrientation:  
        (UIInterfaceOrientation)orientation  
{  
    // 1  
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad &&  
        UIInterfaceOrientationIsLandscape(orientation))  
    {  
        // 2  
        AlbumPageViewController *currentViewController =  
            self.pageViewController.viewControllers[0];  
        NSArray *viewControllers = nil;  
  
        NSUInteger indexOfCurrentViewController =  
            currentViewController.index;  
  
        // 3  
        if (indexOfCurrentViewController == 0 ||  
            indexOfCurrentViewController % 2 == 0)
```

```
{  
    UIViewController *nextViewController = [self  
        pageViewController:self.pageViewController  
        viewControllerAfterViewController:  
            currentViewController];  
    viewControllers =  
        @[currentViewController, nextViewController];  
}  
else  
{  
    UIViewController *previousViewController = [self  
        pageViewController:self.pageViewController  
        viewControllerBeforeViewController:  
            currentViewController];  
    viewControllers =  
        @[previousViewController, currentViewController];  
}  
  
// 4  
[self.pageViewController  
    setViewControllers:viewControllers  
    direction:UIPageViewControllerNavigationDirectionForward  
        animated:YES  
        completion:NULL];  
  
return UIPageViewControllerSpineLocationMid;  
}  
  
// 5  
AlbumPageViewController *currentViewController =  
    self.pageViewController.viewControllers[0];  
  
NSArray *viewControllers = @[currentViewController];  
  
[self.pageViewController setViewControllers:viewControllers  
    direction:UIPageViewControllerNavigationDirectionForward  
        animated:YES  
        completion:NULL];  
self.pageViewController.doubleSided = NO;  
  
return UIPageViewControllerSpineLocationMin;  
}
```

Do not worry, it looks long and complicated but we are going to cover each part of the method so you can see how easy it is!

1. In the if statement you just check to see if you are on an iPad and in landscape orientation, if you are then we'll support the spine in the middle and write some code to make that happen.
2. In the previous implementation of this method you just returned a new `AlbumPageViewController`. You are updating this because you want to get the index of the currently displayed `AlbumPageViewController` and do some stuff accordingly. In order to do this you can use the `viewControllers` property of the page view controller which returns an array of the view controllers being displayed, you get the view controller at index 0 (the first one) and assign it to an `AlbumPageViewController` variable. After this you create an `NSArray` named `viewControllers` that you will use to set the view controllers of the page view controller, finally you just read the index of the album page view controller you just stored and save it into a variable of type `NSUInteger`.
3. Inside this if statement you check to see whether the index of the album page view controller you are currently displaying is 0 or an even number, If so then you use the page view controller's data source method to return the album page view controller that goes after the one currently displayed. You store both the current and next album page view controllers in the `NSArray` variable you declared before. If the statement isn't true then it means the index of the current album page view controller is an odd number, you do something very similar as before but you ask the data source to return the album page view controller before the one being currently displayed.
4. After properly configuring the view controllers you set them on the page view controller, once again you use the forward navigation with an animated transition and you return the mid spine location (which will result in two pages being displayed, like an open book).
5. If you are not on an iPad in landscape orientation then you present the page view controller's first view controller, store it in an `NSArray`, set it on the page view controller, change the `doubleSided` property to `NO` and return the min spine location.

If you are on an iPhone you will return the same view controller, if you are on an iPad (which means the user might have been using the app in landscape) you get the first view controller being displayed by the page view controller.

Note: When returning the mid spine location, the page view controller's `doubleSided` property is automatically set to `YES`. If you do not want this to happen then set it to `NO` when moving back to a min or max spine location.

Let's now look at the changes in the data source methods (they are much simpler luckily)!

Replace your `pageViewController:viewControllerBeforeViewController` method with the following:

```
- (UIViewController *)pageViewController:  
    (UIPageViewController *)pageViewController  
viewControllerBeforeViewController:  
    (UIViewController *)viewController  
{  
    AlbumPageViewController *previousViewController =  
        (AlbumPageViewController *)viewController;  
  
    if (previousViewController.index == 0)  
    {  
        return nil;  
    }  
  
    AlbumPageViewController *albumPageViewController =  
        [self.storyboard instantiateViewControllerWithIdentifier:  
            @"AlbumPageViewController"];  
    albumPageViewController.index =  
        previousViewController.index - 1;  
  
    return albumPageViewController;  
}
```

This method gets called when the user navigates backwards in the album so the first thing you do is cast and store the view controller being passed into the method (the view controller being currently displayed).

You have a simple if statement to check whether this is the first view controller, if it is you return nil (because there are no pages that can return before the first one) otherwise you create and return a new `AlbumPageViewController` instance (using the storyboard's method) and decrease its index by one (the page number before the one currently displayed).

Now let's take a look at the other data source method left:

```
- (UIViewController *)pageViewController:  
    (UIPageViewController *)pageViewController  
viewControllerAfterViewController:  
    (UIViewController *)viewController  
{  
    AlbumPageViewController *previousViewController =  
        (AlbumPageViewController *)viewController;  
  
    if (previousViewController.index == 9)  
    {  
        return nil;  
    }
```

```
AlbumPageViewController *albumPageViewController =  
[self.storyboard instantiateViewControllerWithIdentifier:  
 @"AlbumPageViewController"];  
albumPageViewController.index =  
 previousViewController.index + 1;  
  
return albumPageViewController;  
}
```

The code is exactly the same except instead of checking to see if you are on the first page, you now check to see if you are on the last one. For testing purposes you are going to use the value 9 inside the if statement which will give us 10 pages for the album, when you setup the pictures we'll calculate this at runtime (depending on the amount of pictures you have).

If you're on the last page you return nil (because there are no pages after the last one) otherwise you instantiate and return a new `AlbumPageViewController` with the index that follows the one currently displayed.

That's almost all you need to do to limit the number of pages, the last thing you need to do is set the index of the album page view controller when the app launches. In the `viewDidLoad` method add this line right after you instantiate an `AlbumPageViewController`:

```
albumPageViewController.index = 0;
```

Build and run the app and you should now have only 10 pages to navigate between, and on the iPad you even get two pages in landscape orientation.

Note: The current implementation requires that we have an even number of pages to display (2, 4, 6, 8, 10...), if you have an odd number of pages then the last one will not show when using an iPad in landscape mode. So if you have 9 pages then you will only show pages 7 and 8 but never 9 (you need pass two view controllers to the page view controller). Keep this in mind if you use this code in your project, otherwise write something to suit your needs.

How cool is the page view controller? You have written very little code and achieved something that would have required serious customization in previous versions of iOS.

The photo album is ready to show some photos, which means you need to add them to the project. You are going to store the photos in the app bundle and load an array with the names from a property list file.

Because the purpose of this chapter is to cover the `UIPageViewController` then this implementation will be fine. For your own projects you could read pictures from Facebook, Twitter, an RSS feed or just about anything you can imagine!

The resources for this chapter include the property list file and all of the pictures you're going to use for your album, so go ahead and locate those files and add them to your project.

The property list file contains a dictionary as the root object and within it you have an array of strings with the names of our photos. You are going to load this array from the property list file and save it into the root view controller's `picturesArray`.

In order to properly size and scale the images you are going to determine the width and height at runtime and position them on the view controller accordingly.

Let's load up the array! Open **RootViewController.m** and put this code inside the `viewDidLoad` method, preferably before the `pageViewOptions` declaration:

```
NSDictionary *picturesDictionary = [NSDictionary
    dictionaryWithContentsOfFile:[[NSBundle mainBundle]
    pathForResource:@"Photos"
    ofType:@"plist"]];
self.picturesArray = picturesDictionary[@"PhotosArray"];
```

Now you have the array loaded with the picture names, great, but you need to pass them into the album page view controller so they can be shown on each page. In order to do this you have to make a few changes to the page view controller's data source methods as well as the `viewDidLoad` method.

Let's get started with the page view controller data source methods, in **RootViewController.m** replace the `viewControllerBeforeViewController:` method with the following code:

```
- (UIViewController *)pageViewController:
    (UIPageViewController *)pageViewController
viewControllerBeforeViewController:
    (UIViewController *)viewController
{
    AlbumPageViewController *previousViewController =
    (AlbumPageViewController *)viewController;

    if (previousViewController.index == 0)
    {
        return nil;
    }

    AlbumPageViewController *albumPageViewController =
    [self.storyboard instantiateViewControllerWithIdentifier:
```

```
 @"AlbumPageViewController"];
albumPageViewController.index =
    previousViewController.index - 1;

NSRange picturesRange;

if ([[UIDevice currentDevice] userInterfaceIdiom] ==
    UIUserInterfaceIdiomPad)
{
    NSUInteger startingIndex =
        ((albumPageViewController.index) * 4);

    picturesRange.location = startingIndex;
    picturesRange.length = 4;

    if ((picturesRange.location + picturesRange.length) >=
        self.picturesArray.count)
    {
        picturesRange.length = self.picturesArray.count -
            picturesRange.location - 1;
    }
}
else
{
    NSUInteger startingIndex =
        ((albumPageViewController.index) * 2);

    picturesRange.location = startingIndex;
    picturesRange.length = 2;

    if ((picturesRange.location + picturesRange.length) >=
        self.picturesArray.count)
    {
        picturesRange.length = self.picturesArray.count -
            picturesRange.location - 1;
    }
}

albumPageViewController.picturesArray =
    [self.picturesArray subarrayWithRange:picturesRange];

return albumPageViewController;
}
```

Fear not, I will explain what this code is doing. I will not go into great detail as to how the pictures algorithm works because you are covering the

`UIPageViewController`. Nevertheless, I'll give a brief overview so you can see how it's all connected together.

Let's get started.

As usual you begin by getting the previous view controller and checking if it's the first page (index 0), if it is then you don't pass a view controller before this one so you return nil. If you can show a page before the one we came from then you instantiate a new album page view controller object and decrease its index from the current one.

Simple stuff, right? Now you get into the picture calculations, for this you declare an `NSRange` to determine which picture names you are going to get from the array and pass them into the album page view controller.

You have an if statement depending on whether you are on iPhone or iPad, on iPad you will display 4 pictures per page but on iPhone you will only show 2 because of the limited screen size.

Whether you are on iPhone or on iPad you use the album page view controller's index to determine at what index you will begin looking for the names of the pictures, once you do that you get either 4 strings for iPad or 2 for iPhone and pass them into the new album page view controller object.

Inside here you do make sure you don't get any invalid indexes in the array or try to access objects that don't exist. Go over the code and it will make sense to you, very simple!

Next replace `viewControllerAfterViewController:` method with the following:

```
- (UIViewController *)pageViewController:  
    (UIPageViewController *)pageViewController  
    viewControllerAfterViewController:(UIViewController *)  
    viewController  
{  
    AlbumPageViewController *previousViewController =  
    (AlbumPageViewController *)viewController;  
  
    NSUInteger pageCount;  
  
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==  
        UIUserInterfaceIdiomPad)  
    {  
        pageCount = (NSUInteger)  
            ceilf(self.picturesArray.count * 0.25);  
    }  
    else  
    {  
        pageCount = (NSUInteger)
```

```
        ceilf(self.picturesArray.count * 0.5);
    }

    pagesCount--;

    if (previousViewController.index == pagesCount)
    {
        return nil;
    }

    AlbumPageViewController *albumPageViewController =
    [self.storyboard instantiateViewControllerWithIdentifier:
     @"AlbumPageViewController"];
    albumPageViewController.index =
    previousViewController.index + 1;

    NSRange picturesRange;

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
         UIUserInterfaceIdiomPad)
    {
        NSUInteger startingIndex =
        ((albumPageViewController.index) * 4);

        picturesRange.location = startingIndex;
        picturesRange.length = 4;

        if ((picturesRange.location + picturesRange.length) >=
              self.picturesArray.count)
        {
            picturesRange.length = self.picturesArray.count -
            picturesRange.location - 1;
        }
    }
    else
    {
        NSUInteger startingIndex =
        ((albumPageViewController.index) * 2);
        picturesRange.location = startingIndex;
        picturesRange.length = 2;

        if ((picturesRange.location + picturesRange.length) >=
              self.picturesArray.count)
        {
            picturesRange.length = self.picturesArray.count -
```

```
        picturesRange.location = 1;
    }
}

albumPageViewController.picturesArray = [self.picturesArray
    subarrayWithRange:picturesRange];

return albumPageViewController;
}
```

The code is very similar to the previous method except you check to see if you are on the last page already. Then you proceed to use the index of the new page view controller to determine which range of picture names you need to get from our array.

Before you leave the `RootViewController` you need to add some code to the `viewDidLoad` method. Both page view controller data source methods are called when you flip to a new page, but when the application first launches you need to pass in some pictures to the first page that will be displayed.

Add this code after the declaration of the `albumPageViewController` and before you set its data source and delegates:

```
if (self.picturesArray.count > 0)
{
    NSRange picturesRange;

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad)
    {
        picturesRange.location = 0;

        if (self.picturesArray.count >= 4)
        {
            picturesRange.length = 4;
        }
        else
        {
            picturesRange.length = self.picturesArray.count;
        }
    }
    else
    {
        picturesRange.location = 0;
    }
}
```

This code just finds either 2 pictures for iPhone or 4 for iPad, passes them into the new album page view controller and continues as before.

Awesome, you are getting the `NSStrings` with the names of the pictures properly passed into the album page view controllers, now you need to go to the album page view controller and properly position the pictures on the page!

Before you do that go back to the `Storyboard_iPhone` and `Storyboard_iPad` files and delete the colored `UIViews` you placed on your album page view controller. I also changed the view's background color to black.

Feel free to customize the look of your app as much as you want, perhaps add an old looking page, or colored backgrounds, just have fun with this.

Showing the pictures on screen

Now that you have the storyboards clean and customized, let's take a look at the final portion of code you must implement in order for the pictures to be placed on screen. Again I will not go into much detail with these algorithms because they aren't part of the page view controller API, nevertheless you will see what's going on so you can study things on your own.

Open up `AlbumPageViewController.m` and add a private property declaration:

```
@interface AlbumPageViewController()  
  
@property (nonatomic, strong) NSMutableArray *pictureViews;  
  
@end
```

You create an `NSMutableArray` property named `pictureViews` that will store the `UIImageView` objects, this will save you from having to reload images and re create any objects just because the interface rotation changed.

Add this private method to the class:

```
- (void)layoutPicturesAnimated:(BOOL)animated  
withDuration:(NSTimeInterval)duration  
forInterfaceOrientation:  
(UIInterfaceOrientation)interfaceOrientation  
{  
    if (animated)  
    {  
        [UIView beginAnimations:nil context:nil];  
        [UIView setAnimationDuration:duration];  
        [UIView setAnimationDelay:0];  
        [UIView setAnimationCurve:UIViewAnimationCurveEaseOut];
```

```
    }

    CGRect orientationFrame;

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad)
    {
        if (UIInterfaceOrientationIsPortrait(
            interfaceOrientation))
        {
            orientationFrame = CGRectMake(0, 0, 768, 1024);
        }
        else
        {
            orientationFrame = CGRectMake(0, 0, 512, 768);
        }
    }
    else
    {
        if (UIInterfaceOrientationIsPortrait(
            interfaceOrientation))
        {
            orientationFrame = CGRectMake(0, 0, 320, 480);
        }
        else
        {
            orientationFrame = CGRectMake(0, 0, 480, 320);
        }
    }

    if (self.picturesArray.count > 0)
    {
        if (self.picturesArray.count == 1)
        {
            [self setPictureAtIndex:0 inFrame:orientationFrame];
        }
        else if (self.picturesArray.count == 2)
        {
            CGRect frameOne;
            CGRect frameTwo;

            if ([[UIDevice currentDevice] userInterfaceIdiom] ==
                UIUserInterfaceIdiomPhone &&
                UIInterfaceOrientationIsLandscape(interfaceOrientation)))
            {

```

```
        frameOne = CGRectMake(0, 0,
orientationFrame.size.width * 0.5, orientationFrame.size.height);
        frameTwo = CGRectMake(orientationFrame.size.width
* 0.5, 0, orientationFrame.size.width * 0.5,
orientationFrame.size.height);

        [self setPictureAtIndex:0 inFrame:frameOne];
        [self setPictureAtIndex:1 inFrame:frameTwo];
    }
else
{
    frameOne = CGRectMake(0, 0,
orientationFrame.size.width, orientationFrame.size.height * 0.5);
    frameTwo = CGRectMake(0,
orientationFrame.size.height * 0.5, orientationFrame.size.width,
orientationFrame.size.height * 0.5);

    [self setPictureAtIndex:0 inFrame:frameOne];
    [self setPictureAtIndex:1 inFrame:frameTwo];
}
}
else
{
    CGRect frameOne;
    CGRect frameTwo;
    CGRect frameThree;
    CGRect frameFour;

    frameOne = CGRectMake(0, 0,
orientationFrame.size.width * 0.5, orientationFrame.size.height *
0.5);
    frameTwo = CGRectMake(orientationFrame.size.width *
0.5, 0, orientationFrame.size.width * 0.5,
orientationFrame.size.height * 0.5);
    frameThree = CGRectMake(0,
orientationFrame.size.height * 0.5, orientationFrame.size.width *
0.5, orientationFrame.size.height * 0.5);
    frameFour = CGRectMake(orientationFrame.size.width *
0.5, orientationFrame.size.height * 0.5,
orientationFrame.size.width * 0.5, orientationFrame.size.height *
0.5);

    [self setPictureAtIndex:0 inFrame:frameOne];
    [self setPictureAtIndex:1 inFrame:frameTwo];
    [self setPictureAtIndex:2 inFrame:frameThree];
```

```
        if (self.picturesArray.count == 4)
    {
        [self setPictureAtIndex:3 inFrame:frameFour];
    }
}

if (animated)
{
    [UIView commitAnimations];
}
}
```

This method receives a Boolean to determine whether you want the picture positioning to be animated or not, the duration of the animation and the frame in which to position the pictures.

At the beginning and end of the method you will see that you check whether you wanted the transition to be animated and if so, you do some simple `UIview` animations for the resizing and position of the pictures using the duration passed as a parameter.

Using the orientation passed into the method you determine the frame depending on the device you are on and use that to setup the pictures, then you check whether there are any pictures to show or not as you don't want to be accessing empty arrays or invalid indexes.

If you have just one picture you call on another private method (that you will see in a minute) that handles setting up the pictures, you don't have to divide the screen or anything for one picture, just scale and center it.

When there are two pictures you check whether the device is an iPhone in landscape or either an iPhone or iPad in portrait so you can divide the view horizontally or vertically respectively. Once that's done you call the `setPictureAtIndex:` private method to handle laying out the pictures at the given frames.

The pages on an iPad can have 3 or 4 pictures as well, in this case we divide the screen into four equal rectangles and pass their frames into your `setPictureAtIndex:` method to handle positioning them.

Nothing too complicated, just some simple math to determine how many pictures you have, what orientation and device you are using the app on and where to position the pictures.

Now paste this method right above the `layoutPicturesAnimated:` method:

```
- (void)setPictureAtIndex:(NSUInteger)index
inFrame:(CGRect)frameForPicture
{
    UIImageView *picture = self.pictureViews[index];

    CGFloat scale;

    if (picture.image.size.width > picture.image.size.height)
    {
        scale = picture.image.size.height /
picture.image.size.width;

        picture.frame = CGRectMake(0, 0,
(frameForPicture.size.width * 0.80), (frameForPicture.size.width *
0.80) * scale);
        picture.center = CGPointMake(frameForPicture.origin.x +
(frameForPicture.size.width * 0.5), frameForPicture.origin.y +
(frameForPicture.size.height * 0.5));
    }
    else
    {
        scale = picture.image.size.width /
picture.image.size.height;

        picture.frame = CGRectMake(0, 0,
(frameForPicture.size.height * 0.80) * scale,
(frameForPicture.size.height * 0.80));
        picture.center = CGPointMake(frameForPicture.origin.x +
(frameForPicture.size.width * 0.5), frameForPicture.origin.y +
(frameForPicture.size.height * 0.5));
    }

    [self.view addSubview:picture];
}
```

This method receives the picture index and a `CGRect` that contains the frame where the picture is to be positioned. You first get a `UIImageView` with the corresponding image and create a `CGFloat` variable to determine the ratio between the width and height of the picture.

Some calculations are done depending on whether the image width is larger than its height or vice versa. After that's determined you get the scale and proceed to make the image 20% smaller than the frame that contains it.

Finally the image gets centered and added as a sub-view of the album page view controller's view.

You just need to customize three more `UIViewController` methods in order to get things working, let's start with `viewDidLoad`:

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

    for (NSString *pictureName in self.picturesArray)
    {
        if (!self.pictureViews)
        {
            self.pictureViews = [NSMutableArray array];
        }

        UIImageView *picture = [[UIImageView alloc]
initWithImage:[UIImage imageNamed:pictureName]];
        [self.pictureViews addObject:picture];
    }
}
```

You create a `UIImageView` for every name of a picture you have in the `picturesArray`, you make sure to initialize the `pictureViews` array in case you haven't already. And finally these two methods:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    [self layoutPicturesAnimated:NO withDuration:0
forInterfaceOrientation:self.interfaceOrientation];
}

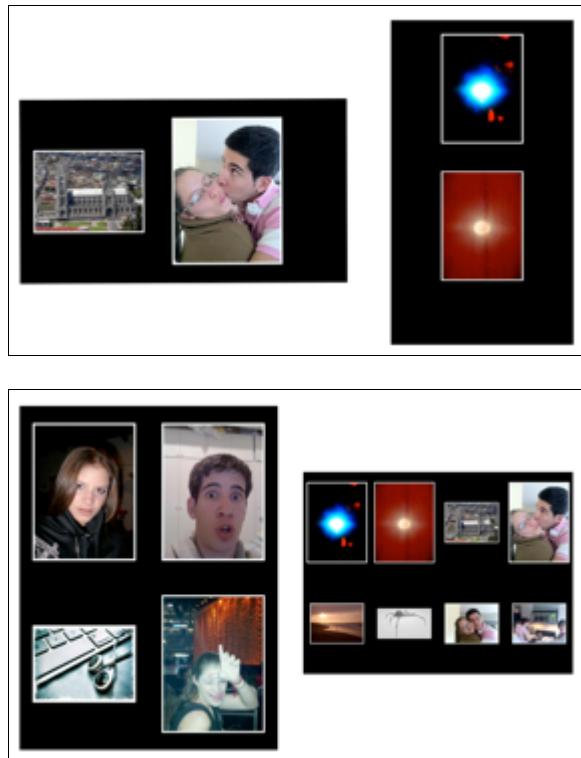
-
(void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation duration:(NSTimeInterval)duration
{
    [self layoutPicturesAnimated:YES withDuration:duration
forInterfaceOrientation:toInterfaceOrientation];
}
```

When your view first appears you want to position the pictures immediately, so you pass `NO` into the `layoutPicturesAnimated:` method. You also pass the current interface orientation so you construct the frame for the pictures to be placed in as well as a duration, which in your case is `0` because you want things to load immediately when you first load.

On the `willRotateToInterfaceOrientation:` method you layout the pictures again except this time you animate the transition, you use the new orientation you are going to rotate to and the same duration as the transition.

And with that... YOU ARE DONE!

A big yay for us as you have covered a lot of ground on this chapter. Build and run your application and this is what the results look like:



Where To Go From Here?

There are many things you can do with the `UIPageViewController`, perhaps load pictures or articles from RSS Feeds. How about a Facebook or Twitter client that uses this elegant interface?

You could also improve the pictures algorithm and put your own images in the property list file. Even better you can load images from the user's image library.

The possibilities with this new type of container controller, `UIPageViewController`, are endless and I cannot wait to see what you guys and girls come up with!

Chapter 18: Beginning Turn Based Gaming

By Jacob Gundersen

Apple's Game Center has been an amazing boon for game developers. It has made adding achievements, leaderboards, multiplayer, and live chat into your game much easier. It also makes your games more social, adding additional exposure for your games.

In iOS 5, Game Center added a new API that makes it even easier to create another type of game – turn-based games! This new API is perfect for board games, turn based strategy, word games, and casual games. You can take a turn, wait for a friend to take his turn, and then get a notification when it's your turn again!

The new API is a good choice, even over existing third party or home grown solutions, because it takes advantage of the social networking elements of Game Center. Players can send invitations from the developer's games to others, even friends that haven't yet installed the game, and the opportunity will be given to download that game.

Because of the large existing base of game center users, the leaderboards and achievements, this new API is very attractive.

In this tutorial you will build a simple UIKit based game called "Spinnig Yarn". In this game you will take turns writing a story with up to four friends. You'll build the scaffolding for the game first, then you'll spend the bulk of the tutorial exploring the new Turn Based Gaming APIs.

Turn based gaming overview

In a turn based game, only one player can affect the game state at a time. This player will hold the baton, or the game state. They will take a turn, which will change the game state, then pass the turn on with the new game state.

However, when you're playing a turn-based game you don't have to sit there twiddling your thumbs while you're waiting for the other player to take their turn. You can play multiple games at once, take a look at the state of your various

games, or switch out of the app and do something else while you wait. So in your game, you'll give the user the ability to switch between the matches they're currently in.

Here are the major classes in the Turn Based Gaming API, which you'll be covering in detail in this tutorial:

- **GKTurnBasedMatch**: Contains all the information and game state for a match. You'll use this object to encapsulate the match players, the match game state, and the important information about who holds the turn and who's still playing. This object is passed around as the 'baton' each turn.
- **GKTurnBasedMatchmakerViewController**: This will be your primary UI element for interacting with matches, switching between matches, and creating new matches. You'll use it as your command center.
- **GKTurnBasedEventHandlerDelegate**: This delegate object receives callbacks when important events happen. Those events include when the turn moves from player to player, when the player is invited to a new match, or when the game ends. You will make a helper class implement this protocol.

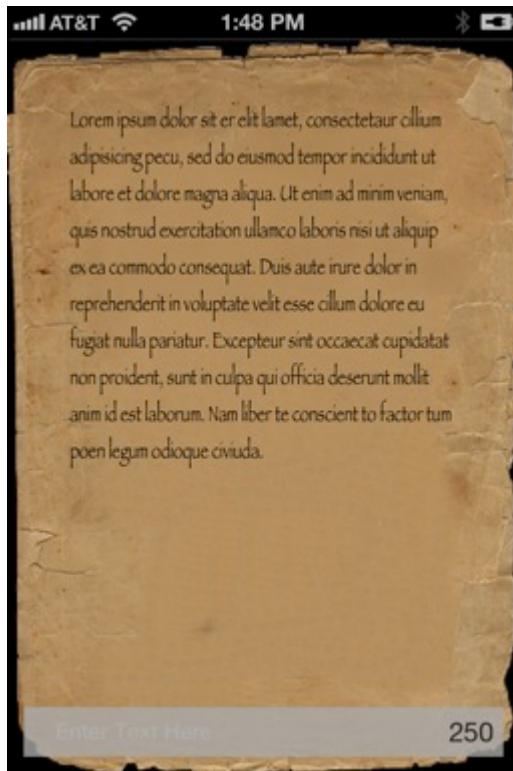
Note: The Game Center app will automatically keep track of who the user is playing with and other data about that game, so users can always switch to Game Center to see their game status.

Introducing Spinning Yarn

If you harken back to your days in grade school, you may remember a writing exercise where each child started writing a story. After a certain period, you passed your paper to your neighbor and he gave you his. You read what he wrote and continued the story. This continued back and forth until you ended up with a story, often with hilarious results!

This is the turn-based game you'll be making in this tutorial. I've made a starter project that contains the UI already made, so you can keep the focus mainly on the Turn Based Gaming APIs.

You'll find this in the resources for this tutorial, so go ahead and open it in Xcode and run it. You should see the following UI:



Right now this project just contains a text view displaying some sample text, and a text field at the bottom of the screen that you can enter text into. Note it doesn't actually do anything yet - it's just placeholder code for now. The app also has a pretty paper background (thanks to playingwithbrushes on Flickr!)

You can poke around the Storyboard file as well as the **ViewController.m** file to see what code is already included. There's some handling of the text field input, as well as code that animates the text field so that it stays on top of the keyboard. That's about it so far.

Setting up Game Center

Before you can start writing any Game Center code, you need to do two things:

1. Create and set up an App ID
2. Register your app in iTunes Connect

I'll go through each of these in turn.

Create and set an App ID

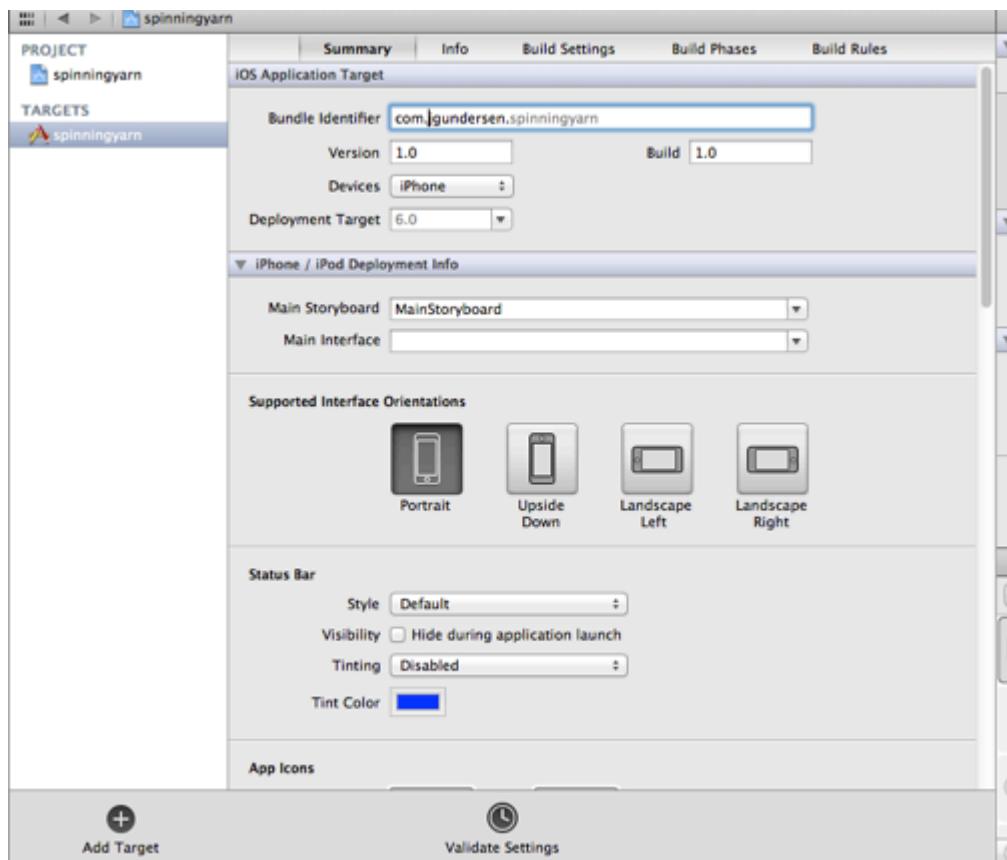
The first step is to create and set an App ID. To do this, log onto the iOS Dev Center, and from there log onto the iOS Provisioning Portal.

From there, select the App IDs tab, and create a new App ID for your app, similar to the following (except you'll be choosing different values):

The screenshot shows the 'Provisioning Portal : Third Rail, LLC' interface. The left sidebar has a 'App IDs' tab selected. The main area is titled 'Create App ID' and contains fields for 'Description' (containing 'spinningyarn'), 'Bundle Seed ID (App ID Prefix)' (set to 'Use Team ID'), and 'Bundle Identifier (App ID Suffix)' (containing 'com.~~qundered~~ spinningyarn'). A note at the bottom says 'Example: com.domainname.apponame'.

The most important part is the Bundle Identifier – you need to set this to a unique string (so it can't be the same as the one I used!) It's usually good practice to use a domain name you control followed by a unique string to avoid name collisions.

Once you're done, click Submit. Then open the **SpinningYarn** Xcode project, select the **spinningyarn** target, and in the **Summary** tab set your **Bundle Identifier** to whatever you entered in the iOS Provisioning portal, as shown below (except you'll be entering a different value):



One last thing. Xcode sometimes gets confused when you change your bundle identifier mid-project, so to make sure everything's dandy take the following steps:

- Delete any copies of the app currently on your simulator or device
- Quit your simulator if it's running
- Do a clean build with Project\Clean

Congrats – now you have an App ID for your app, and your app is set up to use it! Next you can register your app with iTunes Connect and enable Game Center.

Register your app in iTunes Connect

The next step is to log on to iTunes Connect and create a new entry for your app.

Once you're logged onto iTunes Connect, select Manage Your Applications, and then click the blue '**Add New App**' button in the upper left. Choose iOS App if prompted.

On the first screen, enter something like Spinning Yarn for the App Name (put your initials or something afterwards though, because I've already taken that name), some random number for SKU Number, and select the bundle ID you created earlier, similar to the screenshot below:

Enter the following information about your app.

Default Language	English
App Name	Spinning Yarn RJW
SKU Number	1234-RJW
Bundle ID	spinningyarn - com.razeware.spinningyarn

You can register a new Bundle ID [here](#).

Click Continue, and follow the prompts to set up some basic information about your app. Don't worry about the exact values to put in, since it doesn't really matter and you can change any of this later – you just need to put something (including a dummy icon and screenshot) in to make iTunes Connect happy.

When you're done, click Save. If all works well you should be in the '**Prepare for Upload**' stage and will see a screen like this:

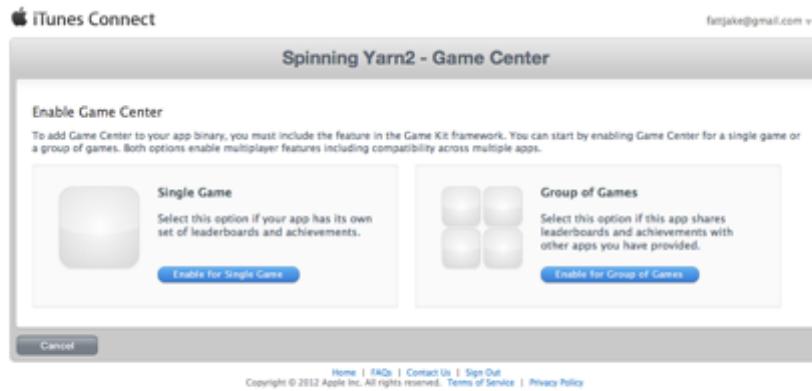
App Information [Edit](#)

Identifiers SKU 222 Bundle ID com.jgundersen.spinningyarn Apple ID 568958070 Type iOS App Default Language English	Links View in App Store	Rights and Pricing Manage In-App Purchases Manage Game Center Set Up iAd Network Newsstand Delete App
--	---	--

Versions

Current Version  Version 1.0 Status ● Prepare for Upload Date Created Oct 8, 2012 View Details	
---	--

Click the blue '**Manage Game Center**' button to the upper right, click '**Enable for Single Game**', and click '**Done**'.



One last, necessary step.

This is new. Click '**View Details**' beneath the game's icon in the '**Current Version**' section. In that view there's a second button that you must press in order to enable game center for that version of the app. Redundant, redundant, I know, I know.

Reference Name	Leaderboard ID	Type
0 of 0 Selected to submit (25 of 25 Leaderboards Remaining)		

That's it – Game Center is enabled for your app, and you're ready to write some code!

You may be wondering what the other option, '**Enable for Group of Games**' is for. This allows you to share Leaderboards and/or Achievements across different games. This is a new feature in iOS 6, and this author is very excited about it!

By the way – inside the "Manage Game Center" section, you might have noticed some options to set up Leaderboards or Achievements. You won't be using those in this book, but if you ever need them that's where they are!

Note: If you want to learn more about leaderboards, achievements, and challenges, check out Chapters 13 and 14 in *iOS 6 by Tutorials*, "Beginning and Intermediate Challenges with GameKit".

Authenticating the local player

When your game starts up, the first thing you need to do is authenticate the local player.

You can think of this as "logging the player into Game Center." If he's already logged in, it will say "Welcome back!"

In iOS 5 if the player wasn't logged in, a Game Center login view would automatically be presented and give the player the chance to log in using their username and password. Things have changed slightly in iOS 6.

Now, the developer is given control of when the Game Center login view is presented. This is done with a new API. In this app you'll be implementing both and using the one appropriate for the current OS your user is on.

I'll show you how you used to do it in iOS 5 first, then you'll add the iOS 6 code.

The issue is that the state of the user authentication may change while the user's not in your app. He can be using your app, switch to the Game Center app, log in or out from there, and switch back to your app.

So your app needs to know whenever the authentication status changes.

In iOS 5 you can find out about these by registering for an "authentication changed" notification. In iOS 6 the authentication handler is a block that's called every time your app enters the foreground.

So, your strategy to authenticate the player will be as follows:

1. Create a singleton object to keep all the Game Center code in one spot.
2. When the singleton object starts up, it will register for the "authentication changed" notification.
3. The game will call a method on the singleton object to authenticate the user.
4. Whenever the user is authenticated (or logs out), the "authentication changed" callback will be called.

The callback will keep track of whether the user is currently authenticated, for use later. Now that you're armed with this plan, try it out!

Coding the implementation

In the SpinningYarn Xcode project, create a new file with the **Objective-C class template**. Name the class **GCTurnBasedMatchHelper** and make it a **subclass of NSObject**.

Then replace **GCTurnBasedMatchHelper.h** with the following:

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

@interface GCTurnBasedMatchHelper : NSObject {
    BOOL userAuthenticated;
}

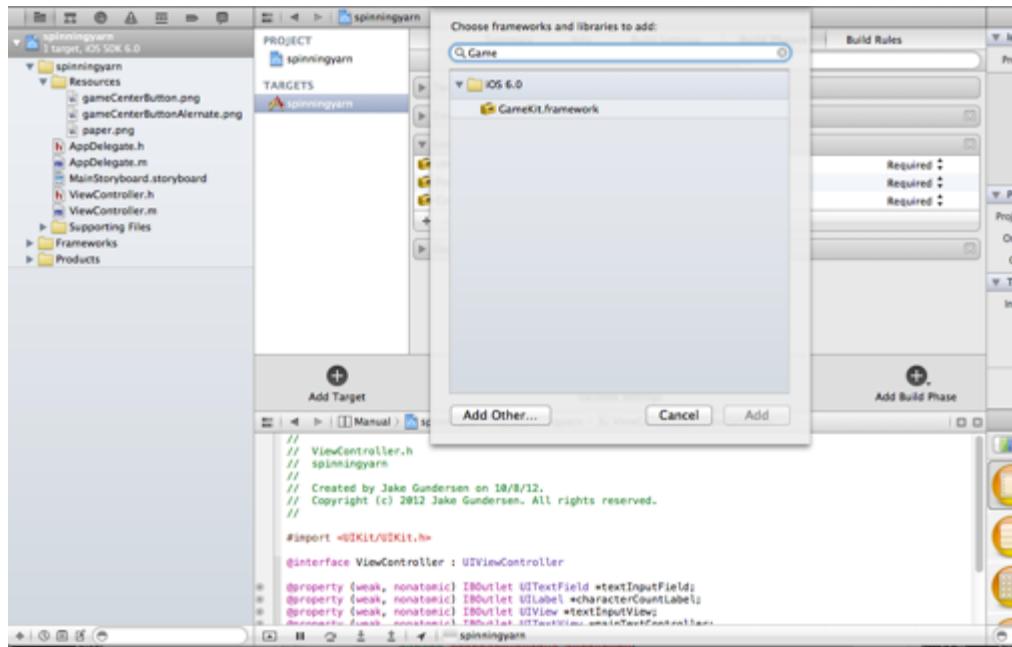
@property (assign, readonly) BOOL gameCenterAvailable;

+ (GCTurnBasedMatchHelper *)sharedInstance;
-(void)authenticateLocalUserFromViewController:
    (UIViewController *)authenticationPresentingViewController;

@end
```

This imports the **GameKit** header file, and then creates an object with two booleans – one to keep track of if game center is available on this device, and one to keep track of whether the user is currently authenticated.

You need to add the **GameKit** framework. Select the project container at the top left of the file navigator. Select the **Build Phases** tab. Expand the **Link Binaries with Libraries** group. Click the **+** button and select the **GameKit.Framework**.



Back to the code. It also creates a property so the game can tell if game center is available, a static method to retrieve the singleton instance of this class, and another method to authenticate the local user (which will be called when the app starts up).

Next, switch to **GCTurnBasedMatchHelper.m** and add the following right inside the `@implementation`:

```
#pragma mark Initialization

+ (GCTurnBasedMatchHelper *) sharedInstance {
    static GCTurnBasedMatchHelper *sharedHelper;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedHelper = [[GCTurnBasedMatchHelper alloc] init];
    });

    return sharedHelper;
}
```

This uses the most up to date method to create a singleton instance of this class.

Next add the following method right after the `sharedInstance` method:

```
- (BOOL)isGameCenterAvailable {
    // check for presence of GKLocalPlayer API
    Class gcClass = (NSClassFromString(@"GKLocalPlayer"));

    // check if the device is running iOS 4.1 or later
```

```
NSString *reqSysVer = @"4.1";

NSString *currSysVer = [[UIDevice currentDevice] systemVersion];
BOOL osVersionSupported = ([currSysVer compare:reqSysVer options:NSNumericSearch] != NSOrderedAscending);
return (gcClass && osVersionSupported);
}
```

This method is straight from Apple's Game Kit Programming Guide. It's the way to check if Game Kit is available on the current device. By making sure Game Kit is available before using it, this app can still run on iOS 4.0 or earlier (just without network capabilities).

Note: The turn-based gaming APIs you'll use later on aren't available until iOS 5.0. So if you want to support older devices, you should check if those APIs are available before using them as well (but I haven't included that in that check in this chapter for brevity).

Next add the following right after the `isGameCenterAvailable` method:

```
-(id)init {
    if ((self = [super init])) {
        _gameCenterAvailable = [self isGameCenterAvailable];
        if (_gameCenterAvailable) {
#ifndef __IPHONE_6_0 || __IPHONE_OS_VERSION_MIN_REQUIRED <
__IPHONE_6_0
            [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(authenticationChanged)
name:GKPlayerAuthenticationDidChangeNotificationName object:nil];
#endif
        } else {
            NSLog(@"Game Center is not available, this game
requires game center");
        }
    }
    return self;
}

-(void)authenticationChanged {
    if ([GKLocalPlayer localPlayer].isAuthenticated &&
!userAuthenticated) {
        userAuthenticated = YES;
```

```
        NSLog(@"Authentication changed: player authenticated.");
    } else if (![GKLocalPlayer localPlayer].isAuthenticated &&
userAuthenticated) {
    userAuthenticated = NO;
    NSLog(@"Authentication changed: player not
authenticated");
}
}
```

The `init` method checks to see if Game Center is available. If it is, it then executes the block of code that registers for the “authentication changed” notification, if you’re not using iOS 6. It’s important that the app registers for this notification before attempting to authenticate the user, so that it’s called when the authentication completes.

The notification isn’t necessary on iOS 6 because the authentication handler is called every time the user brings the app to the foreground. You can handle any issues there, instead of the function called by the notification (they basically do the same thing).

The `authenticationChanged` callback is very simple at this point – it checks to see whether the change was due to the user being authenticate or deauthenticated, and updates a status flag accordingly.

Note that in practice this might be called several times in a row for authentication or deauthentication, so by making sure the `userAuthenticated` flag is different than the current status, it only logs if there’s a change since last time.

Finally, add the method to authenticate the local user right after the `authenticationChanged` method:

```
- (void)authenticateLocalUser {
    if (![_gameCenterAvailable]) return;

    NSLog(@"Authenticating local user . . .");
    presentingViewController =
        authenticationPresentingViewController;

    if ([GKLocalPlayer localPlayer].authenticated == NO ) {
        [[GKLocalPlayer localPlayer]
authenticateWithCompletionHandler:nil];
    } else {
        NSLog(@"Already authenticated");
    }
}
```

This calls the `authenticateWithCompletionHandler` method mentioned earlier to tell Game Kit to authenticate the user. Note it doesn’t pass in a completion handler.

Since you've already registered for the "authentication changed" notification it's not necessary.

As of iOS 6, there's a new way to authenticate that replaces both the call to `authenticateWithCompletionHandler`: and to some degree the need to use the `GKPlayerAuthenticationDidChangeNotificationName` notification. In fact, if you might see a warning in your current project that says `authenticateWithCompletionHandler:` is deprecated in iOS 6.

The new way of authenticating is to set the new `.authenticationHandler` property on the `GKLocalPlayer` object. This property is a block that looks like this:

```
^(UIViewController *viewController, NSError *error) {  
    //Code goes here  
};
```

Once this property is set, it's executed and is executed again every time the app returns to the foreground.

Replace `authenticateLocalUser` with a new modified method that will use the iOS 6 authentication if it's available:

```
- (void)authenticateLocalUserFromViewController:  
    (UIViewController *)authenticationPresentingViewController {  
    if (!gameCenterAvailable) return;  
  
    NSLog(@"Authenticating local user . . .");  
    presentingViewController =  
        authenticationPresentingViewController;  
  
    GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];  
  
#if !defined(__IPHONE_6_0) || __IPHONE_OS_VERSION_MIN_REQUIRED <  
__IPHONE_6_0  
    localPlayer.authenticateHandler =  
        ^(UIViewController *viewController, NSError *error)  
    {  
        if (viewController) {  
            [authenticationPresentingViewController  
                presentViewController:viewController  
                animated:YES completion:^{  
                    userAuthenticated = YES;  
  
                }];  
        } else if (localPlayer.authenticated) {  
            userAuthenticated = YES;  
        }  
    };  
}
```

```
    } else {
        userAuthenticated = NO;
        NSLog(@"Error with Game Center %@", error);
    }
};

#else
    if (localPlayer.authenticated == NO) {
        [[GKLocalPlayer localPlayer]
authenticateWithCompletionHandler:nil];
    } else {
        NSLog(@"Already authenticated");
    }
#endif
}
```

In this case you are going to need to have a reference to the current `viewController` because if necessary you'll be presenting the Game Center login controller. You will be passing that in when you call this method.

After making sure Game Center is available it will call one of two blocks of code.

If it's iOS 6, it will call the block set to the `authenticateHandler` variable. This block passes in a `viewController` in the event that no `GKPlayer` is currently logged into Game Center. This allows you, as the developer, to control when and how the login view is presented to your user instead of it happening immediately after requesting authentication.

In this case you present it immediately, by calling `presentViewController:animated:completion:`. In the completion block for that method you will set `userAuthenticated` to `YES` so that you can refer to it in the future.

If you haven't been given a `viewController`, you check if the player is already logged in. If so, you can just proceed.

If no player is logged in and you haven't been given a login view, then something has gone wrong and you log the error to the console and set the `userAuthenticated` variable to `NO`.

If it's running on iOS 5, then it calls the method covered earlier and acts the same. In this case the notification system will keep track of if the user has logged out and update `userAuthenticated` appropriately.

This new method of authentication makes it easier to handle Game Center authentication and requires less code for us devs, which is always welcome.

One issue that's not specific to the new authentication method, but worth mentioning at this point; it's possible that while the app was in the background, the previous player logged out, and someone else logged in to Game Center.

You will need to keep track of who is logged in and change the state of your game if that person has changed. For the most part, in this tutorial, you'll be fetching all the game state every time the view appears, so this isn't a problem. It's just going to load the games for the currently logged in player, whoever they might be.

OK – `GCTurnBasedMatchHelper` now contains all of the code necessary to authenticate the user, so you just have to use it! Switch to **ViewController.m** and import the header at the top of the file:

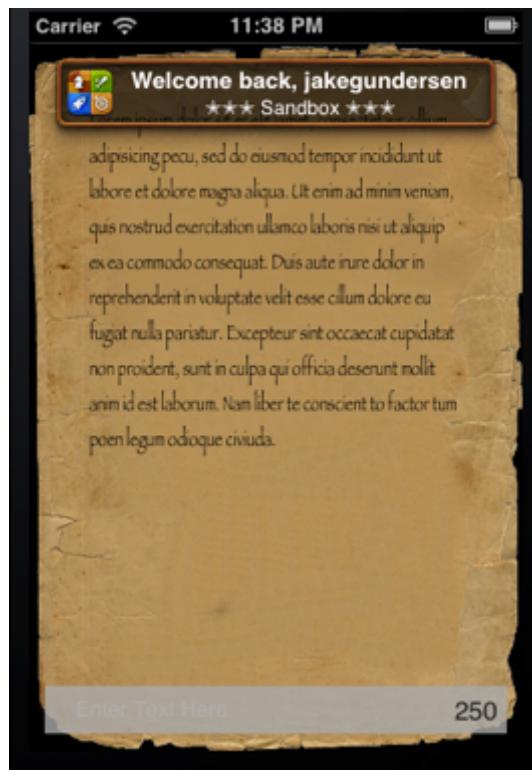
```
#import "GCTurnBasedMatchHelper.h"
```

Then add the following at the end of `viewDidLoad`:

```
[ [GCTurnBasedMatchHelper sharedInstance]
    authenticateLocalUserFromViewController:self];
```

This creates the Singleton instance (which registers for the “authentication changed” callback as part of initialization), then calls the `authenticateLocalUserFromViewController` method.

That's it! Compile and run your project, and if you're logged into Game Center you should see something like the following:



Turn Based Gaming basics

Now that you're successfully authenticating with Game Center and have a starter project ready, you're ready to start talking about the meat of this tutorial: Turn Based Gaming.

In a turn based game, you don't need to be playing simultaneously with your opponents (although you can be). For example, you can take your turn while your friend is asleep, and then they can wake up, take their turn while you're showering, and so on. A player can be in the middle of playing many of these asynchronous matches at the same time.

Visualize control of the game as a baton in a relay race. Only one player can hold the baton (or take a turn) at a time. When the baton is passed, it needs to contain everything that the player needs to know about that game.

To understand more about how it works, I'll start by reviewing the Turn Based Gaming classes in more detail.

GKTurnBasedMatch

This class stores information about an individual match, such as:

- **creationDate**: The date that the match was first created.
- **currentParticipant**: The `GKTurnBasedParticipant` who currently holds the baton (his/her turn). More on this below.
- **matchID**: An `NSString` uniquely identifying the match. This is typically long and not easily readable.
- **message**: An `NSString` to be displayed to help the user identify the match in the `GKTurnBasedMatchmakerViewController`. You can set this to whatever you want.
- **participants**: An `NSArray` of all `GKTurnBasedParticipants` who are included in the match (includes those who have quit).
- **status**: The current state of the match, as an `GKTurnBasedMatchStatus`. Includes values like Open, Ended, Matching, etc.
- **matchData**: An `NSData` object, up to 64k in iOS 6 in size that contains the data that you've defined for your game state.

GKTurnBasedParticipant

This class stores information about an individual player, such as:

- **playerID**: An `NSString` unique identifier about the player, never changes. This is not the same as the user's Game Center nickname, and you should usually not display this because it isn't easily readable.
- **lastTurnDate**: An `NSDate` of last turn. This is null until the player has taken a turn.

- **matchOutcome:** The outcome of the match as a `GKTurnBasedMatchOutcome`. Includes values such as Won, Lost, Tied, 3rd etc.
- **status:** The current state of the player, as a `GKTurnBasedParticipantStatus`. Includes values like Invited, Declined, Active, Done, etc.

GKTurnBasedMatchmakerViewController

This is the standard user interface written by Apple to help players work with turn-based gaming matches. It allows players to:

- **Create matches.** You can use this view controller to create matches, either by auto match or by invitation. When you create a new match, you get to play the first turn right away (even if the system hasn't found an auto match partner yet!) When the system has found someone and they take their turn, you'll get a notification again. Note that there is currently no programmatic way to create new matches except by using this controller.
- **Switch matches.** As discussed earlier, you can have many turn-based games going on at once. You can use this view controller to view different games you're playing - even if it's not your turn or the game is over (you can view the current game state in that case).
- **Quit matches.** Finally, you can use the view controller to quit from a match you no longer want to play.

GKMatchRequest

You use this to initialize a `GKTurnBasedMatchmakerViewController`, much the same way you create a match for normal (live) Game Center multiplayer games.

When you create a `GKMatchRequest`, you specify a minimum and maximum number of players. You can also segment players (by location, skill level, custom groups, etc.) using this object.

GKTurnBasedMatchmakerViewControllerDelegate

When you create a `GKTurnBasedMatchmakerViewController`, you can specify a delegate that implements this protocol. It provides callback methods for when the view controller loads a new match, cancels, fails due to error, etc.

GKTurnBasedEventHandler

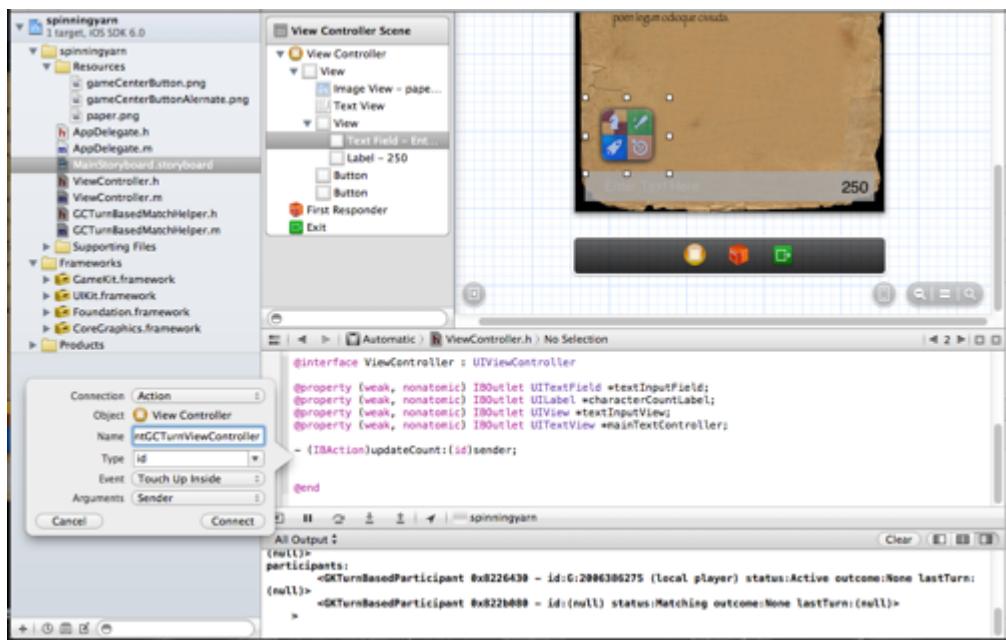
Last but not least, this singleton class has a delegate protocol, **GKTurnBasedEventHandlerDelegate** that provides notifications when the turn passes from one player to another, when you're invited by a friend to a match, or when the game ends.

Note: In iOS 5, the only way to save the match data to the Game Center servers was to send the turn to the next player. In iOS 6 there's a new method that allows you to save the match data between turns.

Turn Based Match view controller

The first thing you need to do to play a turn based match is create one! As discussed above, you can do that using Apple's provided `GKTurnBasedMatchmakerViewController`. You'll add a button you can use to present it to the screen.

Open **MainStoryboard.storyboard**, drag a button onto the view controller, set the type to Custom, remove the title, and set the image to `gameCenterButton.png`, as shown below:



You can hit command equals (Cmd=) to easily autosize the button to the same size as the image.

Then make sure the Assistant Editor is up and showing **ViewController.h**, and control-drag from the button down below the `@interface`. Set the Connection type to **Action**, name the method **presentGCTurnViewController**, and click Connect.

Note: Although I'm using the Game Center icon for this tutorial, you shouldn't use the game center icon in your apps. This practice is discouraged by Apple, and may cause an app to be rejected.

Their reasoning is that Game Center is not just the view controller. It includes leaderboards, achievements, view controllers (turn based and live), etc. Using this button creates an inconsistent user experience across different apps. So in your apps, you should use some other kind of visual icon or button.

Now, you're not actually going to do the heavy lifting to present the `GKTurnBasedMatchmakerViewController` inside of your `ViewController` class. Rather, you'll let the `GCTurnBasedMatchHelper` class do all that work for you. This way, the Game Center code will be nicely separated and more easily reusable in future projects.

So open up **GCTurnBasedMatchHelper.h** and add a new instance variable:

```
UIViewController *presentingViewController;
```

Also add a new method:

```
- (void)findMatchWithMinPlayers:(int)minPlayers  
    maxPlayers:(int)maxPlayers  
    viewController:(UIViewController *)viewController;
```

Here you create a new variable to store the view controller that will present the `GKTurnBasedMatchmakerViewController`, and a method that you'll use to present it to the screen to find a match with a specified number of players.

Next switch to **GCTurnBasedMatchHelper.m** and add the following method after `authenticateLocalUser`:

```
- (void)findMatchWithMinPlayers:(int)minPlayers  
    maxPlayers:(int)maxPlayers  
    viewController: (UIViewController *)viewController {  
  
    if (!_gameCenterAvailable) return;  
    presentingViewController = viewController;  
  
    GKMatchRequest *request = [[GKMatchRequest alloc] init];  
    request.minPlayers = minPlayers;  
    request.maxPlayers = maxPlayers;  
    GKTurnBasedMatchmakerViewController *mmvc =  
        [[GKTurnBasedMatchmakerViewController alloc]  
            initWithMatchRequest:request];  
    mmvc.turnBasedMatchmakerDelegate = self;  
    mmvc.showExistingMatches = YES;  
  
    [presentingViewController presentViewController:mmvc  
        animated:YES completion:nil];  
}
```

First, you check the status of `_gameCenterAvailable`. You can't do anything if game center isn't connected, so in that case you bail.

Once you establish that you're connected to game center, you set up a `GKMatchRequest`. All multiplayer Game Center games use this object, whether turn

based or live. You set the minimum and maximum players for the request. It will control the `GKTurnBasedViewController`, not allowing you to include more than your max players or less than your min players.

You're going to let up to 12 players play a game of SpinningYarn at a time. Because the game isn't live, the amount of data and bandwidth required for a turn based game is less intensive and so it's easier to have many players in a game. A live multiplayer game has a maximum of four players, but the turn-based game can support up to 16!

You then create a new `GKTurnBasedMatchmakerViewController`, passing in the `GKMatchRequest`. You set the delegate of that object to `self` (`GCTurnBasedMatchHelper`). This will throw an error. You'll fix that shortly.

Then you set the `showExistingMatches` property to `YES`. This property controls what's presented to the user. If you set it to `YES` then you'll see all the matches you have been involved in. This includes current matches where it's the player's turn, matches where it's some other player's turn, and matches that have ended.

There's a '+' button on the top right that can be used to create a new game. This presents a view that starts with the minimum number of players, and you can add players until you reach the specified max. Each slot can be filled with an invitation to a specific player, or can be an auto-match slot. If you set the `showExistingMatches` property to `NO`, then you'll be presented only with the create new game view.

Almost done - you just need to call this new method. Open `ViewController.h` and import the helper's header at the top of the file:

```
#import "GCTurnBasedMatchHelper.h"
```

Then switch to `ViewController.m` and replace the implementation of `presentGCTurnViewController` as follows:

```
- (IBAction)presentGCTurnViewController:(id)sender {
    [[GCTurnBasedMatchHelper sharedInstance]
        findMatchWithMinPlayers:2 maxPlayers:12
        viewController:self];
}
```

Build and run, and you'll see the `GKTurnBasedMatchmakerViewController` presented when you tap the game center button:



Of course, if you try to create a match or do anything else, the game will crash, because you haven't implemented the delegate methods yet! Let's fix that next.

Matchmaker view controller delegate

The first step is to open **GCTurnBasedMatchHelper.h** and modify the `@interface` as follows:

```
@interface GCTurnBasedMatchHelper : NSObject
<GKTurnBasedMatchmakerViewControllerDelegate> {
```

Here you simply mark your helper class as implementing the delegate protocol for the matchmaker view controller.

Next, switch to **GCTurnBasedMatchHelper.m** and add some placeholder implementations of the protocol at the end of the file:

```
- (void)turnBasedMatchmakerViewController:
(GKTurnBasedMatchmakerViewController *)viewController
didFindMatch:(GKTurnBasedMatch *)match {
    [presentingViewController dismissViewControllerAnimated:YES
                                                completion:nil];
    NSLog(@"did find match, %@", match);
}
```

```
- (void)turnBasedMatchmakerViewControllerWasCancelled:  
    (GKTurnBasedMatchmakerViewController *)viewController {  
    [presentingViewController dismissViewControllerAnimated:YES  
        completion:nil];  
    NSLog(@"has cancelled");  
}  
  
- (void)turnBasedMatchmakerViewController:  
    (GKTurnBasedMatchmakerViewController *)viewController  
didFailWithError:(NSError *)error {  
    [presentingViewController dismissViewControllerAnimated:YES  
        completion:nil];  
    NSLog(@"Error finding match: %@",  
        error.localizedDescription);  
}  
  
- (void)turnBasedMatchmakerViewController:  
    (GKTurnBasedMatchmakerViewController *)viewController  
playerQuitForMatch:(GKTurnBasedMatch *)match {  
    NSLog(@"playerquitforMatch, %@", "%@", match,  
        match.currentParticipant);  
}
```

The first method (`didFindMatch`) is fired when the user selects a match from the list of matches. This match could be one where it's currently your player's turn, where it's another player's turn, or where the match has ended.

The second method (`wasCancelled`) will fire when the cancel button is clicked. The third method (`didFail`) fires when there's an error. This could occur because you've lost connectivity or for a variety of other reasons.

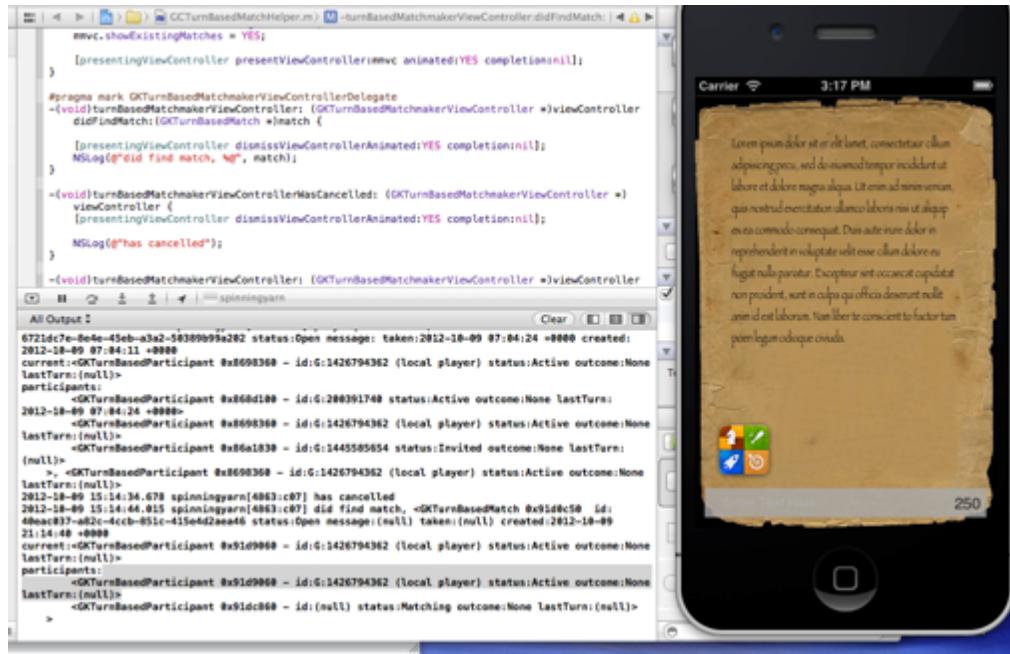
The final method (`playerQuitForMatch`) method is fired when a player swipes a match (while it's their turn) and quits it. Swiping a match will reveal a quit (if it's a match that is still active) or remove button. If a player quits a match while it's still their turn, they need to handle the match, update its state, and pass it along to the next player. If a player were to quit without passing the turn on to the next player, the match would not be able to progress forward!

A match that is finished will stay on Apple's servers and can be viewed by players that participated in it. If one player removes the match, it will no longer show up in that player's list of matches. However, because there are multiple players involved in a match, that match data still persists on the Game Center servers until all players have removed it.

In each of these methods you're simply dismissing the view controller and just logging out some information for now. An exception is the `playerQuit` method,

where you don't dismiss the view controller, because the user might want to keep doing something else.

Build and run now. Start a new match with an auto-matched player and you should have a log that looks like the following:

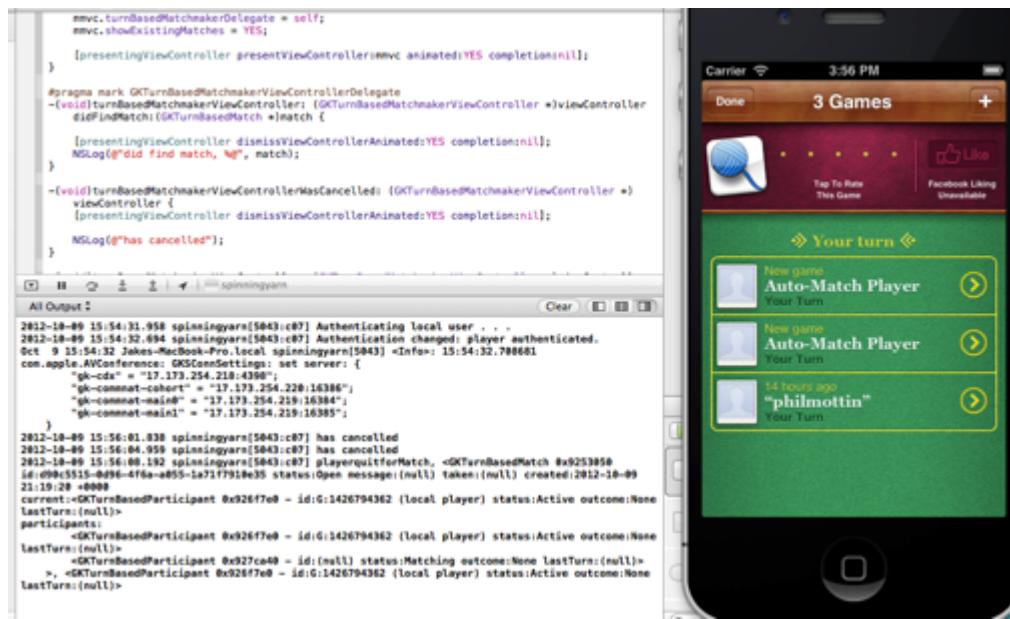


Note how the view controller dismisses immediately after you start a new match, even though it couldn't possibly have found a player to join your game yet! If you look at the logs, you'll see one of the participants is yourself (the local player) and the second has an id of null (it hasn't found a partner for you yet).

Note: In iOS 5 you would also have a participant with a null id for every auto-matched slot. In iOS 6, it only gives you one null participant, regardless of how many auto-match slots you've selected. This may be a bug. There's a thread in the Apple developer forum where it is being discussed.

It could also be a feature that enables the game to continue with a varying number of players. At the time of writing, it's unclear. Keep an eye on the [raywenderlich.com forums](#) for this chapter for an update.

Keep playing around to see if you can get the other log messages to appear. If you press the cancel button, or swipe a started match, you should see the cancelled or player quit messages, and if you disconnect your network and try to start a match you'll see the did fail message.



You're going to return to these methods and finish their implementation later. For now, leave them as they are.

Sending a turn

You need to set up a method that sends a turn. When it's your turn, you want to be able to add a string of text (max 250 characters) to the end of the current story. When you send a turn, you'll add that text to the previous text, then you'll send the whole string with your turn in the match data.

It's probably a good time to talk about the `GKTurnBasedMatch` object - remember this object is passed to you in the `didFindMatch` callback that gets called when the user creates a new match (or joins an existing one). All of the methods that signal a new turn will also give you a `GKTurnBasedMatch` to work with.

As you saw in the logs earlier, this object has a `participants` array that contains a `GKTurnBasedParticipant` for each player. You can find out whose turn it is by looking at the `currentParticipant` property.

It also contains a `matchData` property which holds an `NSData` object of up to 64k bytes that you'll use to record the state of the match (the string of the story for this game), which will be passed from one player to the next.

First, you will need to keep track of the current match, so you can modify the match data and have each player take their turn. So open `GCTurnBasedMatchHelper.h` and add the following after the `@interface`:

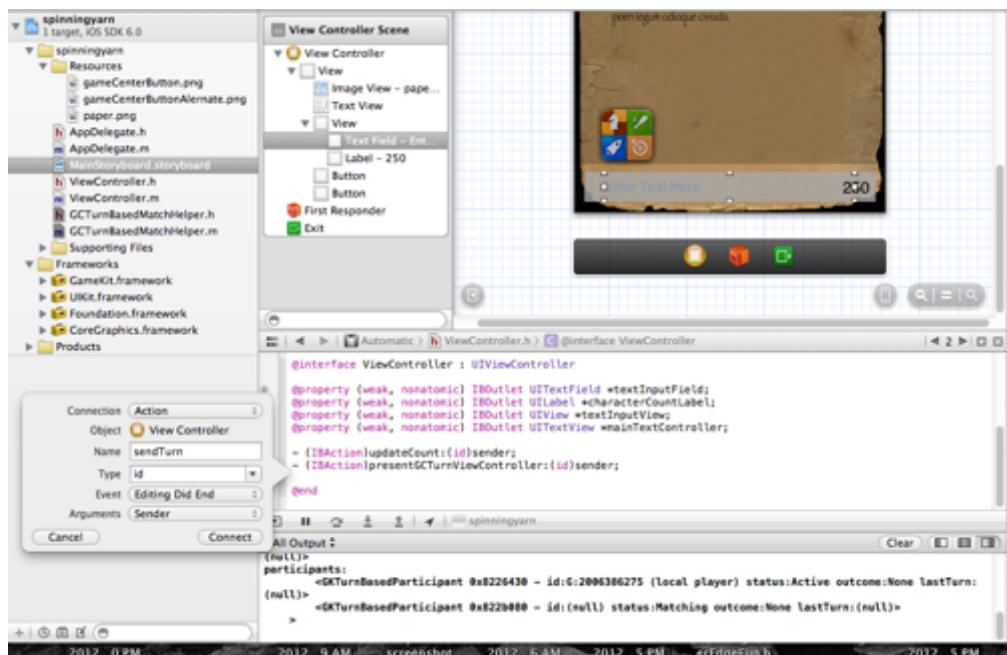
```
@property (strong) GKTurnBasedMatch * currentMatch;
```

Modify the `turnBasedMatchmakerViewController:didFindMatch:` callback to set the current match to the passed in match. Just add this to the bottom of the method:

```
self.currentMatch = match;
```

Now create a method to let the player take his turn (by adding a new bit of text to the story in this game). You'll call this method when the user enters some text in the text field and hits 'Done' on the keyboard.

Open up **MainStoryboard.storyboard**, and bring up the Assistant Editor so **ViewController.m** is visible. Control-drag from the text field down below the @interface to bring up the Connection popup. Set the Connection to **Action**, the name to **sendTurn**, and click Connect.



Now implement the method as follows:

```
- (IBAction)sendTurn:(id)sender {
    GCTurnBasedMatch *currentMatch =
        [[GCTurnBasedMatchHelper sharedInstance] currentMatch];

    NSString *newStoryString;
    if ([self.textInputField.text length] > 250) {
        newStoryString =
            [self.textInputField.text substringToIndex:249];
    } else {
        newStoryString = self.textInputField.text;
    }

    NSString *sendString = [NSString stringWithFormat:@"%@ %@",
```

```
self.mainTextController.text, newStoryString];

NSData *data =
[sendString dataUsingEncoding:NSUTF8StringEncoding ];

self.mainTextController.text = sendString;

NSUInteger currentIndex = [currentMatch.participants
indexForObject:currentMatch.currentParticipant];

NSMutableArray *participants = [NSMutableArray array];
for (int i = 0; i < [currentMatch.participants count]; i++)
{
    int indx = (i + currentIndex + 1) %
[currentMatch.participants count];
[participants addObject:[currentMatch.participants
objectAtIndex:indx]];
}

[currentMatch endTurnWithNextParticipants:participants
turnTimeout:600 matchData:data completionHandler:
^(NSError *error) {
    if (error) {
        NSLog(@"%@", error);
    }
}];

NSLog(@"Send Turn, %@", data, participants);
self.textField.text = @"";
self.characterCountLabel.text = @"250";
self.characterCountLabel.textColor = [UIColor blackColor];
}
```

This method is doing a whole bunch of things. I'll step through each bit.

First you set up the `currentMatch` variable by retrieving the match from the `GCTurnBasedMatchHelper` singleton. Your user may be involved in many matches at a time, but you'll only be displaying one match at a time. You'll keep track of this one match in your `currentMatch` variable.

Next you need to check to see if the length of the string you input into the text field is too long. If the string is longer than 250 characters you cut it off with the `substringToIndex` call. If not, you just store the string in the `newStoryString` variable.

Next you create an `NSData` object by combining the string that's in the `mainTextController` with the string you just created. The `dataUsingEncoding`

method converts the `NSString` representation to a UTF8 string and stores it in the `data` object.

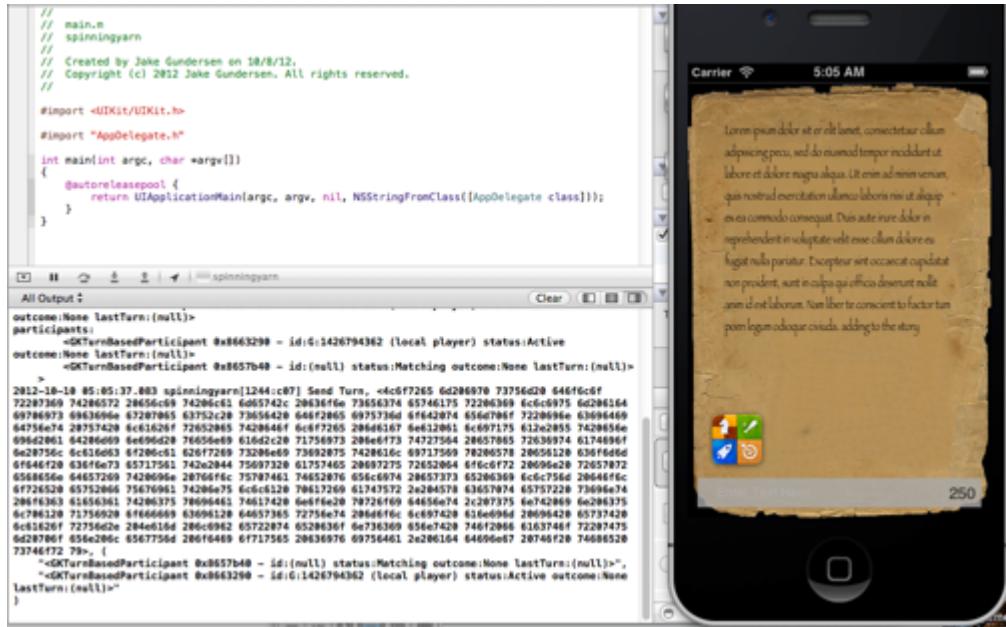
The next few lines set up the `nextParticipants` array. Each time you send a turn you need to send information about next people in the turn rotation. You can set up your game to send the turn to any set of participants, in any order, including the same player. You can follow a strict rotation where the order of players always follows the same order. Or, you can set up other rules about who gets the next turn.

In order to build the array, you first get the index of the current participant in the matches participants array. You create a loop with the same number of iterations as the participants in the current match. You start by getting the next person in the participants array by adding one to the `currentIndex` variable (and then applying the modulo operator). When you are done you have a new array that starts with the next person after the current participant.

Once you have the `NSData` and the `nextParticipant` set, you can make the `endTurnWithNextParticipant:matchData:completionHandler:` call. The completion handler is a block that will be called when ending the turn successfully completes (or fails). In your case you are just logging any error that occurs.

You also log out the `nextParticipants` array and the `data` object. You can take a look at this to see if this is working as expected. The last action is to clear the contents of the `textInputField` and reset the `characterCountLabel` to 250 so when you load the next game, it's not cluttered with old text.

Build and run now, and start a new match with the match controller. Type something into the text field and press done. You should see something like the following in your log:



The screenshot shows the Xcode interface with the log output on the left and an iPhone simulator on the right. The log output shows the following sequence of events:

```

// main.m
// spinningyarn
// Created by Jake Gunderson on 10/8/12.
// Copyright (c) 2012 Jake Gunderson. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "AppDelegate.h"

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

2012-10-18 05:05:03 spinningyarn[1244:c07] Send Turn, <4cf7f265 6d206970 73756d08 646f6cd6f
72287369 74286572 28656c9 74286c63 28636f6e 73656374 65746175 72286369 6cd69379 6d206164
69786954 69656966 67286953 63765248 73656420 646f6d65 69757364 6f6e6974 856e6975 72286966 63696469
64776374 64656374 72652649 74286572 28636f6e 73656374 65746175 61286369 74286564
69857362 64286669 6e765429 72652649 65746175 74286572 28636f6e 65746175 61286369 74286564
6e28756c 6c126083 61286129 73286520 73286529 73286528 74286126 69737569 78286579 28658120 636f6d66
6f646129 636f6e73 65717561 74286244 75697329 61757465 28697275 72652664 6f6c6f72 28696209 72557872
65686564 64657269 74286564 28766f6c 75787461 74652876 656c6974 28657373 65286369 6cd7c564 20646f6c
6f726529 65752664 75670661 74286675 6c6c6129 70617260 61747572 2e286579 63657874 65757220 73696674
286f6363 61565361 74286375 76069643 74617428 6e6f6e62 78726169 64656674 2e287373 6e742866 6d206375
6e776129 73758929 6f666669 63698128 64657365 72756674 28686f6e 6c697429 816e6964 28696428 65737428
6e6f6e62 77566209 2846f164 286c6962 65722874 65286367 6e736369 656e7429 74612966 61637481 72287475
6d287861 656e7260 85677564 28616469 61717563 28636876 69756461 2e286164 64696e67 28746f29 74606674
73746172 739... <4GKTurnBasedParticipant 0x8657b40 - id:(null) status:Matching outcome:None lastTurn:(null)>,
<4GKTurnBasedParticipant 0x8663290 - id:1426794362 (local player) status:Active outcome:None lastTurn:(null)>
}

```

The iPhone simulator shows a game screen with a parchment scroll containing placeholder Latin text: "Lorem ipsum dolor sit et est bene, connectetur cladem adipiscitur pecus, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupiditate non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Non liber te consciente factorum pain legum edocere cuncta, addendo to the story". Below the scroll, the character count is displayed as 250.

Taking turns

There are several ways that you can take a turn in your app. When you use auto-match for a slot in your game, one of two things will happen. Either

1. You'll get back a new match, with your player as the first participant, or
2. You'll get back an existing match where your player has been placed in a second, third, fourth, etc slot in an existing match.

In both scenarios, `didFindMatch` will be called. You'll need to write some code to distinguish between the two cases, because you want different things to happen in each case:

If you get back a new match, you want to create a clean slate for your story. In this app, you'll use the string "Once upon a time..." every time you begin a new story.

If you get back an existing match, then the `match.matchData` object will be populated with `NSData` with the story so far. Instead of the starter text, you want to show this in the text view.

You just need to use a reliable test to distinguish between these two cases and deal with them differently.

Each participant has a `lastTurnDate` property that is null until that participant takes his first turn. You're going to use the `lastTurnDate` property of the first participant in the `participants` array to determine if you are the first turn, or if another player has started the match. If you find that `lastTurnDate` is null, you'll assume that you're dealing with a new match, otherwise you'll assume that you already have `matchData` that you'll be dealing with.

So open up **GCTurnBasedMatchHelper.m** and replace the `turnBasedMatchmakerViewController:didFindMatch` method as follows:

```
- (void)turnBasedMatchmakerViewController:(GKTurnBasedMatchmakerViewController *)viewController didFindMatch:(GKTurnBasedMatch *)match {

    [presentingViewController dismissViewControllerAnimated:YES completion:nil];
    self.currentMatch = match;

    GKTurnBasedParticipant *firstParticipant =
        [match.participants objectAtIndex:0];
    if (firstParticipant.lastTurnDate) {
        NSLog(@"existing match");
    } else {
        NSLog(@"new match");
    }
}
```

{

You're getting the participant at the first index in your `participants` array, then testing to see if the `lastTurnDate` is populated. If it is, you know you're entering a match that has already started. If not, you know it's a new match and you are logging the case either way.

You can run most of these tests on the simulator. When you start writing the `GKTurnBasedEventHandler` methods, these will only work on a device. However, most of these tests will work on both the simulator and the device, so you can log in to two different sandbox accounts and test between one device and the simulator. Make sure that you are logged into the sandbox and not the regular Game Center account. You can tell if you are in the sandbox by opening game center:



In order to log into the sandbox:

1. Log out of game center,
2. Start spinning yarn
3. It will prompt you to login to game center

The only difference is that you aren't logging in from the game center app (but you are logging out there).

So, run the app on two devices (run one of them in the debugger so you can see its log messages). Join a few matches and take your turn on each device, watching

what shows up in your log. You should see some new matches, and hopefully some existing matches too, which proves the match-making worked!

At the time of writing this tutorial, the time between creating a new auto-match and being able to jump in to the second player position can be up to five minutes. This will probably get much faster by the time you are reading it, but don't be alarmed if you start an auto-match on one device and immediately on a second device instead of putting the second player into the first player's game, it creates another new match for the second player.

One more word about auto-match while I'm on the subject. Auto-match doesn't start matching until the first player is finished taking his/her turn. This is true for an invited slot as well. If you start a game with three players, the third player won't be invited or auto-matched until after the first and second player have both taken their turn.

Implementing a new delegate protocol

Now is a good time to introduce a new delegate protocol. You are going to build a protocol to manage the communication between the `GCTurnBasedMatchHelper` class and `ViewController`. This is better than integrating the `ViewController` into the `GCTurnBasedHelper`, because it will allow you to more easily reuse this class.

The `GCTurnBasedMatchHelper` will do some of the work, like distinguishing between a new and existing match, but will then pass the match through to your `ViewController` class to handle what do for each case, because that's particular to the game logic.

Go ahead and build the delegate protocol now. Add the following `@protocol` declaration to **`GCTurnBasedMatchHelper.h`**:

```
@protocol GCTurnBasedMatchHelperDelegate
- (void)enterNewGame:(GKTurnBasedMatch *)match;
- (void)layoutMatch:(GKTurnBasedMatch *)match;
- (void)takeTurn:(GKTurnBasedMatch *)match;
- (void)recieveEndGame:(GKTurnBasedMatch *)match;
- (void)sendNotice:(NSString *)notice forMatch:(GKTurnBasedMatch *)
* )match;
@end
```

Also, add a this new property for the delegate:

```
@property (nonatomic, assign) id <GCTurnBasedMatchHelperDelegate>
delegate;
```

The delegate object will be sent the methods, and it will be up to that delegate (in your case the `ViewController` class) to implement them.

Let's quickly go through each method now. Then later when you implement them I'll explain them in more detail.

1. **enterNewGame**. When the user is presented with a new game from the `didFindMatch` method, you want to display the "Once upon a time" starter text to the screen.
2. **layoutMatch**. The `layoutMatch` method is used when you want to view a match where it's another player's turn (just to check the state of the story for example). You want to prevent the player from sending a turn in this case, but you still want to update the UI to reflect the most current state of the match.
3. **takeTurn**. The `takeTurn` method is for those cases when it is your player's turn, but it's an existing match. This scenario exists when your player chooses an existing match from the `GKTurnBasedMatchmakerViewController`, or when a new turn notification comes in. I'll talk about notifications a little later in this tutorial.
4. **recieveEndGame**. The `receiveEndGame` method will be called when a match has ended on your player's turn, or when you receive a notification that a match has ended on another player's turn. For this simple game, you'll just end the game when you are getting close to the current `NSData` turn-based game size limit (4096 bytes).
5. **sendNotice**. The `sendNotice` method happens when you receive an event (update turn, end game) on a match that isn't one you're currently looking at. If you receive an end game notice on a match that you've got loaded into your `currentMatch` variable, you'll update the UI to reflect the current state of that match, but if you receive the same notice on a match other than the one you're looking at, you don't want to automatically throw the user into that match, taking them away from the match they are currently looking at. You'll decide how to handle this later on.

Go ahead and send the delegate methods in your `turnBasedMatchmakerViewController:didFindMatch` method for your new match and existing match scenarios. Replace the `NSLog` methods with calls to the delegate methods, like so:

```
- (void)turnBasedMatchmakerViewController:(GKTurnBasedMatchmakerViewController *)viewController didFindMatch:(GKTurnBasedMatch *)match {
    [presentingViewController dismissViewControllerAnimated:YES completion:nil];
    self.currentMatch = match;
    GKTurnBasedParticipant *firstParticipant = [match.participants objectAtIndex:0];
    if (firstParticipant.lastTurnDate) {
        //New Code
        [self.delegate takeTurn:match];
    }
}
```

```
    } else {
//New Code
    [self.delegate enterNewGame:match];
}
}
```

Next, open up **ViewController.m** and mark the class as implementing the new protocol:

```
@interface ViewController() <UITextFieldDelegate,
GCTurnBasedMatchHelperDelegate>
```

Then, in the same file, implement the `enterNewGame:` and `takeTurn:` methods:

```
-(void)enterNewGame:(GCTurnBasedMatch *)match {
    NSLog(@"Entering new game...");
    self.mainTextController.text = @"Once upon a time";
}

-(void)takeTurn:(GCTurnBasedMatch *)match {
    NSLog(@"Taking turn for existing game...");
    if ([match.matchData bytes]) {
        NSString *storySoFar = [NSString
stringWithUTF8String:[match.matchData bytes]];
        self.mainTextController.text = storySoFar;
    }
}
```

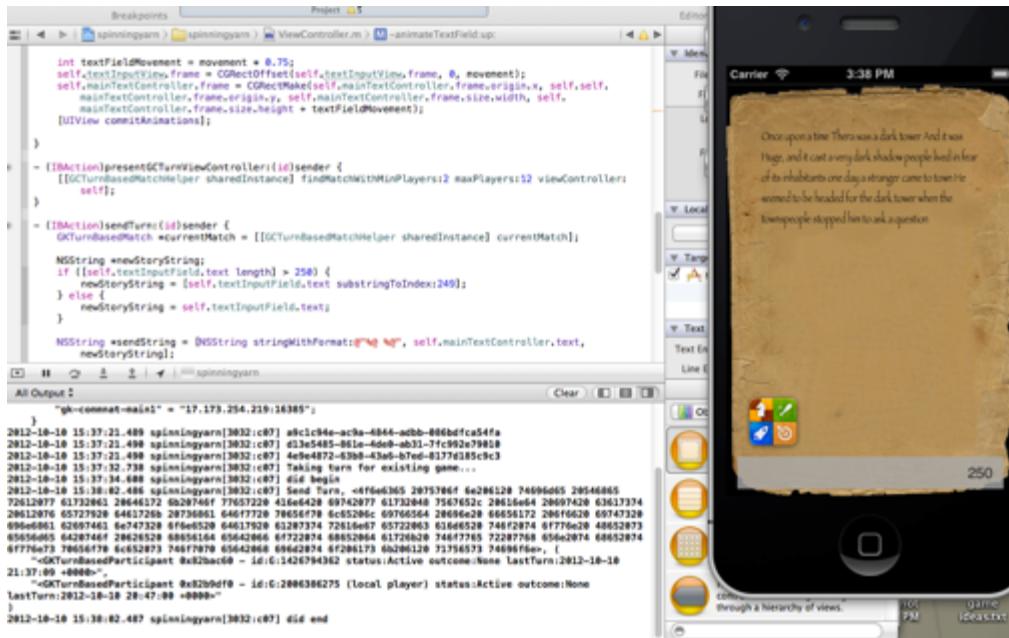
Both methods are logging their action to make it easier for you to verify that your app is working properly by watching the console.

The first method just creates the beginning of a new story and places it in the `mainTextController` text view. The second checks first to make sure that there is some content in `matchData` (because if not the next line will cause it to crash). Then it just sets the value of the `mainTextController` to the string that is contained in `matchData`.

One last thing you must do is set the delegate property of the `GCTurnBasedMatchHelper` class to your `viewController`. Let's do that in the `viewDidLoad` method at the end, like so:

```
[GCTurnBasedMatchHelper sharedInstance].delegate = self;
```

Build and run on both devices. You should be able to pass turns around!



Start a new game (either by invitation or auto-match) and take the first turn. Give it a few minutes to make sure the server is updated, then enter that game from a second device.

If you have created participants by invitation you should have a game waiting in the 'your turn' section, if done by auto-match create another auto-match game on the other device with the same number of players. Again, the auto-match won't always put you into the match you want, it does create new matches frequently.

Note that after the other side takes a turn, your screen won't update because you haven't added the code in for that. You can get the game to recognize it's your turn by tapping the game center button, and selecting the game.

Too many matches?

At this point I have about five matches that have wrong data and other problems with them. Some of your old games may act wrong because the data in them is incomplete or missing. I'd remove all of them and start fresh. A good way to clean out all this data is to call a method that programmatically deletes all the matches for a user.

Create this new method in **GCTurnBasedMatchHelper.m**:

```
-(void)deleteAllMatches {
    [GKTurnBasedMatch loadMatchesWithCompletionHandler:
     ^(^NSArray *matches, NSError *error){
        for (GKTurnBasedMatch *match in matches) {
            NSLog(@"%@", match.matchID);
        #if !defined(__IPHONE_6_0) || __IPHONE_OS_VERSION_MIN_REQUIRED <
            __IPHONE_6_0
    
```

```
[match removeWithCompletionHandler:  
    ^(NSError *error) {  
        NSLog(@"%@", error);  
    }];  
  
#else  
  
[match participantQuitOutOfTurnWithOutcome:  
    GKTurnBasedMatchOutcomeTied  
    withCompletionHandler:^(NSError *error) {  
        NSLog(@"%@", error);  
        [match removeWithCompletionHandler:  
            ^(NSError *error) {  
                NSLog(@"%@", error);  
            }];  
    }];  
  
#endif  
}  
}  
}
```

This just calls for all the matches that the logged in user is currently involved in, then deletes them one at a time. It logs each match before it deletes it. If there's a problem, it will log the error to the console. One problem that might see is that you can't delete a match unless you hold the 'baton' (it's your turn).

Note that there is a slightly different way to delete a match on iOS 6 – you first have to mark the participant as having quit first.

I've also had some weird matches that I can't delete or access for unknown reasons. Don't fret if you can't get everything deleted, this is just to make it easier while you're testing. A rogue match or two isn't going to get in the way.

Modify the `authenticateLocalUserFromViewController:` method as follows:

```
- (void)authenticateLocalUserFromViewController:  
    (UIViewController *)authenticationPresentingViewController {  
    if (![_gameCenterAvailable]) return;  
  
    NSLog(@"Authenticating local user . . .");  
    presentingViewController =  
        authenticationPresentingViewController;  
  
    GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];  
  
    #if !defined(__IPHONE_6_0) || __IPHONE_OS_VERSION_MIN_REQUIRED <  
    __IPHONE_6_0  
    localPlayer.authenticateHandler =  
        ^(UIViewController *viewController, NSError *error)  
    }
```

```
{  
    if (viewController) {  
        [authenticationPresentingViewController  
            presentViewController:viewController animated:YES  
            completion:^{  
                userAuthenticated = YES;  
                [self deleteAllMatches];  
            }];  
    } else if (localPlayer.authenticated) {  
        userAuthenticated = YES;  
        [self deleteAllMatches];  
    } else {  
        userAuthenticated = NO;  
        NSLog(@"Error with Game Center %@", error);  
    }  
};  
  
#else  
    if (localPlayer.authenticated == NO ) {  
        [[GKLocalPlayer localPlayer]  
            authenticateWithCompletionHandler:nil];  
        [self deleteAllMatches];  
    } else {  
        NSLog(@"Already authenticated");  
        [self deleteAllMatches];  
    }  
#endif  
}
```

If you look through the above method, you can see that you've put the call to `deleteAllMatches` in four places. You probably need to put it in one or two places only, depending on which OS you're running.

Remember to comment it out after you use it – you don't want to start every time with a clean slate!

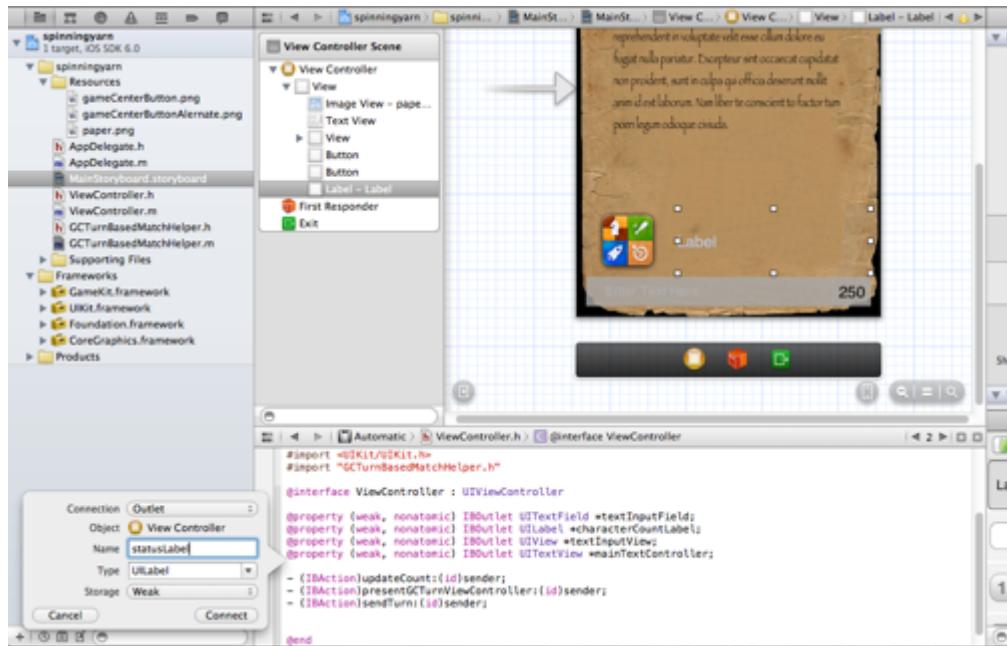
What to do when it's not your turn

Currently you have the ability to input text and run the game while it's not your turn. While the API prevents you from updating the game state outside our turn, your app will throw errors and it would be better to indicate to the player when they are unable to enter text.

When it's not the current player's turn, you want to update a status label telling the player that the match is currently in another player's turn. Also, you should disable the text field.

Open up **MainStoryboard.storyboard**. Drag a label next to the Game Center icon, and make it as wide as the screen. Also set the number of lines to 0 (which means unlimited lines) in the attributes inspector so the text can wrap.

Bring up the Assistant Editor, make sure **ViewController.m** is visible, control-drag from the label down inside the `@interface` and connect it to an outlet named **statusLabel**.



You'll need to make some changes in both the **GCTurnBasedMatchHelper** class and the **ViewController** class in order to keep track of whose turn it is. Let's start by editing your **turnBasedMatchmakerViewController:didFindMatch** method in **GCTurnBasedMatchHelper.m**:

```
- (void)turnBasedMatchmakerViewController:
(GKTurnBasedMatchmakerViewController *)viewController
didFindMatch:(GKTurnBasedMatch *)match {

    [presentingViewController dismissViewControllerAnimated:YES
                                                completion:nil];
    self.currentMatch = match;

    GKTurnBasedParticipant *firstParticipant =
        [match.participants objectAtIndex:0];
    if (firstParticipant.lastTurnDate == NULL) {
        //It's a new game
    }
}
```

```
[self.delegate enterNewGame:match];
} else {
    if ([match.currentParticipant.playerID
isEqualToString:[GKLocalPlayer localPlayer].playerID]) {
        //It's your turn
        [self.delegate takeTurn:match];
    } else {
        //Someone else's turn
        [self.delegate layoutMatch:match];
    }
}
```

You now have a three-pronged logic tree instead of just two. If the `lastTurnDate` is `NULL`, then it's a brand new match and you send the appropriate method as before. However, if it's not a new game, you need to determine if the current participant is the same as the local player or not.

You do that by comparing the `playerID` of the current participant with the `playerID` of the local player. If it is the local player, then you send the `takeTurn` delegate method, otherwise you'll send the `layoutMatch` method.

The `layoutMatch` method will just load the current match's state, without allowing the player to take any action that would alter the `matchData`. This may happen if it's someone else's turn, or in the case where the match has ended.

Here's the implementation for the `layoutMatch` method. This code should go in **ViewController.m**:

```
-(void)layoutMatch:(GKTurnBasedMatch *)match {
    NSLog(@"Viewing match where it's not our turn...");
    NSString *statusString;

    if (match.status == GKTurnBasedMatchStatusEnded) {
        statusString = @"Match Ended";
    } else {
        int playerNum = [match.participants
            indexOfObject:match.currentParticipant] + 1;
        statusString = [NSString stringWithFormat:
            @"Player %d's Turn", playerNum];
    }

    selfStatusLabel.text = statusString;
    self.textField.enabled = NO;
    NSString *storySoFar = [NSString
        stringWithUTF8String:[match.matchData bytes]];
```

```
    self.mainTextController.text = storySoFar;
}
```

First, you construct the string you put in `statusLabel`. You check the `GKTurnBasedMatchStatus` `match.status` and set the string based on whether the game is running or ended.

If you currently waiting on another player, you want to get the position of the player in the array. The array index starts at zero, but human beings start a list at one, so you'll add one to get a human readable position ☺. You construct the string for the player's turn and set the label to that string.

Next, you are disabling the `textInputField` so that your player won't have the ability to enter text. Finally, as you have before, you update the `mainTextController` to hold the body of the story.

Let's go back and make a few edits to your other two methods to incorporate the new label and the logic that will turn on the text field. Modify the following two methods:

```
- (void)enterNewGame:(GKTurnBasedMatch *)match {
    NSLog(@"Entering new game...");
    int playerNum = [match.participants
        indexOfObject:match.currentParticipant] + 1;
    selfStatusLabel.text = [NSString stringWithFormat:
        @"Player %d's Turn (that's you)", playerNum];
    self.textField.enabled = YES;
    self.mainTextController.text = @"Once upon a time";
}

-(void)takeTurn:(GKTurnBasedMatch *)match {
    NSLog(@"Taking turn for existing game...");
    int playerNum = [match.participants
        indexOfObject:match.currentParticipant] + 1;

    NSString *statusString = [NSString stringWithFormat:
        @"Player %d's Turn (that's you)", playerNum];

    selfStatusLabel.text = statusString;
    self.textField.enabled = YES;

    if ([match.matchData bytes]) {
        NSString *storySoFar =
            [NSString stringWithUTF8String:[match.matchData bytes]];
        self.mainTextController.text = storySoFar;
    }
}
```

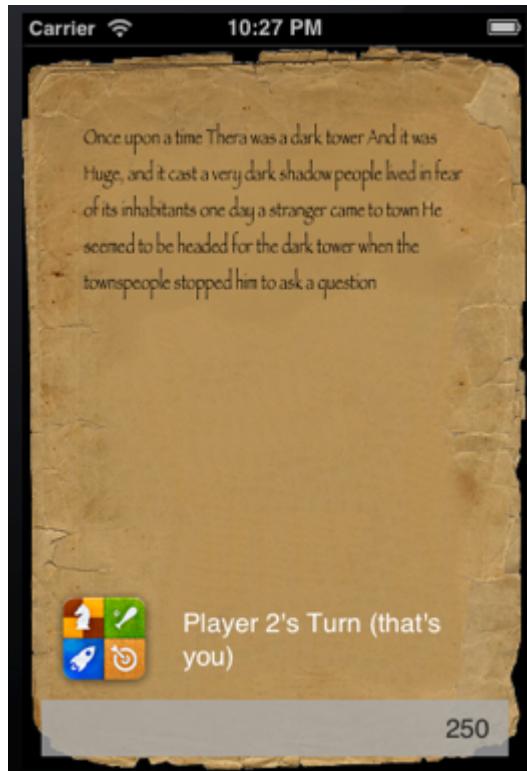
In both these methods you are calculating the player's position and updating the status label to let them know that it's their turn. You also enable the `textInputField` object so that they can click on it and add to the story. The rest of the code in these methods was already included.

It's time to set the initial value and state of your new `statusLabel` and `textInputField`. First change the `statusLabel` to two lines and make it tall enough to fit two lines in your storyboard file. Now, add the following code to the end of the `viewDidLoad` method:

```
self.textField.enabled = NO;
self.statusLabel.text = @"Welcome. Press Game Center to get
started.";
```

Build and run, and start a new game or enter an existing game to enter text. You'll notice that the status label now gives you a better indication of where you are in the match. It doesn't correctly update the state when you take a turn yet though.

Also, you still have to go back into Game Center and re-select a game to get updates, but you're one step closer to a functional game!



Finishing the Delegate Methods

You're finished with the `didFindMatch` method, but you still have to finish up the rest of the `GKMatchmakerViewController` delegate methods.

The `didCancel` method only needs to dismiss the view controller, so it's fine as is. The error method is also satisfactory as is, in a polished implementation you'd want to handle the various errors in a more elegant way, but for your purpose here, logging the error is fine.

But you do have to change the

`turnBasedMatchmakerViewController:playerQuitForMatch:` method, so update it as follows in **GCTurnBasedMatchHelper.m**:

```
- (void)turnBasedMatchmakerViewController:
  (GKTurnBasedMatchmakerViewController *)viewController
  playerQuitForMatch:(GKTurnBasedMatch *)match {

    NSUInteger currentIndex = [match.participants
      indexOfObject:match.currentParticipant];
    GKTurnBasedParticipant *part;

    NSMutableArray *nextParticipants = [NSMutableArray array];
    for (int i = 0; i < [match.participants count]; i++) {
      part = [match.participants objectAtIndex:
        (currentIndex + 1 + i) % match.participants.count];
      if (part.matchOutcome == GKTurnBasedMatchOutcomeNone) {
        [nextParticipants addObject:part];
      }
    }

    [match participantQuitInTurnWithOutcome:
      GKTurnBasedMatchOutcomeQuit
      nextParticipants:nextParticipants turnTimeout:600
      matchData:match.matchData completionHandler:nil];

    NSLog(@"playerquitforMatch, %@", match,
      match.currentParticipant);
}
```

If the player currently holds the baton and quits a match, that match is stuck. This is because only the player with the baton can submit a turn, but that player has quit!

To fix this, you need to add some code so that if the player quits during their turn, it hands the baton off to another player first. That's what this method does.

This method is called when you quit a game from the view controller and it's our turn. If it's not our turn and we quit, then another method, `playerQuitOutOfTurn`, is called for you and all that is dealt with automatically.

The trick here is to iterate through a list of the participants for the match and create an array including all those participants with a `GKTurnBasedMatchOutcomeNone`.

If the participant is no longer playing, they will have any number of other turn based match outcomes. Turn based match outcomes include quit, tied, first, second, third, etc.

You start with the participant that comes after the current participant. Then include, in order, all the participants that are still playing.

When you finish creating the participants array, you call

`participantQuitInTurnWithOutcome:nextParticipants:turnTimeout:matchData:completionHandler:` which assigns an outcome to the quitting player (in this case, quit) passes the match to the next player, and end the turn.

In this game, you don't need to do anything with the `matchData`, but pass it on. In other scenarios, the game may require something to be done to the game state before it can be passed on.

While you're fixing this, you should make some of the same changes to your `sendTurn` method. In a case like this one, you want to iterate through the participants and make sure the one you're passing the match to hasn't quit.

In fact, if you build and run now, start a match with three players (in a two player game if one quits the game ends), go around a few times, then have a player quit, then try to pass the quit player the match, you'll see this:

```
20746865 72652077 61732061 206e6577 2073746f 72792061 626f7574 20616e20 61707020 77697468 20627567  
73207468 61742065 76656e74 75616c6c 7920676f 74206669 78656420 73686f75 6c642074 68696e6b 2061626f  
7574<, <GKTurnBasedParticipant 1d4160 - id:G:1445585654 status:Active outcome:None lastTurn:  
2011-10-07 17:27:57 +0000>  
2011-10-07 11:29:42.375 spinningyarn[238:707] Error Domain=GKErrorDomain Code=3 "The requested  
operation could not be completed due to an error communicating with the server." UserInfo=0x126db0  
{NSUnderlyingError=0x1832a0 "The operation couldn't be completed. status = 5003, Session:  
33fb35fb-7df6-482a-942d-d79e9f855071 current turn: 6 isn't as expected: 3",  
NSLocalizedDescription=The requested operation could not be completed due to an error communicating  
with the server.}
```

To fix this, change the `sendTurn` method in **ViewController.m** to this:

```
- (IBAction)sendTurn:(id)sender {  
    GKTurnBasedMatch *currentMatch =  
        [[GCTurnBasedMatchHelper sharedInstance] currentMatch];  
  
    NSString *newStoryString;  
    if ([self.textInputField.text length] > 250) {  
        newStoryString =  
            [self.textInputField.text substringToIndex:249];  
    } else {  
        newStoryString = self.textInputField.text;  
    }  
  
    NSString *sendString = [NSString stringWithFormat:@"%@ %@",  
                           self.mainTextController.text, newStoryString];  
  
    NSData *data =  
        [sendString dataUsingEncoding:NSUTF8StringEncoding];
```

```
self.mainTextController.text = sendString;

NSUInteger currentIndex = [currentMatch.participants
    indexOfObject:currentMatch.currentParticipant];

NSMutableArray *nextParticipants = [NSMutableArray array];
for (int i = 0; i < [currentMatch.participants count]; i++){
    int indx = (i + currentIndex + 1) %
        [currentMatch.participants count];
    GKTurnBasedParticipant *participant =
        [currentMatch.participants objectAtIndex:indx];
    //1
    if (participant.matchOutcome ==
        GKTurnBasedMatchOutcomeNone) {
        [nextParticipants addObject:participant];
    }
}

[currentMatch endTurnWithNextParticipants:nextParticipants
    turnTimeout:600 matchData:data completionHandler:
    ^(NSError *error) {
        if (error) {
            NSLog(@"%@", error);
            //2
            selfStatusLabel.text = @"Oops, there was a problem.
Try that again.";
        } else {
            //3
            selfStatusLabel.text = @"Your turn is over.";
            self.textField.enabled = NO;
        }
    }];
}

NSLog(@"Send Turn, %@", data, nextParticipants);
self.textField.text = @"";
self.characterCountLabel.text = @"250";
self.characterCountLabel.textColor = [UIColor blackColor];
}
```

1. In the first marked section, you are modifying the method that creates the `nextParticipants` array to include only those participants that have a `matchOutcome` of `GKTurnBasedMatchOutcomeNone`. In this way, you can avoid passing a match to a participant that is no longer player.

2. In the second section you notify the user if there's a problem sending the match to the next player, by setting the `statusLabel`.
3. In the third section, you are setting the `statusLabel` and disabling interaction with the `textInputField`. This way, you won't allow them to call the `sendTurn` method when it's not their turn.

Build and run now, and you'll see that the status is updated when you send a turn!



Event Handler Delegate

Your game is coming along well so far, but there's one major problem - you never get updated when the other players take their turn! It's quite annoying having to constantly check by bringing up the Game Center UI.

As you've been playing with the app, you may have noticed you sometimes get badges and/or system notifications of new turns/invitations to your game. This is being done by the `GKTurnBasedEventHandler` object. This object sends notifications and badges to your app when certain events happen, like when it's your player's turn.

Note: The `GKTurnBasedEventHandler` delegate only works on the device. So, if you're passing turns between the simulator and your device, you're only going to get these callbacks on the device.

There are three delegate callbacks that give your app a way to deal with these notifications as they come in. One is the method that gets called if you start an invite for that game from within the game center app, one when the turn advances (even if it's not your turn, there's a callback every time the match changes hands), and one is fired when the game ends.

In order to receive and handle these events, you first need to set yourself as the delegate of the `GKTurnBasedEventHandler`. This object is a singleton and the only time you deal with it directly is when you set yourself as its delegate.

Like the view controller delegate protocol, you're going to be using the `GCTurnBasedMatchHelper` class to act as an intermediary for all the communication from these notifications. So, that's what you need to set up as the delegate. You have to set the delegate after you have logged in to game center or it may not work. Change the `authenticateLocalUserFromViewController:` method to the following:

```
- (void)authenticateLocalUserFromViewController:
    (UIViewController *)authenticationPresentingViewController {
    if (! _gameCenterAvailable) return;

    NSLog(@"Authenticating local user . . .");
    presentingViewController =
        authenticationPresentingViewController;

    GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];

#ifndef __IPHONE_6_0 || __IPHONE_OS_VERSION_MIN_REQUIRED <
__IPHONE_6_0
    localPlayer.authenticateHandler =
        ^(UIViewController *viewController, NSError *error)
    {
        if (viewController) {
            [authenticationPresentingViewController
                presentViewController:viewController
                animated:YES completion:^{
                    userAuthenticated = YES;
                    //1
                    GKTurnBasedEventHandler *ev =
                        [GKTurnBasedEventHandler
                            sharedTurnBasedEventHandler];
                    ev.delegate = self;
                    //#[self deleteAllMatches];
                }];
        } else if (localPlayer.authenticated) {
            userAuthenticated = YES;
            //2
        }
    };
}
```

```

        GKTurnBasedEventHandler *ev =
            [GKTurnBasedEventHandler
                sharedTurnBasedEventHandler];
        ev.delegate = self;
        [self deleteAllMatches];
    } else {
        userAuthenticated = NO;
        NSLog(@"Error with Game Center %@", error);
    }
};

#else
    if (localPlayer.authenticated == NO) {
        [[GKLocalPlayer localPlayer]
        authenticateWithCompletionHandler:^(NSError *error) {
            GKTurnBasedEventHandler *ev = [GKTurnBasedEventHandler
sharedTurnBasedEventHandler];
            ev.delegate = self;
        }];
        // [self deleteAllMatches];
    } else {
        NSLog(@"Already authenticated");
        //4
        GKTurnBasedEventHandler *ev =
            [GKTurnBasedEventHandler sharedTurnBasedEventHandler];
        ev.delegate = self;
        // [self deleteAllMatches];
    }
}
#endif
}

```

In the sections marked 1, 2, 3, and 4 you're setting the `GCTurnBasedMatchHelper` object as the delegate of the `GKTurnBasedEventHandler`. You have to do it these four times because only one, but any one, of those four sections of code will be executed.

Alright, you're ready to declare that the `GCTurnBasedMatchHelper` object implements the `GKTurnBasedEventHandler`. Next, you'll implement that protocols methods.

In `GCTurnBasedMatchHelper.h`, change the `@interface` line to:

```

@interface GCTurnBasedMatchHelper : NSObject
    <GKTurnBasedMatchmakerViewControllerDelegate,
    GKTurnBasedEventHandlerDelegate> {

```

And add the following code to `GCTurnBasedMatchHelper.m`:

```

-(void)handleInviteFromGameCenter:(NSArray *)playersToInvite {

```

```

    NSLog(@"new invite");
}

-(void)handleTurnEventForMatch:(GKTurnBasedMatch *)match
didBecomeActive:(BOOL)becameActive {
    NSLog(@"Turn has happened");
}

-(void)handleMatchEnded:(GKTurnBasedMatch *)match {
    NSLog(@"Game has ended");
}

```

Build and run (on a device, not the simulator) now. You should be able to test the handleTurn event by letting the other player take a turn, and you should see the log message. Hip, hip, hooray!

```

All Output ▾
2012-10-12 10:18:34.789 spinningyarn[5784:907] Authenticating local user . .
Oct 12 10:18:35 fattitakes-hotspot spinningyarn[5784] <Info>: 10:18:35.951476 com.apple.AVConference: GKSConnSettings: set server: {
    "gk-cdx" = "17.173.254.218:4398";
    "gk-commnat-cohort" = "17.173.254.220:16386";
    "gk-commnat-main0" = "17.173.254.219:16384";
    "gk-commnat-main1" = "17.173.254.219:16385";
}
2012-10-12 10:18:51.767 spinningyarn[5784:907] Taking turn for existing game...
2012-10-12 10:18:51.983 spinningyarn[5784:907] Warning: Attempt to dismiss from view controller <ViewController: 0x1d060130> while a presentation or dismissal is in progress!
2012-10-12 10:18:51.984 spinningyarn[5784:907] Taking turn for existing game...
2012-10-12 10:18:53.355 spinningyarn[5784:907] did begin
2012-10-12 10:18:57.328 spinningyarn[5784:907] Send Turn, <4f6e6365 2875708f 6e206128 74696d65 28746573 7428796c 61796572 28332078
6c617965 72203228 61286228 62206328 61626328 7475726 28746573 74>, (
    "<GKTurnBasedParticipant 0x1d061fe0 - id:0:1426794362 status:Active outcome:None lastTurn:2012-10-12 10:09:06 +0000>",
    "<GKTurnBasedParticipant 0x1d061fe0 - id:0:2006794178 (local player) status:Active outcome:None lastTurn:2012-10-11 07:36:55 +0000>"
)
2012-10-12 10:18:57.324 spinningyarn[5784:907] did end
2012-10-12 10:28:11.673 spinningyarn[5784:907] Turn has happened

```

Again, note that you must be running a device receiving the event. The simulator won't receive these events.

The handleMatchEnded will require you to write a method that ends the game. The handleInviteFromGameCenter is only fired when you start a game from inside the game center app.

If you send an invite from game center, the callback needs to instantiate a new GKMatchRequest and either programmatically or with the view controller (GKTurnBasedMatchmakerViewController) set up the new match. Invites sent from within the app won't need to call this method.

Handling invitations

Let's write handleInviteFromGameCenter next. You may assume, as I did, that this method fires whenever you receive a named invite to a game. This is not what the method is for at all! There is no event for that, an invite sent from within a game just shows up in your list of available matches.

This method handles the incoming data from game center when you create an invite for one of your friends for that game. So, when you switch from game center

to the game, there's information about whom you want to invite to a new game included in the callback (`playersToInvite`). This is called on the inviting player's game, not the invitee. I include this personal mistake here because I'm not the only one who was confused.

If you get a new invite from game center, you need to instantiate the `GKTurnBasedMatchmakerViewController` with a `GKMatchRequest`. This method will give you an array of players that are supposed to be in the match. You'll use this object to set up the `GKMatchRequest`.

Inside **GCTurnBasedMatchHelper.m**, replace the `handleInviteFromGameCenter:` method with the following:

```
- (void)handleInviteFromGameCenter:(NSArray *)playersToInvite {
    [presentingViewController dismissViewControllerAnimated:YES
        completion:nil];
    GKMatchRequest *request = [[GKMatchRequest alloc] init];

    request.playersToInvite = playersToInvite;
    request.maxPlayers = 12;
    request.minPlayers = 2;

    GKTurnBasedMatchmakerViewController *viewController =
        [[GKTurnBasedMatchmakerViewController alloc]
            initWithMatchRequest:request];

    viewController.showExistingMatches = NO;
    viewController.turnBasedMatchmakerDelegate = self;
    [presentingViewController
        presentViewController:viewController animated:YES
        completion:nil];
}
```

The first thing you do is get rid of any current modal view controller that is present (it's possible that the app is currently looking at another instance of the `GKTurnBasedMatchmakerViewController`).

After that you set up the `GKMatchRequest` and then the `GKTurnBasedMatchmakerViewController`. Note that the `showExistingMatches` is set to NO. You only want to see the new game view for this match. You set up the delegate and then present your view controller.

Build and run now to see how this works. In order to test this callback do the following:

1. Go to the Game Center App
2. Pick the friends tab, choose a friend who has downloaded the game (the game must be listed in the friends view)

3. Select the game
4. Press 'Play' in the top right.

At that point, it will jump to the app, and then load the `GKTurnBasedMatchmakerViewController`, you'll notice that the selected friend is already in slot 2.



Handling the turn event

When `handleTurn` is called, either the match has moved from one player to another, and it's still not our turn, or the turn has moved to our player. In addition, the match that's currently loaded into the game state, may or may not be the match that has received the `handleTurn` call. You need to distinguish between these scenarios and appropriately deal with each.

In `GCTurnBasedMatchHelper.m`, replace `handleTurnEventForMatch:` with the following:

```
-(void)handleTurnEventForMatch:(GKTurnBasedMatch *)match {
    NSLog(@"Turn has happened");
    if ([match.matchID
        isEqualToString:self.currentMatch.matchID]) {
```

```
if ([match.currentParticipant.playerID  
isEqualToString:[GKLocalPlayer localPlayer].playerID]) {  
    // it's the current match and it's our turn now  
    self.currentMatch = match;  
    [self.delegate takeTurn:match];  
} else {  
    // it's the current match, but it's someone else's turn  
    self.currentMatch = match;  
    [self.delegate layoutMatch:match];  
}  
}  
} else {  
    if ([match.currentParticipant.playerID  
isEqualToString:[GKLocalPlayer localPlayer].playerID]) {  
        // it's not the current match and it's our turn now  
        [self.delegate sendNotice:  
            @"It's your turn for another match" forMatch:match];  
    } else {  
        // it's the not current match, and it's someone else's  
        // turn  
    }  
}  
}  
}
```

You end up with four scenarios. You'll send delegate methods for three of them, the fourth you'll ignore, but you've set it up here in case you wish to handle it in another game.

The first two are that the current match is the same as the match you're in. In that case you'll send `takeTurn`, if it's our turn, and `layoutMatch` if it's not. You're still setting the `currentMatch` to your passed in `match`, because even though the `matchID` is the same, the incoming match has updated state data (`match.matchData`) and the `currentParticipant` has changed.

If you are sent a notice for a match that you're not looking at, and it has become our turn, you'll send the `sendNotice` delegate method. You'll use this method to display an alert letting the user know.

If the turns change on matches that you're not looking at, and it's not our turn, you'll do nothing. The users can go looking at those matches by loading them using the `GKTurnBasedMatchmakerViewController`, you don't need to interrupt them every time things change on every match. In other kinds of games, you may want to do just that, so that section is available to you.

Build and run the game to try it out. Each time a new turn is sent, you should see the UI on your game update and the `statusLabel` should let you know whose turn it is!



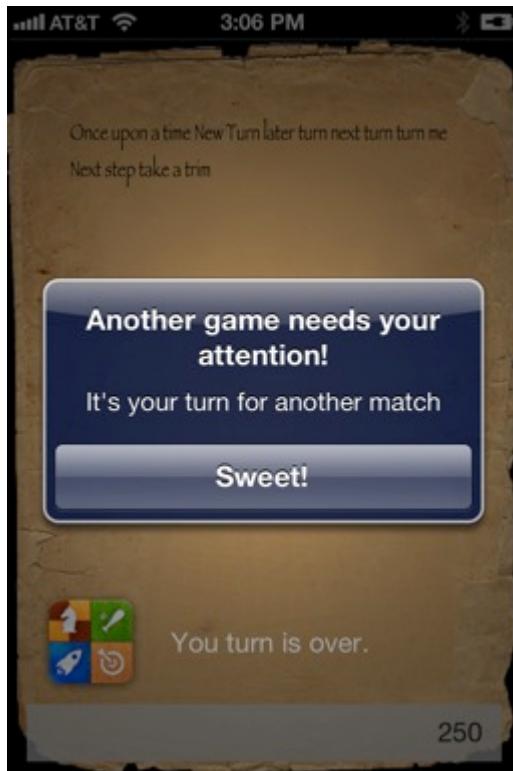
Finally, it's starting to look like a real game!

Depending on the state of your game, you might get an error saying the `sendNotice:forMatch:` callback isn't implemented yet. That's your next step anyway, so go ahead and implement it in **ViewController.m**:

```
- (void)sendNotice:(NSString *)notice forMatch:
(GKTurnBasedMatch *)match {
    UIAlertView *av = [[UIAlertView alloc] initWithTitle:
        @"Another game needs your attention!" message:notice
        delegate:self cancelButtonTitle:@"Sweet!"
        otherButtonTitles:nil];
    [av show];
}
```

This is pretty straightforward. You're just letting the player know that another match needs their attention. You'll let them use the `GKTurnBasedMatchmakerViewController` to load the match when they are ready. You should be able to get this alert if you are playing multiple games.

Build and run again, and this time everything should work great!



Game over, man, game over!

You've really only got one thing left to accomplish, and that's ending the game. But, you probably also want to give your users some advance notice that they only have a certain number of turns left.

You'll do this with a new method that checks the length of the `NSData`. You'll end the game when it gets to above 3800 characters, but let's start letting your players know when the game gets to about 3000 characters.

This new method will be called each time a match is loaded, so you'll put it in your `takeTurn` method and your `layoutMatch` method. If the match is getting close to being over, this method will add some information to your `statusLabel`, saying, there's only about 200 characters left.

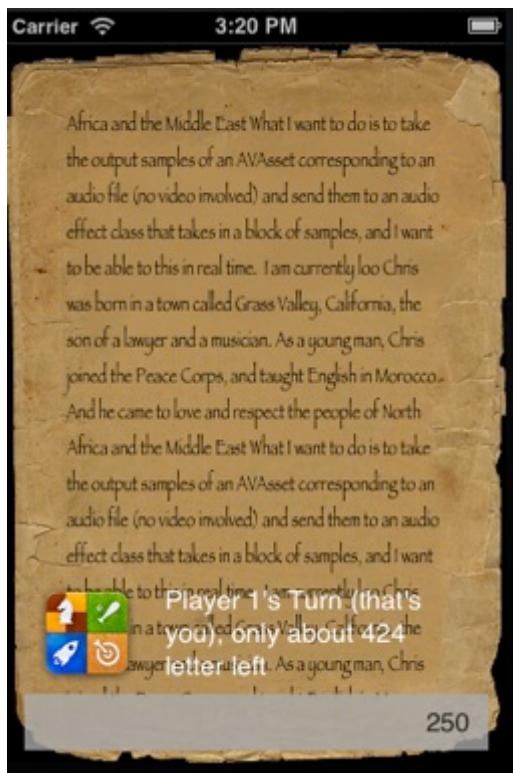
Add this method to `ViewController.m`:

```
- (void)checkForEnding:(NSData *)matchData {
    if ([matchData length]) {
        selfStatusLabel.text = [NSString stringWithFormat:
            @"%@, only about %d letter left",
            selfStatusLabel.text, 4000 - [matchData length]];
    }
}
```

Also call this method at the end of `layoutMatch` and `takeTurn`:

```
[self checkForEnding:match.matchData];
```

Go ahead and build and run, you should get something like this (after you have written a bunch of stuff):



What you'll do is end the game whenever the total character count exceeds 3800. In iOS 5, the max size of the `matchData` was 4k. In iOS 6 that limit has been increased to 64k. The game was built under the previous constraints, thus the 3800 character cutoff. If the `matchData` exceeds the maximum size, Game Center won't accept the end of the match.

Modify `sendTurn` one last time by replacing this block of code:

```
[currentMatch endTurnWithNextParticipants:nextParticipants
turnTimeout:600 matchData:data completionHandler:^(NSError *error)
{
    if (error) {
        NSLog(@"%@", error);
        //2
        selfStatusLabel.text = @"Oops, there was a problem.
Try that again.";
    } else {
        //3
        selfStatusLabel.text = @"Your turn is over.";
    }
}];
```

```

        self.textField.enabled = NO;
    }
}];

```

With this:

```

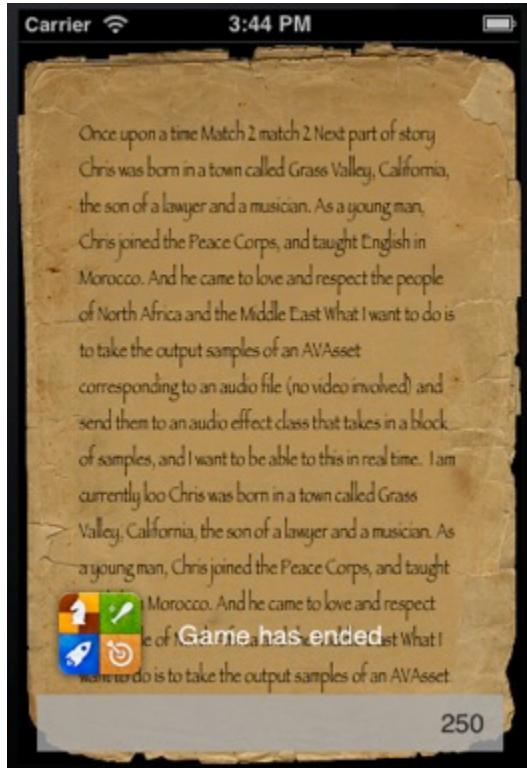
if ([data length] > 3800) {
    for (GKTurnBasedParticipant *part in
        currentMatch.participants) {
        part.matchOutcome = GKTurnBasedMatchOutcomeTied;
    }
    [currentMatch endMatchInTurnWithMatchData:data
        completionHandler:^(NSError *error) {
            if (error) {
                NSLog(@"%@", error);
            }
        }];
    selfStatusLabel.text = @"Game has ended";
} else {
    [currentMatch endTurnWithNextParticipants:nextParticipants
        turnTimeout:36000 matchData:data completionHandler:
        ^(NSError *error) {
            if (error) {
                NSLog(@"%@", error);
                selfStatusLabel.text = @"Oops, there was a problem.
Try that again.";
            } else {
                selfStatusLabel.text = @"Your turn is over.";
                self.textField.enabled = NO;
            }
        }];
}

```

You're just wrapping the current `endTurn` call in an `if` statement. If you are above 3800 characters, instead of calling `endTurn`, you'll call `endMatch`. Once you call `end game` that notification will be sent to all the other players in the `handleMatchEnded` notification.

You'll deal with that in a second, but let's give your end game a whirl. Make sure your `handleEndMatch` notice still has a log statement in it, and hopefully you've still got your last game around (didn't it take forever to write 3000 characters?).

You should be able to get this:



Alright, the end is in sight! Now you just need to replace the `handleMatchEnded` method in **GCTurnBasedMatchHelper.m** with the following:

```
-(void)handleMatchEnded:(GKTurnBasedMatch *)match {
    NSLog(@"Game has ended");
    if ([match.matchID isEqualToString:self.currentMatch.matchID])
    {
        [self.delegate recieveEndGame:match];
    } else {
        [self.delegate sendNotice:@"Another Game Ended!";
    }
}
```

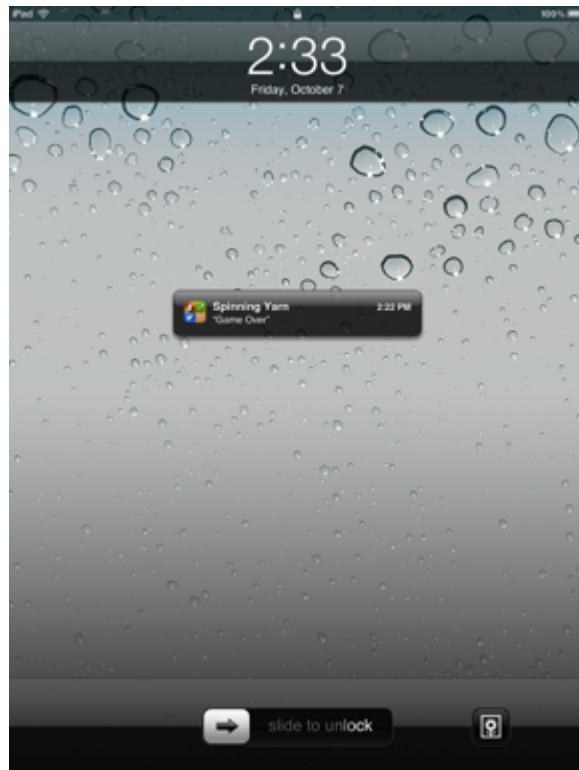
If you're looking at another match you'll just send the notice, that spawns the UIAlertView, otherwise you'll send `recieveEndGame`. Let's implement that as well in **ViewController.m**:

```
-(void)recieveEndGame:(GKTurnBasedMatch *)match {
    [self layoutMatch:match];
}
```

You could do something extra in this case, but because you're setting the `statusLabel` in `layoutMatch` this should suffice. However, should you decide later to

add something to an `endGame` notice, like email our story or invite the same group to play again, you could do that here.

Build and run now. Play through a game until you reach the end by reaching the 3800 character threshold. If you watch another device in the match, you should see a notice like this one:



Now you've covered the scenario when the length of the story reaches the max, but are there other ways to end a game? What if all the players, or all but one, quit playing? You might think, as I did, that Game Center would automatically handle this scenario and end the game. But, it doesn't.

That actually does make sense, because how would you, as the developer, control the outcome of each player at the end game state if Game Center automatically ended a game when the other player's quit? Game Center will allow you continue to play with only one player, so in this game, you'll need to check for this case.

One way to accomplish this is to use the

`turnBasedMatchmakerViewController:playerQuitForMatch:` callback. However, this is only called when the player quitting is the player that holds the current turn baton. If the player is waiting for another player to take their turn, and he quits, then Game Center will handle that scenario, without giving you a callback. In that case, you wouldn't have a way to intervene.

Note: There are several places that you could end a match, in the case that all but one player quits the game. Each has limitation.

The first is when the player quits when he holds the turn. In this case the best option is `turnBasedMatchmakerViewController:playerQuitForMatch:`. If the player quits, then you test for the number of participants still playing and appropriately end the game there. As noted above, the limitation is that this is only available when a player is inside his turn.

A second method would be to do this check on the `handleTurnEventForMatch:` method. This method is an event that is called whenever any player takes their turn. After a turn is taken all other players are notified. The limitation here is that this code block is only executed if the app is currently running in the foreground. If the app is backgrounded or isn't running at all, then the app receives a push notification, but the callback method isn't executed. Checking for the number of active players here would only be effective as long as the app was running in the foreground.

The method that I've chosen,

`turnBasedMatchmakerViewController:didFindMatch:` will always work, however, the UI experience is a little confusing. The player will see that there's a current match that's marked as 'Your Turn', but once they load the match, it will be examined and the match will be ended at that point. The UI then displays the match and informs the user that the match has ended.

It would be better if there was some way to determine that the match was ended outside of a player's turn, and display that in the UI before the match is loaded, but this appears to be a limitation of the current provided `GKTurnBasedMatchmakerViewController` object. You could potentially handle this better using a custom GUI. I cover custom GUI creation in the next chapter.

A better way is to check when `handleTurnEventForMatch:` is called. You want to see how many participants are still playing, and if it's less than two, end the match at that point.

You'll be adding this check to the `turnBasedMatchmakerViewController:handleTurnEventForMatch:` callback. Add this code at the top of the method:

```
//1
NSMutableArray *stillPlaying = [NSMutableArray array];
for (GKTurnBasedParticipant *p in match.participants) {
    if (p.matchOutcome == GKTurnBasedMatchOutcomeNone) {
        [stillPlaying addObject:p];
    }
}
//2
```

```
if ([stillPlaying count] < 2 && [match.participants count] >= 2) {
    //There's only one player left
    //3
    for (GKTurnBasedParticipant *part in stillPlaying) {
        part.matchOutcome = GKTurnBasedMatchOutcomeTied;
    }
    //4
    [match endMatchInTurnWithMatchData:match.matchData
completionHandler:^(NSError *error) {
    if (error) {
        NSLog(@"Error Ending Match %@", error);
    }
    //5
    [self.delegate layoutMatch:match];
}];
}
```

1. In this first section, you create an array of participants called `stillPlaying` and put all the participants who have an outcome of `GKTurnBasedMatchOutcomeNone` into it.
2. You want to trigger an end game when you have less than two participants left that are still playing. But, you don't want to trigger that too early (like when a new game has just started).
3. Before you send the call to end the game, you have to assign every participant an outcome that's not `GKTurnBasedMatchOutcomeNone`. In this case everyone gets tied, so everyone wins. It's like little league soccer!
4. You then send the call that ends the match.
5. In the completion block for the end turn method, you call `layoutMatch` in order to update the interface to reflect the new information.

Build and run one last time, and try the following test:

1. Start a new game on your device and take a turn
2. Wait a few moments, and start a new game on another device (or simulator) with auto-match to join the existing game
3. On the second device (or simulator) quit the game

You then should get a "game over" notice on your device like this:



And that marks the end of the game, and this chapter! ☺

Where To Go From Here?

I expect that we'll see lots of new games that use Apple's turn based API. There are other games that have this capability, but up until now this functionality had to be custom built on developer's own servers. With these new APIs and the handy integration with the Game Center app, it's now easier than ever to make turn based games.

The Turn Based Gaming API is a great tool for all kinds of strategy, card, board, and other games that are played asynchronously between multiple users, so I look forward to seeing what you come up with!

To learn more about Game Center's API, keep reading the next chapter, where I'll show you how you can customize the Turn Based Gaming UI with your own interface programmatically!

Chapter 19: Intermediate Turn Based Gaming

By Jacob Gundersen

In the previous chapter, you learned how to make a simple multiplayer turn-based game with Game Center, using the built-in `GKTurnBasedMatchmakerViewController` user interface.

Although using the built-in view controller is nice and easy, the turn-based gaming APIs allow you to create your own user interface as well. Making your own user interface can often make your game seem more professional, and it can allow you to display much more information about the various games the player is in and make your game more engaging.

In this tutorial, you are going to modify the SpinningYarn app to use a custom user interface for turn-based gaming. You'll modify the app so the user can see all the stories they're in the middle of at a glance, and take their turn where appropriate.

Provided Starter Project

You'll need to use the supplied starter project provided for this chapter. It's based on the code from the previous chapter, but I've made the following changes:

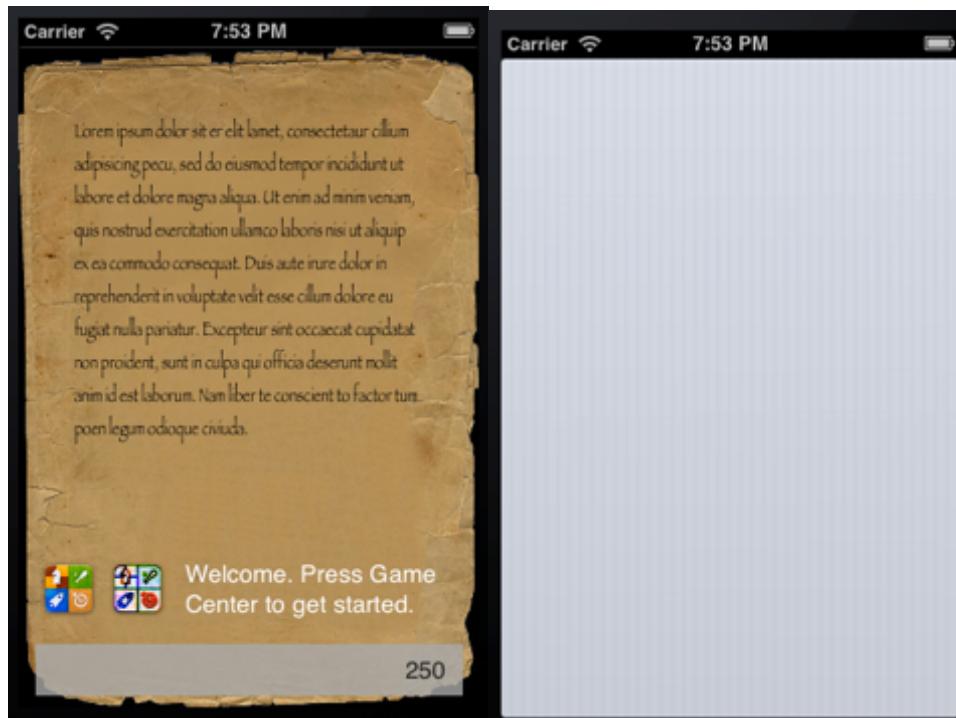
1. Added a second button for an 'alternative' Game Center interface
2. Added a new **UITableViewController** in **MainStoryboard.storyboard** (embedded in a **UINavigationController**)
3. Added a segue from the new button to a **UITableViewController**
4. Added a prototype cell with **UILabel**, **UIButton**s, and a **UITextView**
5. Added a **MatchCell** class with connections to Prototype cell objects
6. Added a few settings on the **UITableView** – set it to grouped style, no selection, and no separator

Note: If you are confused about how any of this is done, refer to Chapters 4 and 5, "Beginning and Intermediate Storyboards". That's where I learned how to do it! ☺

After getting familiar with the changes, don't forget to update your bundle identifier to the same one you set up in the last chapter.

Getting Started

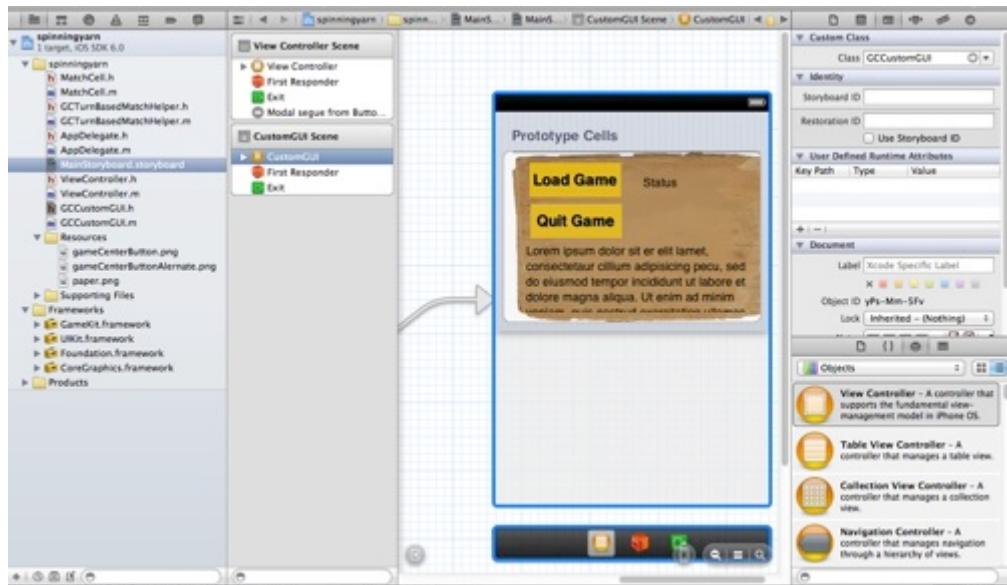
Go ahead and run the start project. It should look familiar, except you have a new button. Press that button, it should look like this:



A nice blank slate to start from!

The first thing you need to do is create a new table view controller which you'll be using to display the games you're currently in. So create a new file with the **iOS\Cocoa Touch\Objective-C class** template, enter **GCCustomGUI** for the class name, **UITableViewController** for subclass of and finish creating the file.

Open the **MainStoryboard.storyboard** file and set the **UITableViewController** to this new **GCCustomGUI** class in the identity inspector, like this:



Then open **CCCustomGUI.m**, and import Game Kit at the top of the file since you will be using that shortly:

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

Then add the following code to the end of `viewDidLoad`:

```
//1
self.tableView.rowHeight = 220;

//2
self.tableView.editing = NO;

//3
UIBarButtonItem *plus = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
    target:self action:@selector(addNewMatch)];
self.navigationItem.rightBarButtonItem = plus;

//4
UIBarButtonItem *cancel = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemCancel
    target:self action:@selector(cancel)];
self.navigationItem.leftBarButtonItem = cancel;
```

Let's go over this section by section.

1. The first thing you're doing is creating tall table view cells. You'll be displaying a bunch of buttons and info about your matches later on, so you want plenty of space for them.

2. Next you turn off the editing capability of the table view. You'll be laying out the table view manually by communicating with the Game Center server and reloading the table view, so don't want users to delete cells by swiping or anything.
3. Next you create two bar buttons for the navigation bar: a plus button (which you'll use to create a new match). It will call an `addNewMatch` method.
4. A cancel button (which you'll use to get back to the main screen). It calls a method called, predictably enough, `cancel`.

Next you need to add these methods so that this code will run. After `viewDidLoad`, add the following code:

```
- (void)cancel {
    [self.parentViewController dismissViewControllerAnimated:YES
                                                completion:nil];
}

-(void)addNewMatch {
    GKMatchRequest *request = [[GKMatchRequest alloc] init];

    request.maxPlayers = 12;
    request.minPlayers = 2;

    [GKTurnBasedMatch findMatchForRequest:request
        withCompletionHandler:^(GKTurnBasedMatch *match,
                               NSError *error) {
        if (error) {
            NSLog(@"%@", error.localizedDescription);
        } else {
            NSLog(@"match found!");
            // you'll load the match here
        }
    }];
}
```

The `cancel` method just dismisses your custom UI from the view.

The `addNewMatch` method might look a little familiar, because it's very close to what you did in the `GCTurnBasedMatchHelper`'s `findNewMatchWithMinPlayers:maxPlayers:viewController:` method.

Just like you did previously, you create a new `GKMatchRequest` and specify the number of players you want for the match. But then you do something different - instead of passing the `GKMatch` to the built-in `GKTurnBasedMatchmakerViewController` like you did previously, now you call `findMatchForRequest` to find a match programmatically.

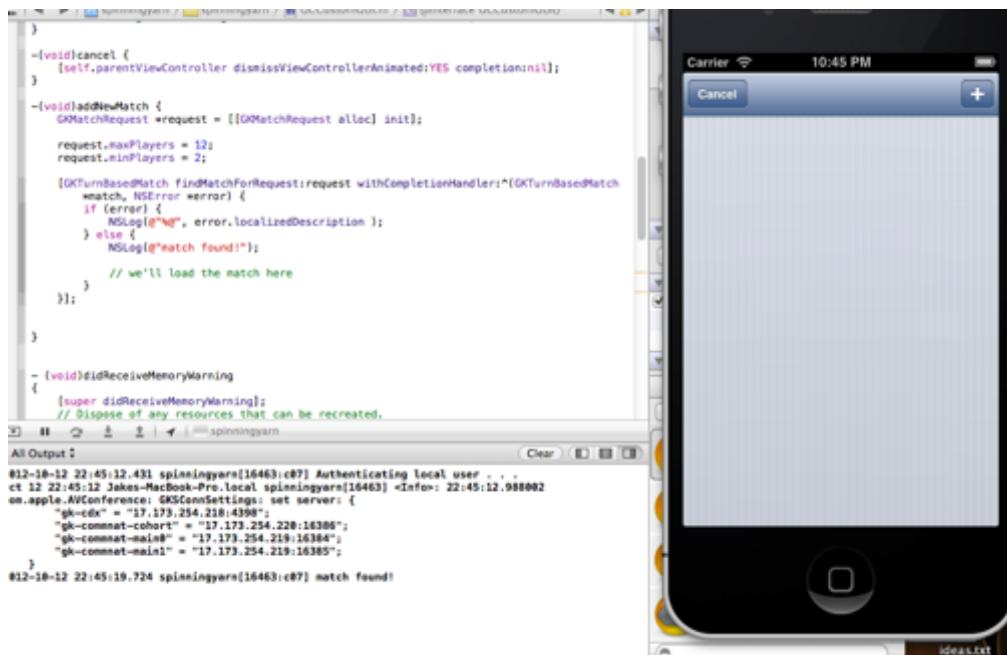
Note: This code will create a match with auto-matched player slots. If you wanted to create a match including invites to specific game center friends, you can do this by setting the `playersToInvite` (`NSArray`) property on the `GKMatchRequest` object.

You create the array by using player objects retrieved by using the provided Game Center methods to look up friends. Creating UI to view and select players for this purpose is beyond the scope of this chapter. If you would like to learn more about how to explore a player's friend list in Game Center, I recommend the official Game Center Programming Guide:

http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/Users/Users.html#/apple_ref/doc/uid/TP40008304-CH8-SW6

You then create a block that will be called when a match is found (or an error occurs). Either way, you just log out what happens for now, and later on you'll load the match upon success.

Let's try it out! Build and run the project, and you should be able to pull up the new navigation controller, dismiss it with the cancel button, and even tap the plus button to create a new match. Note that you won't see anything happen, but you should see a log message. Also, if you check in the `GKTurnBasedMatchmakerViewController` you should see a new match there, waiting for you to take your turn.



Loading the Matches

Now it's time for something fun! You want to load all the matches you're playing and display them in a new, custom UI with some custom table view cells.

Let's start by getting a list of all the matches you're playing. To do this, you're going to call a method named `loadMatchesWithCompletionHandler:` on `GKTurnBasedMatch`. This requests a list of every match your player is involved in, basically the same list of matches that is usually displayed by the built-in `GKTurnBasedMatchmakerViewController`.

Open up **GCCustomGUI.m**, and start by creating an array to keep track of your matches. Change the `@implementation` line to:

```
@implementation GCCustomGUI {
    NSArray *allMyMatches;
}
```

Next, add the code to load the list of matches at the bottom of `viewDidLoad`:

```
[self reloadTableView];
```

Here's the implementation of this method (add it right below `viewDidLoad`). You're breaking it out into a separate method because you'll call it from other objects (later on) in order to keep your GUI up to date:

```
-(void)reloadTableView {
    // 1
    [GKTurnBasedMatch loadMatchesWithCompletionHandler:
     ^(NSArray *matches, NSError *error) {
        // 2
        if (error) {
            NSLog(@"%@", error.localizedDescription);
        } else {
            // 3
            NSMutableArray *myMatches = [NSMutableArray array];
            NSMutableArray *otherMatches =
                [NSMutableArray array];
            NSMutableArray *endedMatches =
                [NSMutableArray array];
            // 4
            for (GKTurnBasedMatch *m in matches) {
                GKTurnBasedMatchOutcome myOutcome;
                for (GKTurnBasedParticipant *par in
                     m.participants) {
                    if ([par.playerID isEqualToString:
                         [GKLocalPlayer localPlayer].playerID]) {
```

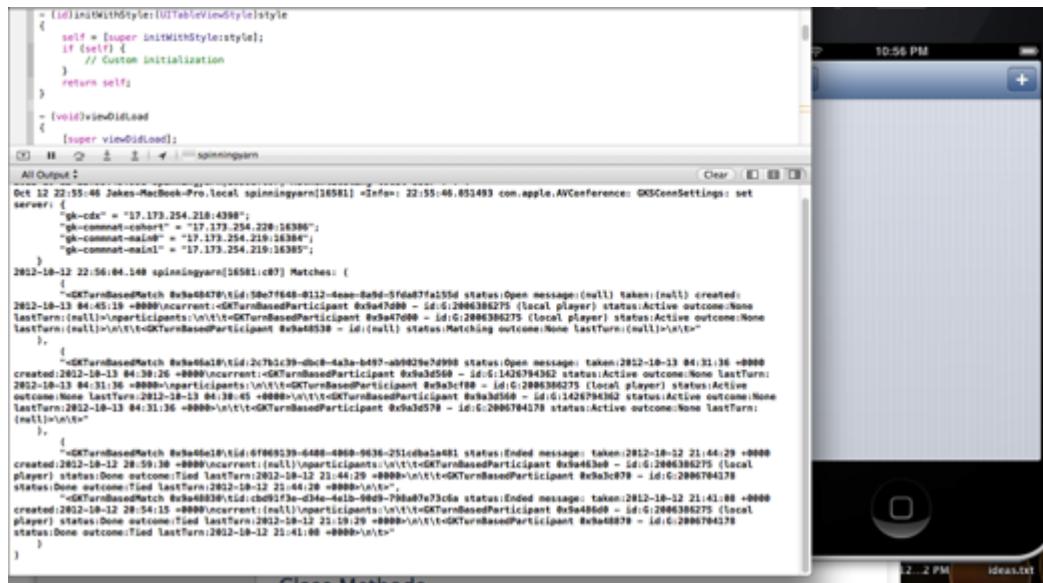
```
        myOutcome = par.matchOutcome;
    }
}
// 5
if (m.status != GKTurnBasedMatchStatusEnded &&
    myOutcome != GKTurnBasedMatchOutcomeQuit) {
    if ([m.currentParticipant.playerID
isEqualToString:[GKLocalPlayer localPlayer].playerID]) {
        [myMatches addObject:m];
    } else {
        [otherMatches addObject:m];
    }
} else {
    [endedMatches addObject:m];
}
}
// 6
allMyMatches = [[NSArray alloc]
    initWithObjects:myMatches,
    otherMatches,endedMatches, nil];
NSLog(@"Matches: %@", allMyMatches);
[self.tableView reloadData];
}
}];
}
```

There's a lot of code here, so let's go over it section by section according to the comments.

1. Here you call `loadMatchesWithCompletionHandler` to get all the matches your player is involved in, and have a big block of code that gets called when results arrive (or an error occurs).
2. When the request comes back, you first test if there was an error. If not, then you have a list of matches. If there was an error, log it.
3. You want to segment these into three types that will be displayed in three sections of your table view. The first is matches where it's currently your turn, the second is the group of matches where it's another player's turn, and the final group is that where the match has ended.
4. You then launch into a for loop where you iterate over each match. Inside the loop, you start by looking through all of the participants to find the entry for the current player. This way you know what the current player's `matchOutcome` is.
5. Next you can determine which list to put the match in based on whether the game has ended, and whether it's your turn or not.

6. When you're done you put all three into the `allMyMatches` array. You log out the results as well, so that you can follow along and make sure that you're getting the data and structure you think you're getting.

Build and run, and bring up the new table view controller. You still won't see anything in the table view, but you should see some data in your log about the matches you're involved in!



Setting up the Table View

Next you need to implement the table view data source methods so you can display this data to the screen.

Let's start by setting up your sections and their titles. Open up **GCCustomGUI.m** and implement `numberOfSectionsInTableView` and `titleForHeaderInSection` as follows:

```

- (NSInteger)numberOfSectionsInTableView:
    (UITableView *)tableView
{
    return 3;
}

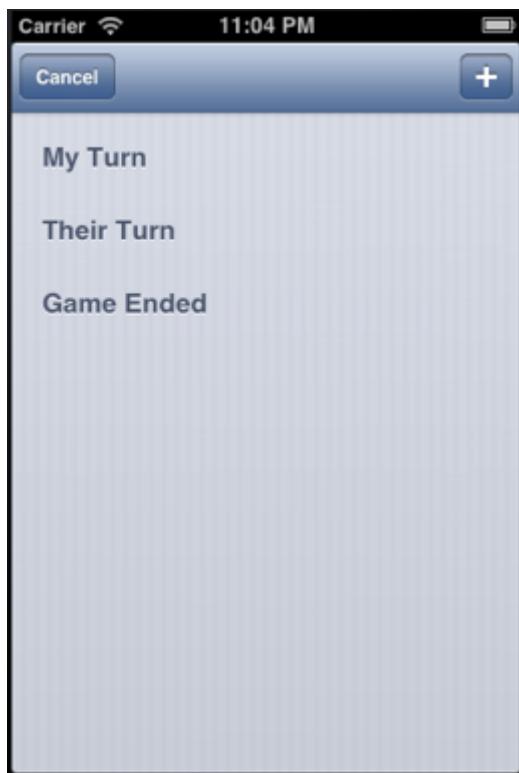
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    if (section == 0) {
        return @"My Turn";
    } else if (section == 1) {
        return @"Their Turn";
    } else {

```

```
        return @"Game Ended";
    }
}
```

The `numberOfSectionsInTableView` does exactly what it says. If you wanted to get more fancy, you could check for data in each type of array and count 'em up. But, you're going the simple route here. Three types of matches, three sections.

The `titleForHeaderInSection` returns an `NSString` based on the provided section integer. Because you have three sections, as returned in the previous method, you need three strings. This simple if else statement does the job fine.



Next, you'll make the table display some custom cells to display the match information in each row.

As I mentioned earlier, I've provided a prototype cell in `MainStoryboard.storyboard` that already has the class `MatchCell` and all the `IBOutlets` connected.

Note: If you'd like to do this part yourself, for good practice, here's a screenshot of the cell and the class. You can delete it and recreate it. Make sure that you set both the reuse identifier and the class if you decide to do so.



Note: This cell has two buttons connected to similarly named methods, load game and quit game. It also has a Label called `statusLabel` that will tell you status information about that match, it also has a `UITextView` that will contain an excerpt of the current story. I have connected an outlet to the `quitButton` because the text will change (If you have already quit that match and it's in a quit list.)

The background uses the same paper image that you are using for your main background it has a delegate protocol and property in order to be able to communicate with other view controllers (without going through the table view).

If you take a look at the files, you'll see they are pretty bare bones at this point. `MatchCell` is just a subclass of `UITableViewCell` (with no extra code added yet).

Open up **MatchCell.h** and import some headers you'll need at the top of the file:

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

```
#import "GCTurnBasedMatchHelper.h"
```

Then manually add a property for the match this cell will display as follows:

```
@property (retain, nonatomic) GKTurnBasedMatch * match;
```

Now that you've got some hooks into your cell, you can import the `MatchCell` class into your `GCCustomGUI` class and start using it! Open up `GCCustomGUI.m` and import `MatchCell.h` at the top of the file:

```
#import "MatchCell.h"
```

Then implement the table view methods as follows:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [[allMyMatches objectAtIndex:section] count];
}

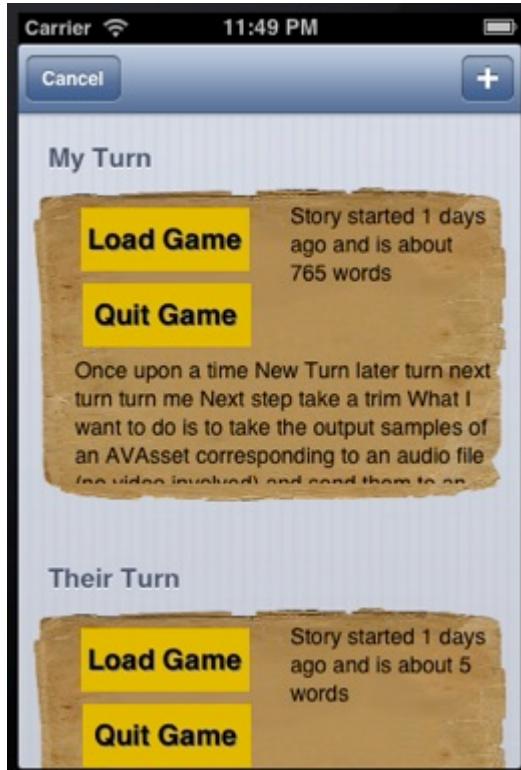
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    //1
    MatchCell *cell = (MatchCell *)[tableView
        dequeueReusableCellWithIdentifier:@"MatchCell"];
    //2
    GKTurnBasedMatch *match = [[allMyMatches
        objectAtIndex:indexPath.section]
        objectAtIndex:indexPath.row];
    cell.match = match;
    //3
    if ([match.matchData length] > 0) {
        //4
        NSString *storyString = [NSString
            stringWithUTF8String:[match.matchData bytes]];
        cell.storyText.text = storyString;
        //5
        int days = -floor([match.creationDate
            timeIntervalSinceNow] / (60 * 60 * 24));
        cell.statusLabel.text = [NSString
            stringWithFormat:@"Story started %d days ago and is
about %d words", days, [storyString length] / 5];
    }
    return cell;
}
```

The first method just tells the view how many cells to lay out for each section. It's easy, you just return the count for the appropriate array.

In `cellForRowAtIndexPath` you're doing a couple of things that aren't in the boilerplate code, so let's go over it section by section according to the comments.

1. First you call the `dequeueReusableCellWithIdentifier` to get a reference to a cell. This is either a new one, or it will be one that has scrolled off the top (or bottom) of the screen. You are passing in the identifier that is set in the Interface Builder so that it will load a `MatchCell` instead of standard cell. You also cast the object to a `MatchCell` so that you can set the attributes.
2. Here you get the match that corresponds to the section and row position for this cell (see how easy that is because your array structures match the cell `indexPath` structure!) Then you assign the match to the cell's `match` property. You're going to use this later, to quit matches and load matches.
3. Next you check if there is match data. If there's no match data (which can happen if a new match is created but the first player hasn't taken his turn yet), you don't display anything.
4. If there is match data, you set the text of the `storyText` text view to the string data in the `NSData` property of the match. `NSString` has the `stringWithUTF8String` method that will convert `NSData` (in UTF8 format), which is what you've got. You can then see the first few lines of each story, which gives you a much better sense for where you are in the story.
5. You also construct a `statusLabel` string. You want to tell the user how long it's been since the match was started. You use the `match.creationDate` and a method called `timeIntervalSinceNow`. That method tells you how many seconds have passed between a date value and now. You can convert that into days by dividing it by (60 seconds * 60 minutes * 24 hours in a day). The `floor` function cuts off any trailing decimals. You also get the length of the current `NSData` string and divide it by five to estimate how many words the story contains so far. Then you use this information to create a status string.

That's it! Compile and run, and you should have custom cells full of match information:



Your custom UI is finally starting to take shape!

Entering a New Match

Right now if you tap the + button in the toolbar, it creates a new match but doesn't let you actually play the match! So let's fix that now.

You need a way to dismiss the entire navigation controller from the table view controller. There are a number of ways to do this, but you're going to use an approach similar to what you did in your `GCTurnBasedMatchHelper` class. You're going to create an instance variable that will reference your `viewController` so you can invoke `dismissViewControllerAnimated` from it.

Open up `GCCustomGUI.h` and import some headers at the top of the file:

```
#import "ViewController.h"
#import "GCTurnBasedMatchHelper.h"
```

Also, add a property for the game's view controller:

```
@property (nonatomic, weak) ViewController * vc;
```

Next, find the `addNewMatch` method, and add the following code after the "you'll load the match here" comment:

```
[self_vc dismissViewControllerAnimated:YES completion:nil];
[ [GCTurnBasedMatchHelper sharedInstance]
    turnBasedMatchmakerViewController:nil didFindMatch:match];
```

Here's where that reference becomes useful. You first dismiss the `modelViewController` (which is the class you're in), and then you call `turnBasedMatchmakerViewController:didFindMatch` on the `GCTurnBasedMatchHelper`.

The reason you call this method is because it already contains the code to distinguish between different kinds of matches (brand new, our turn, other's turn, ended). Furthermore, you want to make sure the singleton stays up to date, because when you send a turn, it's going to ask the singleton for the `currentMatch` in order to send the next turn.

By pretending that you are just like the provided `GCTurnBasedMatchmakerViewController`, you save ourselves trying to replicate all the logic that you built up in the first part of this tutorial. You're just sending `nil` in for the `viewController` here because you are taking care of dismissing ourselves with the first line.

You need to set the `vc` property before this is going to work. Switch to **ViewController.m**. Because this view controller is currently being loaded by a segue, there isn't any code to alter yet. But, you can fix that by using the `prepareForSegue` callback and setting this property there. In order to do this, the first step is to import the `GCCustomGUI` class:

```
#import "GCCustomGUI.h"
```

Now add the `prepareForSegue` method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender {
//1
if ([segue.identifier isEqualToString:@"PresentCustomGUI"]) {
    //2
    UINavigationController *nc = (UINavigationController *)
        segue.destinationViewController;
    //3
    GCCustomGUI *cg = (GCCustomGUI *)nc.viewControllers[0];
    //4
    cg.vc = self;
}
}
```

1. The first thing to do with any `prepareForSegue` call is check which segue is happening. In this app, there's only the one, but it's good practice.

2. If you remember, the `GCCustomGUI` is embedded in a `UINavigationController`, so that's what the segue is actually presenting. You'll need to get the `GCCustomGUI` from the navigation controller.
3. A `UINavigationController` has a `viewControllers` array that contains all the child view controllers in the navigation stack. You know that there's only one, so it's the one at index zero. Here you're using the new subscripting syntax in iOS 6 to retrieve it. This is shorter and easier to read than a call to `objectAtIndex:0`.
4. Finally, you set the `vc` property of the `GCCustomGUI` instance.

There, you should now be able to instantiate a new game from the plus button on the nav bar. Give it a shot:



Loading a Game Programmatically

Next let's implement the code for your "Load Game" button in your table view cell, so you can load up an existing game.

To keep things clean, you are going to create a delegate protocol in your `MatchCell` so that you can communicate with the `GCCustomGUI`, which can then send messages to the `ViewController` class.

Open up `MatchCell.h` and declare the new protocol at the top of the file:

```
@protocol MatchCellDelegate
```

```
- (void)loadAMatch:(GKTurnBasedMatch *)match;
@end
```

Then, add a property for the delegate:

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

Now, switch to **MatchCell.m** and implement `loadGame`:

```
- (IBAction)loadGame:(id)sender {
    [self.delegate loadAMatch:self.match];
}
```

This method implementation is easy. The match that's associated with the cell is the match to send to the delegate method.

Next, you need to set the delegate when you create the `MatchCell`. Open up **GCCustomGUI.m** and mark the class as implementing `MatchCellDelegate` in the `@interface` section:

```
@interface GCCustomGUI () <MatchCellDelegate>
```

Finally, add the following inside `tableView:cellForRowAtIndexPath`, after the line `'cell.match = match'`:

```
cell.delegate = self;
```

The last step is to implement the delegate method as follows:

```
- (void)loadAMatch:(GKTurnBasedMatch *)match {
    [self.vc dismissViewControllerAnimated:YES completion:nil];
    [[GCTurnBasedMatchHelper sharedInstance]
        turnBasedMatchmakerViewController:nil didFindMatch:match];
}
```

Here you dismiss the view controller and notify the `GCTurnBasedMatchHelper` you found a match, like you did earlier when creating a new game.

Build and run now, and you should be able to load existing games from your custom UI!



Quitting Games

Before you dive into code, I should talk about the proper way to quit a game. First of all, there are two properties associated with the `GKTurnBasedParticipant` and one associated with a `GKTurnBasedMatch` that you might need to update when you quit a match.

The `GKTurnBasedParticipant` has a `status` property and an `outcome` property that may or may not be set by the method you use to quit. Also, if your match has all but one player who has quit, you need to change the `status` property of the `GKTurnBasedMatch`. You need to remember to do all of these things, as failure to do so may result in bugs and odd behavior.

There are several methods that you can use to quit from a match, the first two are `participantQuitInTurnWithOutcome:nextParticipants:turnTimeout:matchData:completionHandler:` and `participantQuitOutOfTurnWithOutcome:..`. Both of these methods will set the participant status to quit and the outcome to whatever you pass in as an argument to the method.

The other method available to you is the `removeWithCompletionHandler:` method. This method will remove you from the match, but it doesn't set any of these flags. This method should only be used to remove a match once you have already quit.

None of the methods set the match's status. If you are the second to last player to quit the match, you'll need to be responsible and end the match. So you'll need to do some checking in order to see if this is necessary.

If you quit when it's your turn, you can use the logic already in the `GCTurnBasedMatchHelper`, but if not your turn, you'll need to call a new method to deal with that.

Switch to **MatchCell.m** and implement `quitGame` as follows:

```
- (IBAction)quitGame:(id)sender {
    if ([self.match.currentParticipant.playerID
        isEqualToString:[GKLocalPlayer localPlayer].playerID]) {
        //1
        [[GCTurnBasedMatchHelper sharedInstance]
            turnBasedMatchmakerViewController:nil
            playerQuitForMatch:self.match];
    } else {
        //2
        [self.match
            participantQuitOutOfTurnWithOutcome:
                GKTurnBasedMatchOutcomeQuit
            withCompletionHandler:^(NSError *error) {
                if (error) {
                    NSLog(@"%@", error.localizedDescription);
                }
            }];
    }

    //3
    [self.delegate reloadTableView];
}
```

Let's go through each of the code sections one at a time.

1. The first marked method checks to see if you are quitting a match where it's your turn. If it is, you need to do some things (like pass on the baton to the next player), so you'll use the method you already created in the `GCTurnBasedMatchHelper`.
2. The second section fires if it's not our turn. In this case all you really need to do is call the `quit` method and give ourselves a match outcome. You're giving ourselves the `quit` outcome. The completion handler just prints out any error messages that may come back.
3. Finally, you need to update your GUI. That last method you call is the same that you built to load your GUI in the first place, I told you there was a reason to break it out into its own method.

You do need to add it to the `MatchCellDelegate` protocol, so modify it in `MatchCell.h` as follows:

```
@protocol MatchCellDelegate
-(void)loadAMatch:(GKTurnBasedMatch *)match;
-(void)reloadTableView;
@end
```

Build and run, and you should be able to now quit a match, and after a small pause, see that the GUI will update to reflect the quitting!



Fixing the Quit Button

The next thing you need to do is change the button for the matches that you've quit. You don't want to be quitting the same match multiple times, but you do want to be able to remove matches you've quit from so that you don't have a huge list of matches that you no longer care about.

The first thing to do is go back to your `cellForRowAtIndexPath` method in `GCCustomGUI.m`, and add the following right before the call to `return cell`:

```
if (indexPath.section == 2) {
    [cell.quitButton setTitle:@"Remove"
        forState:UIControlStateNormal];
```

```
}
```

If you're in this third section, then you know that all these games are matches that have ended. You want the button to change its title to 'Remove'.

Alright, you need to go back and deal with this button press. Switch to **MatchCell.m** and modify `quitGame` so it calls a different method if you're in the Remove state:

```
UIButton *send = (UIButton *)sender;
if ([send.titleLabel.text isEqualToString:@"Remove"]) {
    NSLog(@"remove, %@", send.titleLabel.text);
    [self.match removeWithCompletionHandler:^(NSError *error) {
        [self.delegate reloadTableView];
    }];
} else {
    // Put the rest of the code here
}
```

You're checking the button title to see if you need to remove or quit. If it's titled remove, then you can use the `removeWithCompletionHandler` method to get rid of the match. The remove method prevents the match from being retrieved when you call the `loadMatchesWithCompletionHandler:` method. The data still exists on Apple's servers, so other players still can retrieve it, but when this player calls for matches, it will no longer be in the retrieved list.

Everything that was previously in your `quitGame` method should be included in the else bracket. You want to do this first block or the block of code you were just looking at, not both. You're putting the reload method in the completion handler, because if it's called too early you'll still get the removed match back in your list.

If you build and run now you can remove all those old matches. You can also quit a match, then remove it.



Better Notifications

Let's go back and revisit the `sendNotice` function you set up earlier in `ViewController.m`. It would be nice to be able to do a couple of things. The first is that when other games are updated, your GUI should reload to reflect the most recent state. Secondly, you should add some buttons to the `UIAlertView` so that you can either dismiss it, load the match that just received and update, or load your GUI view.

The first thing you need is a way to check if your GUI is loaded or not. Switch to **GCCustomGUI.h** and add the following declarations:

```
- (BOOL)isVisible;  
-(void)reloadTableView;
```

You're going to need to call `reloadTableView` from outside this class. So, even though you don't need to change that method, you need to declare it in the header file.

Then switch to **GCCustomGUI.m** and implement it as the following:

```
- (BOOL)isVisible {  
    return [self isViewLoaded] && self.view.window;  
}
```

This code will return true if your GUI is currently the top view. The `isViewLoaded` method will return true if the view is loaded into memory. The `window` property will return if it's visible, but if you call this by itself, it will load the view into memory if it's already loaded. By using both you can avoid the extra overhead.

You're going to need a few new instance variables in order to keep track of the pieces of this system. The first is a reference to your `GCCustomGUI` class. Add the following property to the `@interface` section of **ViewController.m**:

```
@property (weak, nonatomic) GCCustomGUI *altGUI;
```

This will need to be set when the `GCCustomGUI` is loaded in the `prepareForSegue`. Add this code after the line '`cg.vc = self`':

```
self.altGUI = cg;
```

Next, you need a new match object in order to switch back and forth between multiple matches. Open **GCTurnBasedMatchHelper.h** and add this to the `@interface` section:

```
@property (strong) GKTurnBasedMatch *alternateMatch;
```

This object is going to keep track of the match that you'll be able to switch to when you press the 'Let's see it' button.

Now lets update the `sendNotice` method in **ViewController.m** to check for the GUI and respond accordingly:

```
- (void)sendNotice:(NSString *)notice
forMatch:(GKTurnBasedMatch *)match {
    if ([self.altGUI isVisible]) {
        [self.altGUI reloadTableView];
    } else {
        [GCTurnBasedMatchHelper sharedInstance].alternateMatch =
        match;
        UIAlertView *av = [[UIAlertView alloc]
initWithTitle:@"Another game needs your attention!" message:notice
delegate:self cancelButtonTitle:@"Sweet!"
otherButtonTitles:@"Let's see it!", @"All my Matches", nil];
        av.delegate = self;
        [av show];
    }
}
```

The first part you just check if the `gccustomGUI` is currently visible. If it is, then you just reload the table data and update the view.

If the GUI isn't in view, then you set the `alternateMatch` object. You may need to load this match later, so this keeps track of it.

Then you set up a new `UIAlertView` with a couple buttons. You will have three buttons. 'Sweet' will just dismiss the alert. 'Let's see it!' will load the match that sent the notification. 'All my Matches' will load the Custom GUI.

Finally, you set the class as the delegate of the `UIAlertView`.

You'll need to set the `ViewController` class to implement the `UIAlertViewDelegate` protocol in order to implement the method to handle the button presses. Add this code to the `@interface`:

```
@interface ViewController() <UITextFieldDelegate,
GCTurnBasedMatchHelperDelegate, UIAlertViewDelegate>
```

The protocol has one required method, so add its implementation to **ViewController.m**:

```
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    switch (buttonIndex) {
        case 1:
```

```
[ [GCTurnBasedMatchHelper sharedInstance]
setCurrentMatch:[GCTurnBasedMatchHelper
sharedInstance].alternateMatch];

        if ([[GCTurnBasedMatchHelper
sharedInstance].currentMatch.currentParticipant.playerID
isEqualToString:[GKLocalPlayer localPlayer].playerID]) {

            [self takeTurn:[GCTurnBasedMatchHelper
sharedInstance].currentMatch];
        } else {
            [self layoutMatch:[GCTurnBasedMatchHelper
sharedInstance].currentMatch];
        }
        break;
    case 2:
        [self performSegueWithIdentifier:@"PresentCustomGUI"
sender:nil];
        break;
    default:
        break;
}
}
```

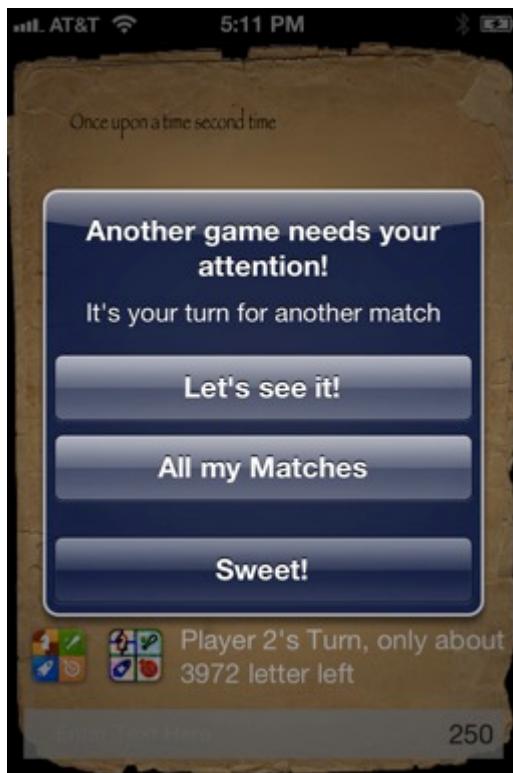
This method gets passed the `buttonIndex` value which in your case is a value of 0 if the “Sweet!” (Cancel) is pressed, a value of 1 if the “Let’s See It!” is pressed, and 2 if the “All my Matches” is pressed.

You handle the cancel button the same as you did before, it simply dismisses the alertview. You don’t have do handle it in the switch.

If you decide you want to load the notified match, you’ll first set the current match to the alternate match. Then you pass the appropriate method to `ViewController` object. If it’s your turn you use the `takeTurn` method, if not you use `layoutMatch`.

If the user pressed “All my Matches”, you call the `loadGCCustomGUI` method, this will act just like when you push your button.

Build and run now, play a few turns, you’ll be given options to load your games:



Sweet, you now have a fully functional turn-based game with a custom interface!

Where To Go From Here?

As you can see, the Turn Based Gaming API is a great tool for developers of turn based games. It will both help ease the development of the baton passing and help market your game.

Your users will be able to invite their Game Center friends to play your turn based game even if their friends don't have your game yet. An invite to a turn based game will prompt the user to install the app. This is a great way to market your new game to a built in social network of game players. That along with all the other great Game Center features should promote the use of this new API. That means more users for your turn based game.

You might also enjoy reading Chapter 28 in this book, "New Game Center APIs". It shows you how to import profile pictures and send banner notifications within Game Center.

I have only covered the basics of the API for the purpose of creating a custom UI within game center. By using this API, and the other Game Center APIs, you can create a very rich, social experience. You could use the 4K data, or creating your own game center server to store additional data, to include in game messaging,

player rankings, or any other kind of game information to make the game center experience compelling for your users.

To learn more about the Game Center's API, check out the Apple Documentation, the WWDC videos from 2011, or raywenderlich.com. Send me a line if you have any questions or would like to show off your next great Turn Based Game!

Chapter 20: Beginning Core Image

By Jacob Gundersen

Core Image is a powerful framework that lets you easily apply filters to images, such as modifying the vibrance, hue, or exposure. It uses the GPU (or CPU, user definable) to process the image data and is very fast. Fast enough to do real time processing of video frames!

Core Image filters can stacked together to apply multiple effects to an image or video frame at once. When multiple filters are stacked together they are efficient because they create a modified single filter that is applied to the image, instead of processing the image through each filter, one at a time.

Each filter has its own parameters and can be queried in code to provide information about the filter, its purpose, and input parameters. The system can also be queried to find out what filters are available. At this time, only a subset of the Core Image filters available on the Mac are available on iOS. However, as more become available the API can be used to discover the new filter attributes.

In this tutorial, you will get hands-on experience playing around with Core Image. You'll apply a few different filters, and you'll see how easy it is to apply cool effects to images in real time!

Core Image Overview

Before you get started, let's discuss some of the most important classes in the Core Image framework:

1. **CIContext**. All of the processing of a core image is done in a CIContext. This is somewhat similar to a Core Graphics or OpenGL context.
2. **CIImage**. This class holds the image data. It can be created from a UIImage, an image file, or pixel data.
3. **CIFilter**. The filter class has a dictionary that defines the attributes of the particular filter that it represents. Examples of filters are vibrance filters, color inversion filters, cropping filters, and much more.

You'll be using each of these classes as you create your project.

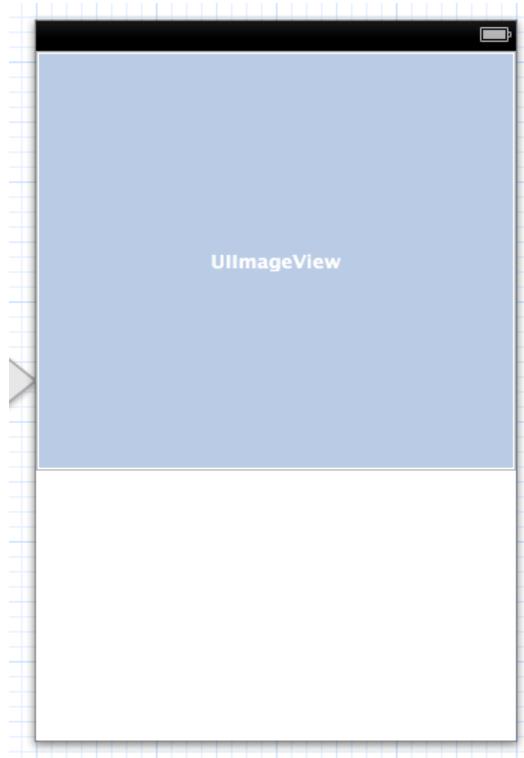
Getting Started

Open up Xcode and create a new project with the **iOS\Application\Single View Application** template. Enter **CoreImageFun** for the Product Name, select **iPhone** for the device family, and make sure that **Use Storyboards** and **Use Automatic Reference Counting** are checked (but leave the other checkboxes unchecked).

First things first, let's add the Core Image framework. On the Mac this is part of the QuartzCore framework, but on iOS it's a standalone framework. Go to the project container in the file view on the left hand side and select the **CoreImageFun** target. Choose the **Build Phases** tab, expand the **Link Binaries with Library** group and press the **+**. Navigate to the **CoreImage** framework and double-click on it.

Second, in the resources for this chapter, add the included **image.png** to your project. Done with setup!

Next open **MainStoryboard.storyboard**, drag an image view into the view controller, and set its mode to Aspect Fit. The position and dimensions should roughly match the following image:



Also, open the **Assistant Editor**, make sure it's displaying **ViewController.h**, and control-drag from the **UIImageView** to below the `@interface`. Set the Connection to Outlet, name it `imageView`, and click **Connect**.

Build and run just to make sure everything is good so far – you should just see an empty screen. The initial setup is complete – now on to Core Image!

Basic Image Filtering

You're going to get started by simply running your image through a `CIFilter` and displaying it on the screen.

Every time you want to apply a `CIFilter` to an image you need to do four things:

1. **Create a `CIImage` object.** `CIImage` has the following initialization methods:
`imageWithURL:`, `imageWithData:`, `imageWithCVPixelBuffer:`, and
`imageWithBitmapData:bytesPerRow:size:format:colorSpace:`. You'll most likely be working with `imageWithURL:` most of the time.
2. **Create a `CIContext`.** A `CIContext` can be CPU or GPU based. A `CIContext` can (and should) be reused, so you shouldn't create it over and over, but you will always need one when outputting the `CIImage` object.
3. **Create a `CIFilter`.** When you create the filter, you configure a number of properties on it that depend on the filter you're using.
4. **Get the filter output.** The filter gives you an output image as a `CIImage` – you can convert this to a `UIImage` using the `CIContext`, as you'll see below.

Let's see how this works. Add the following code to **ViewController.m** inside `viewDidLoad`:

```
// 1
NSString *filePath = [[NSBundle mainBundle]
    pathForResource:@"image" ofType:@"png"];
NSURL *fileNameAndPath = [NSURL fileURLWithPath:filePath];

// 2
CIImage *beginImage =
    [CIImage imageWithContentsOfURL:fileNameAndPath];

// 3
CIFilter *filter = [CIFilter filterWithName:@"CISepiaTone"
    keysAndValues: kCIInputImageKey, beginImage,
    @"inputIntensity", @0.8, nil];
CIImage *outputImage = [filter outputImage];

// 4
UIImage *newImage = [UIImage imageWithCIImage:outputImage];
```

```
self.imageView.image = newImage;
```

Let's go over this section by section:

1. The first two lines create an `NSURL` object that holds the path to your image file.
2. Next you create your `CIImage` with the `imageWithContentsOfURL` method.
3. Next you'll create your `CIFilter` object. A `CIFilter` constructor takes the name of the filter, and a dictionary that specifies the keys and values for that filter. Each filter will have its own unique keys and set of valid values.

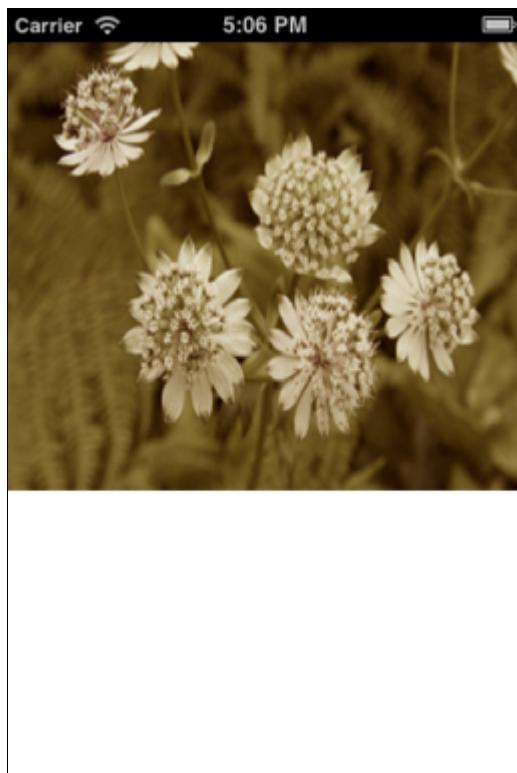
The `cisept Tone` filter takes only two values, the `kCIInputImageKey` (a `CIImage`) and the `@"inputIntensity"`, a float value, wrapped in an `NSNumber` (using the new literal syntax), between 0 and 1. Here you give that value 0.8.

Most of the filters have default values that will be used if no values are supplied. One exception is the `CIImage`, this must be provided as there is no default.

Getting a `CIImage` back out of a filter is easy. You just use the `outputImage` property.

4. Once you have an output `CIImage`, you will need to convert it into a `UIImage`. New in iOS 6 is the `UIImage` method `imageWithCIImage`. This method creates a `UIImage` from a `CIImage`. Once you've converted it to a `UIImage`, you just display it in the image view you added earlier.

Build and run the project, and you'll see your image filtered by the sepia tone filter:



Congratulations, you have successfully used `CIImage` and `CIFilters`!

Putting It Into Context

Before you move forward, there's an optimization that you should know about.

I mentioned earlier that you need a `CIContext` in order to perform a `CIFilter`, yet there's no mention of this object in the above example. It turns out that the `UIImage` method you called (`imageWithCIImage:`) does all the work for you. It creates a `CIContext` and uses it to perform the work of filtering the image. This makes using the Core Image API very easy.

But with ease comes a downside. There is one major drawback – it creates a new `CIContext` every time it's used. `CIContexts` take some resources to create, and can be reused in order to avoid that resource requirement every time you use it. If you want to use a slider to update the filter value, like you'll be doing in this tutorial, creating new `CIContexts` each time you change the filter would be way too slow.

Let's do this properly. Delete the code you added to `viewDidLoad` and replace it with the following:

```
NSString *filePath = [[NSBundle mainBundle]
    pathForResource:@"image" ofType:@"png"];
NSURL *fileNameAndPath = [NSURL fileURLWithPath:filePath];

CIImage *beginImage =
[CIImage imageWithContentsOfURL:fileNameAndPath];

// 1
CIContext *context = [CIContext contextWithOptions:nil];

CIFilter *filter = [CIFilter filterWithName:@"CISepiaTone"
    keysAndValues: kCIInputImageKey, beginImage,
    @"inputIntensity", @0.8, nil];
CIImage *outputImage = [filter outputImage];

// 2
CGImageRef cgimg =
[context createCGImage:outputImage fromRect:
    [outputImage extent]];

// 3
UIImage *newImage = [UIImage imageWithCGImage:cgimg];
self.imageView.image = newImage;

// 4
```

```
CGImageRelease(cgimg);
```

Again, let's go over this section by section.

1. Here you set up the `CIContext` object. The `CIContext` constructor takes an `NSDictionary` that specifies options including the color format and whether the context should run on the CPU or GPU. For this app, the default values are fine and so you pass in `nil` for that argument.
2. Here you use a method on the context object to draw a `CGImage`. Calling the `createCGImage:fromRect:` on the context with the supplied `CIFilter` will produce a `CGImageRef`.
3. Next, you use `UIImage +imageWithCGImage` to create a `UIImage` from the `CGImage`.
4. Finally, release the `CGImageRef`. `CGImage` is a C API that requires that you do your own memory management, even with ARC.

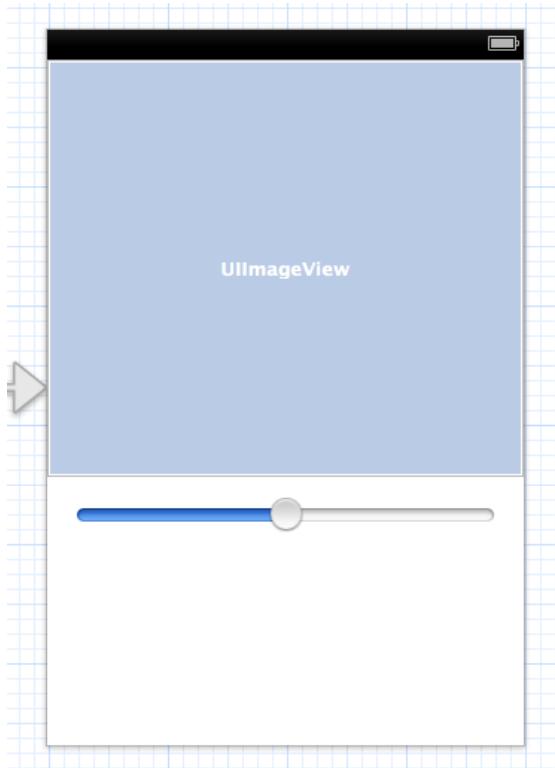
Build and run, and make sure it works just as before.

In this example, adding the `CIContext` creation and handling that yourself doesn't make too much difference. But in the next section, you'll see why this is important performance, as you implement the ability to change the filter dynamically!

Changing Filter Values

This is great, but this is just the beginning of what you can do with Core Image filters. Let's add a slider and set it up so you can adjust the image settings in real time.

Open **MainStoryboard.storyboard** and drag a slider in below the image view like so:



Make sure the Assistant Editor is visible and displaying **ViewController.h**, then control-drag from the slider down below the `@interface`. Set the Connection to **Action**, the name to **amountSliderValueChanged**, make sure that the Event is set to **Value Changed**, and click Connect.

While you're at it let's connect the slider to an outlet as well. Again control-drag from the slider down below the `@interface`, but this time set the Connection to **Outlet**, the name to **amountSlider**, and click Connect.

Every time the slider changes, you need to redo the image filter with a different value. However, you don't want to redo the whole process, which would be very inefficient and would take too long. You'll need to change a few things in your class so that you hold on to some of the objects you create in your `viewDidLoad` method.

The biggest thing you want to do is reuse the `CIContext` whenever you need to use it. If you recreate it each time, your program will run very slow. The other things you can hold onto are the `CIFilter` and the `CIImage` that holds your beginning image. You'll need a new `CIImage` for every output, but the image you start with will stay constant.

You need to add some instance variables to accomplish this task.

Add the following three instance variables to your private `@implementation` in **ViewController.m**:

```
@implementation ViewController {
    CIContext *context;
```

```
CIFilter *filter;
CIImage *beginImage;
}
```

Also, change the variables in your `viewDidLoad` method so they use the instance variables instead of declaring new local variables:

```
beginImage = [CIImage imageWithContentsOfURL:fileNameAndPath];
context = [CIContext contextWithOptions:nil];

filter = [CIFilter filterWithName:@"CISepiaTone"
    keysAndValues:kCIInputImageKey, beginImage, @"inputIntensity",
    @0.8, nil];
```

Now you'll implement the `amountSliderValueChanged:` method. What you'll be doing in this method is changing the value of the `@"inputIntensity"` key in your `CIFilter` dictionary. Once you've altered this value you'll need to repeat a few steps:

1. Get the output `CIImage` from the `CIFilter`.
2. Convert the `CIImage` to a `CGImageRef`.
3. Convert the `CGImageRef` to a `UIImage`, and display it in the image view.

So replace the `amountSliderValueChanged:` method with the following:

```
- (IBAction)amountSliderValueChanged:(UISlider *)slider {
    float slideValue = slider.value;

    [filter setValue:@(slideValue) forKey:@"inputIntensity"];
    CIImage *outputImage = [filter outputImage];

    CGImageRef cgimg = [context createCGImage:outputImage
        fromRect:[outputImage extent]];

    UIImage *newImage = [UIImage imageWithCGImage:cgimg];
    self.imageView.image = newImage;

    CGImageRelease(cgimg);
}
```

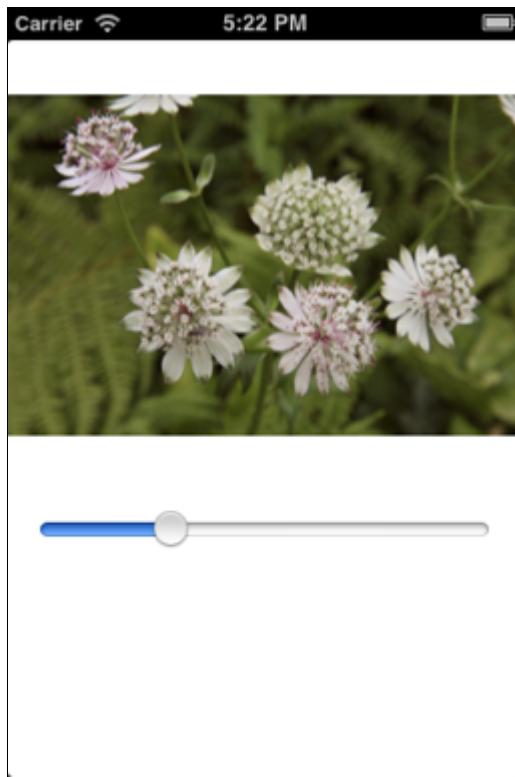
You'll notice that you've changed the variable type from `(id)sender` to `(UISlider *)sender` in the method definition. You know you'll only be using this method to retrieve values from your `UISlider`, so you can go ahead and make this change. If you'd left it as `(id)`, you'd need to cast it to a `UISlider` or the next line would throw an error. Make sure that the method declaration in the header file matches the changes you've made here.

You retrieve the float value from the slider. Your slider is set to the default values – min 0, max 1, default 0.5. These happen to be the right values for this `CIFilter`, how convenient!

The `CIFilter` has methods that will allow you to set the values for the different keys in its dictionary. Here, you're just setting the `@“inputIntensity”` to an `NSNumber` object with a float value of whatever you get from your slider.

The rest of the code should look familiar, as it follows the same logic as your `viewDidLoad` method. You're going to be using this code over and over again. From now on, you'll use the `changeSlider` method to render the output of a `CIFilter` to your `UIImageView`.

Build and run, and you should have a functioning live slider that will alter the sepia value for your image in real time!

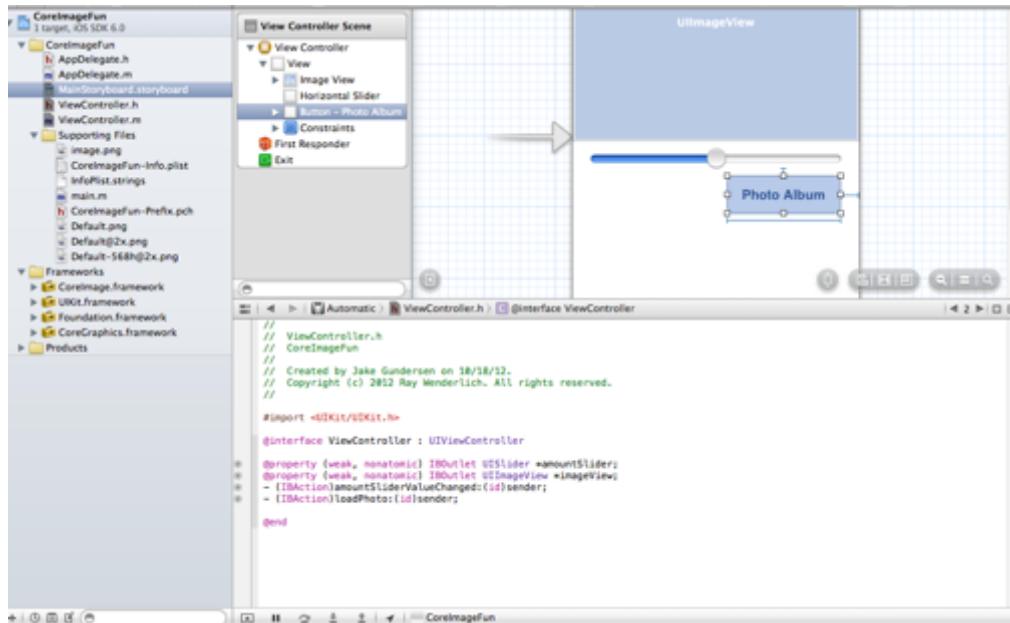


Getting Photos from the Photo Album

Now that you can change the values of the filter on the fly, things are starting to get real interesting! But what if you don't care for this image of flowers? Let's set up a `UIImagePickerController` so you can get pictures from out of the photo album and into your program so you can play with them.

You need to create a button that will bring up the photo album view, so open up **MainStoryboard.storyboard** and drag in a button to the bottom right of the slider and label it "Photo Album".

Then make sure the Assistant Editor is visible and displaying **ViewController.h**, then control-drag from the button down below the `@interface`. Set the Connection to **Action**, the name to `loadPhoto`, make sure that the Event is set to **Touch Up Inside**, and click **Connect**.



Next switch to **ViewController.m**, and implement the `loadPhoto:` method as follows:

```
- (IBAction)loadPhoto:(id)sender {
    UIImagePickerController *pickerC =
        [[UIImagePickerController alloc] init];
    pickerC.delegate = self;
    [self presentViewController:pickerC animated:YES
        completion:nil];
}
```

The first line of code instantiates a new `UIImagePickerController`. You then set the delegate of the image picker to `self` (your `ViewController`).

You get a warning here. You need to setup your `ViewController` as an `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate` and then implement the methods in that delegates protocol.

Still in **ViewController.m**, change the class extension as follows:

```
@interface ViewController () <UIImagePickerControllerDelegate,
    UINavigationControllerDelegate>
```

Now implement the following two methods:

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info {
    [self dismissViewControllerAnimated:YES completion:nil];
    NSLog(@"%@", info);
}

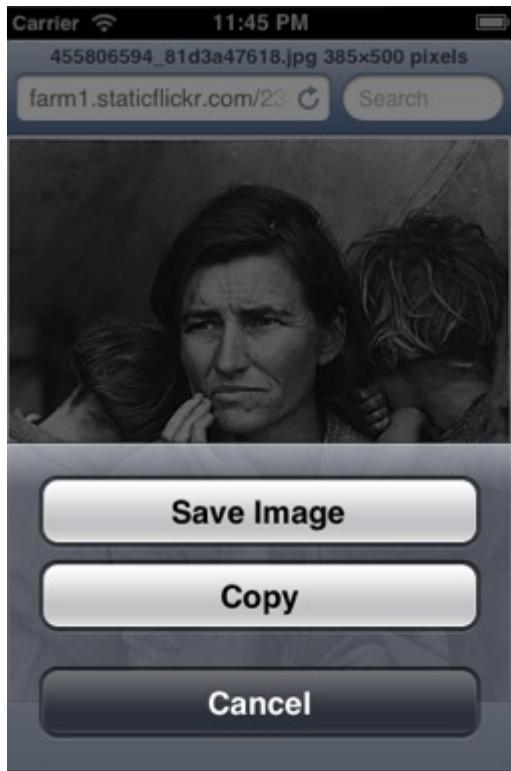
- (void)imagePickerControllerDidCancel:
(UINavigationController *)picker {
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

In both cases, you dismiss the `UIImagePickerController`. That's the delegate's job, if you don't do it there, then you just stare at the image picker forever!

The first method isn't completed yet – it's just a placeholder to log out some information about chosen image. The cancel method just gets rid of the picker controller, and is fine as-is.

Build and run and tap the button, and it will bring up the image picker with the photos in your photo album.

Note: If you are running this in the simulator, you probably won't have any photos in your photo album. However, you can use Safari to save images to your photo album. Open Safari, find an image, tap and hold, and you'll get a dialog. Choose to save that image. Next time you run your app, you'll have it!



Here's what you should see in the console after you've selected an image (something like this):

```
All Output :  
2012-10-18 23:43:44.782 CoreImageFun[12758:c07] {  
    UIImagePickerControllerMediaType = "public.image";  
    UIImagePickerControllerOriginalImage = "<UIImage: 0x71afc00>";  
    UIImagePickerControllerReferenceURL = "assets-library://asset/asset.JPG?id=A34B36D6-CE84-419A-9A68-FA95C93CC991&ext=JPG";  
}
```

Note that it has an entry in the dictionary for the “original image” selected by the user. This is what you want to pull out and filter!

Now that you've got a way to select an image, how do you set your `CIImage beganImage` to use that image?

Simple, just change the delegate method to look like this:

```
- (void)imagePickerController:(UIImagePickerController *)picker  
didFinishPickingMediaWithInfo:(NSDictionary *)info {  
    [self dismissViewControllerAnimated:YES completion:nil];  
    UIImage *gotImage =  
        [info objectForKey:UIImagePickerControllerOriginalImage];  
    beginImage = [CIImage imageWithCGImage:gotImage.CGImage];  
    [filter setValue:beginImage forKey:kCIInputImageKey];  
    [self amountSliderValueChanged:self.amountSlider];
```

```
}
```

You need to create a new `CIImage` from your selected photo. You can get the `UIImage` representation of the photo by finding it in the dictionary of values, under the `UIImagePickerControllerOriginalImage` key constant.

Note: It's better to use a constant rather than a hardcoded string, because Apple could change the name of the key in the future. For a full list of key constants, see Apple's `UIImagePickerController Delegate Protocol Reference` at:

http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIImagePickerControllerDelegate_Protocol/UIImagePickerControllerDelegate/UIImagePickerControllerDelegate.html

You need to convert this into a `CIImage`, but you don't have a method to convert a `UIImage` into a `CIImage`. However, you do have `[CIImage imageWithCGImage:]` method. You can get a `CIImage` from your `UIImage` by calling `UIImage.CGImage`, so you do exactly that!

You then set the key in the filter dictionary so that the input image is your new `CIImage` you just created.

The last line may seem odd. Remember how I pointed out that the code in the `changeValue` ran the filter with the latest value and updated the image view with the result?

Well you need to do that again, so you can just call the `changeValue` method. Even though the slider value hasn't changed, you can still use that method's code to get the job done. You could break that code into its own method, and if you were going to be working with more complexity, you would want to avoid confusion. But, in this case your purpose here is served using the `changeValue` method. You pass in the `amountSlider` so that it has the correct value to use.

Build and run, and now you'll be able to update the image from your photo album!



What if you create the perfect sepia image, how do you hold on to it? You could take a screenshot, but you're not that ghetto! Let's learn how to save your photos back to the photo album.

Saving to Photo Album

To save to the photo album, you need to use the **AssetsLibrary** framework. To add it to your project, go to the project container, select the **CoreImageFun** target, choose the **Build Phases** tab, expand the **Link Binaries with Libraries** group and click the **+** button. Find the **AssetsLibrary** framework, and add it.

Then add the following `#import` statement to the top of **ViewController.m**:

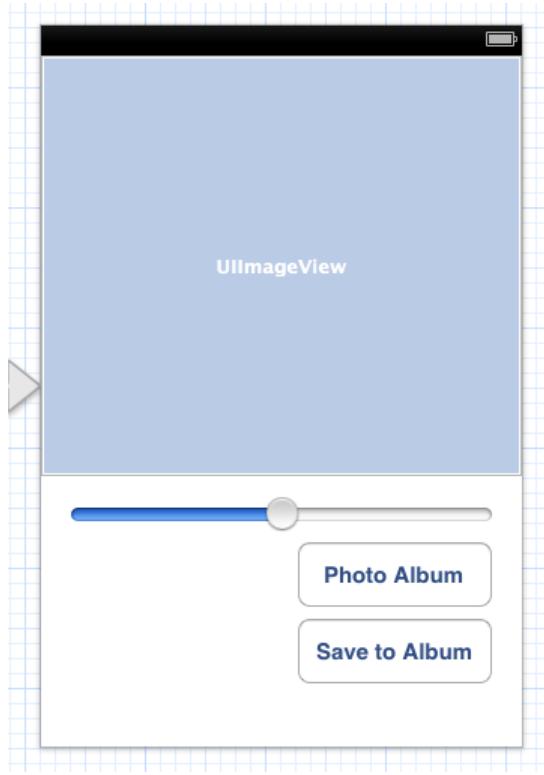
```
#import <AssetsLibrary/AssetsLibrary.h>
```

One thing you should know is that when you save a photo to the album, it's a process that could continue even after you leave the app.

This could be a problem as the GPU stops whatever it's doing when you switch from one app to another. If the photo isn't finished being saved, it won't be there when you go looking for it later!

The solution to this is to use the CPU rendering `CIContext`. The default is the GPU, and the GPU is much faster. You can create a second `CIContext` just for the purpose of saving this file.

Let's add a new button to your app that will let you save the photo you are currently modifying with all the changes you've made. Open **MainStoryboard.storyboard** add a new button labeled "Save to Album":



Then connect it to a new `savePhoto:` method, like you did last time.

Then switch to **ViewController.m** and implement the method as follows:

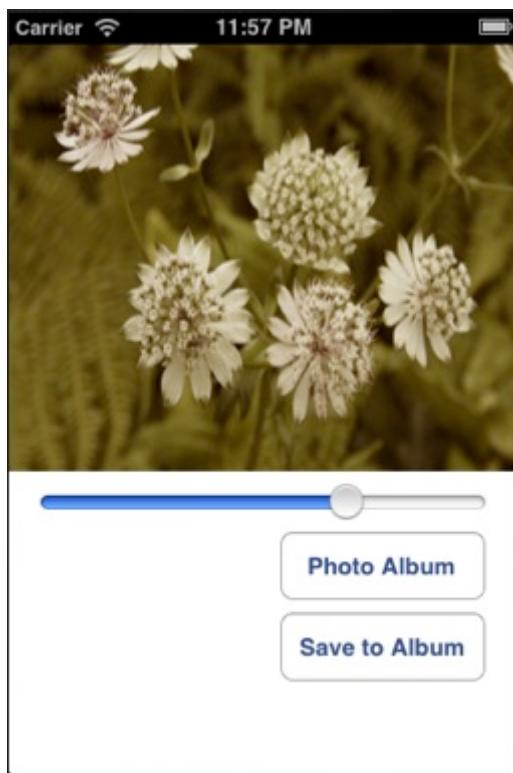
```
- (IBAction)savePhoto:(id)sender {
    // 1
    CIImage *saveToSave = [filter outputImage];
    // 2
    CIContext *softwareContext = [CIContext
        contextWithOptions:
        @{@"kCIContextUseSoftwareRenderer" : @(YES)} ];
    // 3
    CGImageRef cgImg = [softwareContext createCGImage:saveToSave
        fromRect:[saveToSave extent]];
    // 4
    ALAssetsLibrary* library = [[ALAssetsLibrary alloc] init];
    [library writeImageToSavedPhotosAlbum:cgImg
        metadata:[saveToSave properties] completionBlock:
        ^(NSURL *assetURL, NSError *error) {
    // 5
    CGImageRelease(cgImg);
```

```
    }];
}
```

In this code block you:

1. Get the `CIImage` output from the filter.
2. Create a new, software based `CIContext`.
3. Generate the `CGImageRef`.
4. Save the `CGImageRef` to the photo library.
5. Release the `CGImage`. That last step happens in a callback block so that it only fires after you're done using it.

Build and run the app (remember, on an actual device since you're using software rendering), and now you can save that "perfect image" to your photo library so it's preserved forever!



What About Image Metadata?

Let's talk about image metadata for a moment. Image files taken on mobile phones have a variety of data associated with them, such as GPS coordinates, image format, and orientation. Specifically orientation is something that you need to preserve. The process of loading into a `CIImage`, rendering to a `CGImage`, and converting to a `UIImage` strips the metadata from the image. In order to preserve orientation, you'll need to record it and then put it back into the `UIImage`.

Start by adding a new private instance variable to **ViewController.m**:

```
@implementation ViewController {
    CIContext *context;
    CIFilter *filter;
    CIImage *beginImage;
    UIImageOrientation orientation; // New!
}
```

Next, set the value when you load the image from the photo library in the `-imagePickerController:didFinishPickingMediaWithInfo:` method. Add the following line before the “`beginImage = [CIImage imageWithCGImage:gotImage.CGImage]`” line:

```
orientation = gotImage.imageOrientation;
```

Finally, alter the line in `amountsSliderChanged:` creates the `UIImage` that you set to the `imageView` object:

```
UIImage *newImage = [UIImage imageWithCGImage:cgi img scale:1.0
                                         orientation:orientation];
```

Now, if you take a picture taken in something other than the default orientation, it will be preserved.

What Other Filters are Available?

The `CIFilter` API has 130 filters on the Mac OS plus the ability to create custom filters. In iOS 6, it has 93 or more. Currently there isn’t a way to build custom filters on the iOS platform, but it’s possible that it will come.

In order to find out what filters are available, you can use the `[CIFilter filterNamesInCategory:kCICategoryBuiltIn]` method. This method will return an array of filter names. In addition, each filter has an `attributes` method that will return a dictionary containing information about that filter. This information includes the filter’s name, the kinds of inputs the filter takes, the default and acceptable values for the inputs, and the filter’s category.

Let’s put together a method for your class that will print all the information for all the currently available filters to the log. Add this method right above `viewDidLoad`:

```
-(void)logAllFilters {
    NSArray *properties = [CIFilter
        filterNamesInCategory:kCICategoryBuiltIn];
    NSLog(@"%@", properties);
    for (NSString *filterName in properties) {
        CIFilter *fltr = [CIFilter filterWithName:filterName];
        NSLog(@"%@", [fltr attributes]);
    }
}
```

```
}
```

This method simply gets the array of filters from the `filterNamesInCategory` method. It prints the list of names first. Then, for each name in the list, it creates that filter and logs the attributes dictionary from that filter.

Then call this method at the end of `viewDidLoad`:

```
[self logAllFilters];
```

You will see the following in the log output:

```
CISepiaTone,
CISoftLightBlendMode,
CISourceAtopCompositing,
CISourceInCompositing,
CISourceOutCompositing,
CISourceOverCompositing,
CIStraightenFilter,
CIStripesGenerator,
CITemperatureAndTint,
CIToneCurve,
CIVibrance,
CIVignette,
CIWhitePointAdjust
)
2011-09-28 00:53:13.710 CIPProject[4143:707] {
    CIAttributeFilterCategories = (
        CIAttributeFilterCategoryCompositeOperation,
        CIAttributeFilterCategoryVideo,
        CIAttributeFilterCategoryStillImage,
        CIAttributeFilterCategoryInterlaced,
        CIAttributeFilterCategoryNonSquarePixels,
        CIAttributeFilterCategoryHighDynamicRange,
        CIAttributeFilterCategoryBuiltIn
);
    CIAttributeFilterDisplayName = Addition;
    CIAttributeFilterName = CIAdditionCompositing;
    inputBackgroundImage = {
        CIAttributeClass = CIImage;
        CIAttributeType = CIAttributeTypeImage;
    };
    inputImage = {
        CIAttributeClass = CIImage;
        CIAttributeType = CIAttributeTypeImage;
    };
}
2011-09-28 00:53:13.713 CIPProject[4143:707] {
    CIAttributeFilterCategories = (
        CIAttributeFilterCategoryGeometryAdjustment,
        CIAttributeFilterCategoryVideo,
        CIAttributeFilterCategoryStillImage,
```

Wow, that's a lot of filters!

Note: Some of these filters came out on iOS 5, and some came out on iOS 6. The best way to get the list of which filters exist back in iOS 5 is to run this code on a device running iOS 5.

More Intricate Filter Chains

Now that you've looked at all the filters that are available on the iOS 6 platform, it's time to create a more intricate filter chain. In order to do this, you'll create a dedicated method to process the `CIImage`. It will take in a `CIImage`, filter it, and return a `CIImage`. Add the following method:

```
- (CIImage *)oldPhoto:(CIImage *)img withAmount:(float)intensity {
    //1
    CIFilter *sepia = [CIFilter filterWithName:@"CISepiaTone"];
    [sepia setValue:img forKey:kCIIInputImageKey];
    [sepia setValue:@(intensity) forKey:@"inputIntensity"];

    //2
    CIFilter *random = [CIFilter
        filterWithName:@"CIRandomGenerator"];

    //3
    CIFilter *lighten = [CIFilter
        filterWithName:@"CIColorControls"];
    [lighten setValue:random.outputImage
        forKey:kCIIInputImageKey];
    [lighten setValue:@(1 - intensity)
        forKey:@"inputBrightness"];
    [lighten setValue:@0.0 forKey:@"inputSaturation"];

    //4
    CIImage *croppedImage = [lighten.outputImage
        imageByCroppingToRect:[beginImage extent]];

    //5
    CIFilter *composite = [CIFilter
        filterWithName:@"CIHardLightBlendMode"];
    [composite setValue:sepia.outputImage
        forKey:kCIIInputImageKey];
    [composite setValue:croppedImage
        forKey:kCIIInputBackgroundImageKey];

    //6
```

```
CIFilter *vignette = [CIFilter  
    filterWithName:@“CIVignette”];  
[vignette setValue:composite.outputImage  
    forKey:kCIIInputImageKey];  
[vignette setValue:@(intensity * 2)  
    forKey:@“inputIntensity”];  
[vignette setValue:@(intensity * 30) forKey:@“inputRadius”];  
  
//7  
return vignette.outputImage;  
}
```

Let's go over this section by section:

1. In section one you set up the sepia filter the same way you did in the simpler scenario. You're passing in the float in the method to set the intensity of the sepia. This value will be provided by the slider.
2. In the second section you set up a filter that is new to iOS 6 (though not new on the Mac). The random filter creates a random noise pattern, it looks like this:



It doesn't take any parameters. You'll use this noise pattern to add texture to your final old photo look.

3. In section three, you are altering the output of the random noise generator. You want to change it to grey and lighten it up a little bit so the effect is less dramatic. You'll notice that the input image key is set to the .outputImage property of the random filter. This is a convenient way to chain the output of one filter into the input of the next.
4. The fourth section you make use of a convenient method on `CUIImage`. The `imageByCroppingToRect` method takes an output `CUIImage` and crops it to the provided rect. In this case, you need to crop the output of the `CIRandomGenerator` filter because it is infinite. As a generated `CUIImage`, goes on infinitely. If you don't crop it at some point, you'll get an error saying that the filters have 'an infinte

extent'. `CIImages` don't actually contain data, they describe it. It's not until you call a method on the `CIContext` that data is actually processed.

5. In section five you are combining the output of the sepia filter with the output of the alter `CIRandom` filter. This filter does the exact same operation as the 'Hard Light' setting does in a photoshop layer. Most of (if not all, I'm not sure) the options in photoshop are available in Core Image.
6. In the sixth section, you run a vignette filter on this composited output that darkens the edges of the photo. You're using the value from the slider to set the radius and intensity of this effect.
7. Finally, you return the output of the last filter.

That's all for this filter. You can get an idea of how complex these filter chains may become. By combining Core Image filters into these kinds of chains, you can achieve endless different effects.

The next thing to do is implement this method in `amountSliderValueChanged:`. Change these two lines:

```
[filter setValue:@(slideValue) forKey:@"inputIntensity"];
CIImage *outputImage = [filter outputImage];
```

To this one line:

```
CIImage *outputImage =
[self oldPhoto:beginImage withAmount:slideValue];
```

This just replaces the previous method of setting the `outputImage` variable to your new method. You pass in the slider value for the intensity and you use the `beginImage`, which you set in the `viewDidLoad` method as the input `CIImage`.

Build and run now and you should get a cool looking old photo effect!



That noise could probably be more subtle, but I'll leave that experiment to you, dear reader. Now you have the power of Core Image. Go crazy!

Where To Go From Here?

That about covers the basics of using Core Image filters. It's a pretty handy technique, and you should be able to use it to apply some neat filters to images quite quickly!

If you want to learn more about Core Image, keep reading the next chapter, which covers some more advanced photo manipulation techniques such as compositing, masking, and face detection.

Also, our new book *iOS 6 by Tutorials* also has a very awesome chapter on how to use Core Image with **AVFoundation** to create a live video recording app that filters the video in real time.

If you have any questions or comments on this tutorial or Core Image in general, please join the forum discussion on raywenderlich.com!

Chapter 21: Intermediate Core Image

By Jacob Gundersen

In the previous chapter, you learned the basics of using Core Image: creating a context, applying a filter, and generating an image from the result.

You learned about how to apply a single filter, as well as combining multiple filters for a more nuanced effect. In this chapter, you'll see examples of several more filters, along with some other cool things you can do with Core Image, such as compositing, masking, and face detection!

Compositing filters

As you saw in the `oldPhoto` example in the previous chapter, filters can be chained together. There are a number of different kinds of filters, each having its own purpose within the context of chaining filters together.

As of iOS 6, there are roughly 90 filters available in the Core Image framework.

The output of one filter can be passed in as the input of another filter. The ability to composite filters creates virtually any possible desired effect. What's more, the process is lazy. The system doesn't process the filters one at a time, but rather waits until the entire chain is constructed and then creates a composite filter to apply to the image. This results in speedy performance.

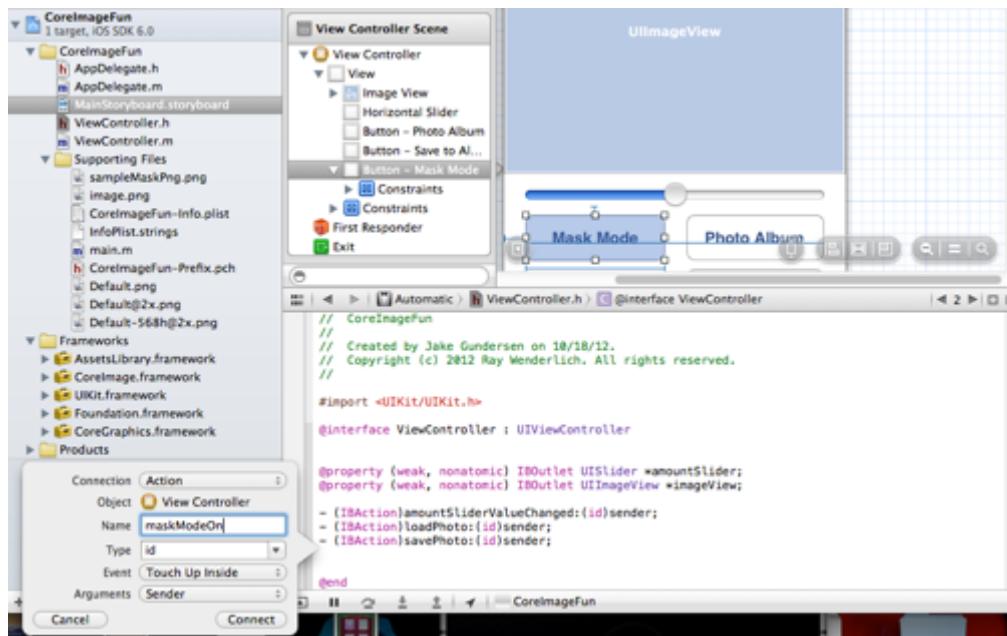
You're going to use this chaining to create a simple app that will allow you to 'paint' a transparent circle on one image, revealing another image underneath.

You'll be using the following two filters: `CISourceAtopComposition` and `CISepiaTone`.

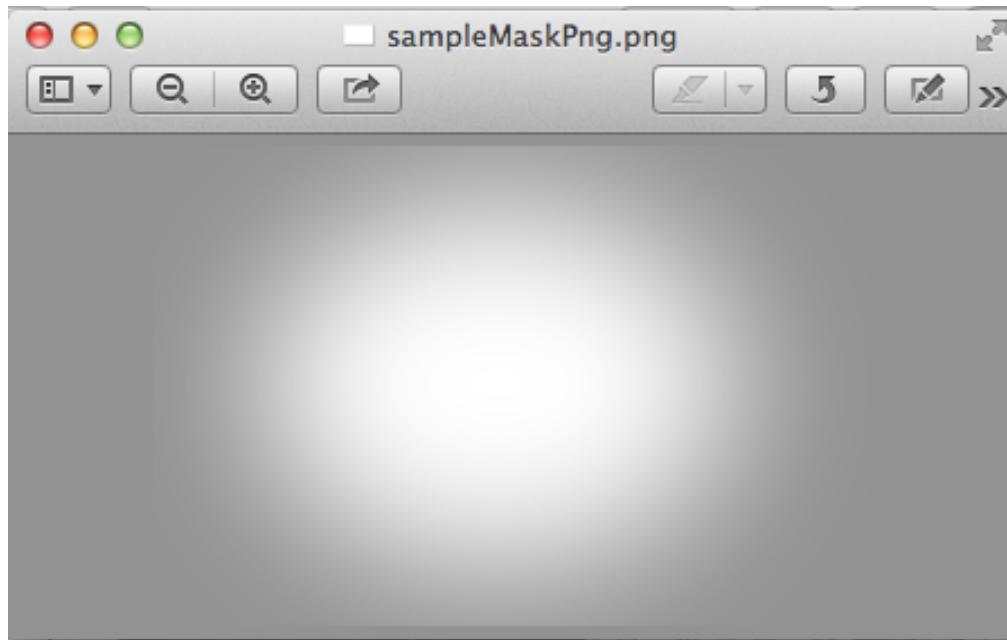
This chapter picks up where you left off in the last chapter, so make sure you have your **CoreImageFun** project open and ready.

First you need to set up a button that will activate this new mode. The button will have an outlet called `maskModeButton`.

To do this, open **MainStoryboard.storyboard** and create a new button to the left of the Photo Album button titled “Mask Mode” and connect it to a new **maskModeOn** method as shown below:



The resources for this chapter include an image you'll be using for a mask - **sampleMaskPng.png**, so add it into your project. Open it up to take a peek, and you'll see it's just a white sphere with a radial gradient that fades to transparent.



You're going to use a filter called `CISourceAtopCompositing`. It uses the white value of your sphere object to composite with the underlying layer. Where there's transparent space, the underlying image will be masked.

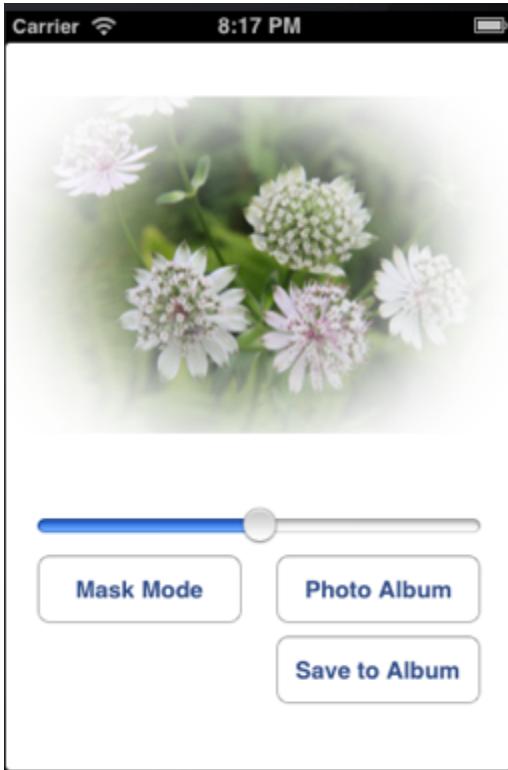
Implement the `maskModeOn:` method as follows:

```
- (IBAction)maskModeOn:(id)sender {
    //1
    CIImage *maskImage = [CIImage imageWithCGImage:
        [UIImage imageNamed:@"sampleMaskPng.png"].CGImage];
    //2
    CIFilter *maskFilter = [CIFilter
        filterWithName:@"CISSourceAtopCompositing"
        keysAndValues:kCIInputImageKey, beginImage,
        @"inputBackgroundImage", maskImage, nil];
    //3
    CGImageRef cgImg = [context
        createCGImage:maskFilter.outputImage
        fromRect:[maskFilter.outputImage extent]];
    //4
    [self.imageView setImage:[UIImage imageWithCGImage:cgImg]];
    //5
    CGImageRelease(cgImg);
}
```

This code creates a `CIImage` from your `sampleMaskPng.png` file. You're using a different method here to load the image. It's probably less efficient to do it this way, because you are first creating the `UIImage` object, then getting the `CGImage` from the `UIImage`, which you use to create the `CIImage`. But, I wanted to demonstrate that you can use `UIImage` as an intermediate step if that's convenient (like if you already are using the `UIImage`).

You create the new filter. This filter has only two input keys, `kCIInputImageKey`, and `@"inputBackgroundImage"`. Then you are using the same process to display the image. You don't use your `[self changeValue]` method here because that would load the SepiaTone Filter.

Build and run and hit the new button, and you should see the following:

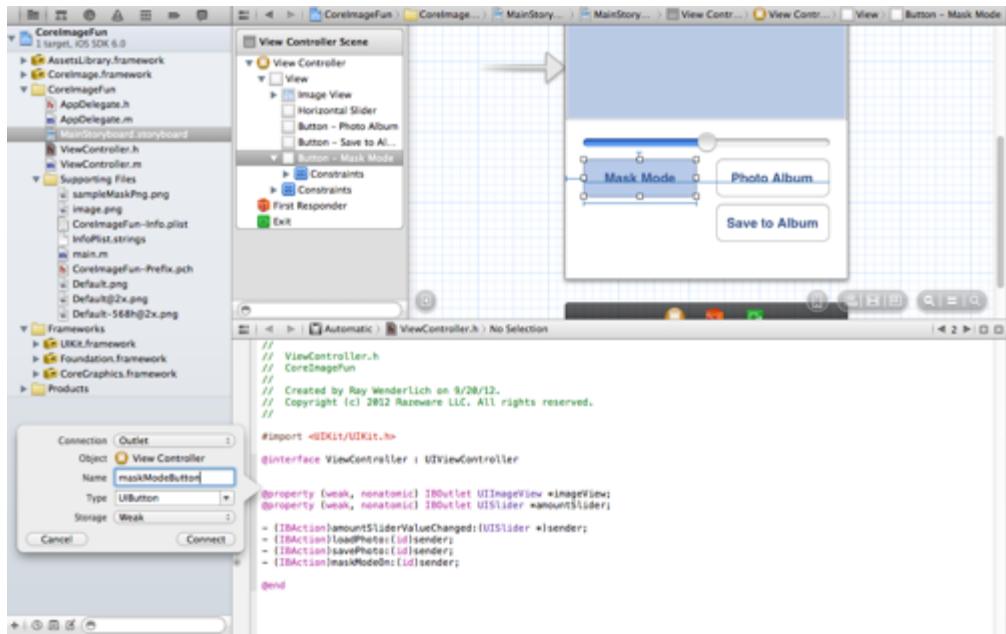


You'll notice that if you use the slider, it will revert to the previous `oldPhoto` method. Press the button again, and it switches back to your new `maskModeOn` method. Now you'll change the mask mode to use both at the same time.

You'll need to create a boolean or some other method to keep track of whether the mask mode is on or off, and you'll need to plugin the `oldPhoto` filter chain. In this case, you'll just use the text of the button to keep track of whether it's on or off, by changing the text when on to "Mask Mode Off".

Because the slider's method is the one controlling the changes when you slide, it's there that you need to check for whether the mask mode is on or off. Also, you need to add the output of your `CISepiaTone` filter to the input of the `CISourceCompositingAtop` filter.

First you need to create an instance variable in **MainStoryboard.storyboard** for the mask mode button. Just use a control drag to create an `IBOutlet` called `maskModeButton` in the header. Like this:



Then you need to change your methods to incorporate some new code. You're going to refactor the code in the current `maskModeOn`. First, add a new instance variable to the `@implementation` block in **ViewController.m**:

```
BOOL maskMode;
```

Next, initialize the value of this Boolean to `NO`. Also, you can comment out the method that prints out all the filters to the console. This code goes in `viewDidLoad`:

```
maskMode = NO;
//[[self logAllFilters];
```

Now, change the operation of the `maskModeOn` to simply flip the value of the `maskMode` Boolean:

```
- (IBAction)maskModeOn:(id)sender {
    if (maskMode) {
        maskMode = NO;
        [self.maskModeButton setTitle:@"Mask Mode On"
forState:UIControlStateNormal];
    } else {
        maskMode = YES;
        [self.maskModeButton setTitle:@"Mask Mode Off"
forState:UIControlStateNormal];
    }
    [self.amountSliderValueChanged:self.amountSlider];
}
```

This method first checks the value of the Boolean. If it's positive, then you will set to the opposite and vice versa. It changes the title on the button to reflect the action that pressing the button will perform. Also, the method calls `amountSliderValueChanged` and passes in the slider object.

The `maskModeOn` method is no longer doing any of the work that sets the mask. That will now be moved to its own method.

Implement this new method:

```
- (CIImage *)maskImage:(CIImage *)inputImage {
    UIImage *mask = [UIImage imageNamed:@"sampleMaskPng.png"];
    CIImage *maskImage =
        [CIImage imageWithCGImage:mask.CGImage];
    CIFilter *maskFilter = [CIFilter
        filterWithName:@"CISourceAtopCompositing"
        keysAndValues:kCIInputImageKey, inputImage,
        kCIInputBackgroundImageKey, maskImage, nil];
    return maskFilter.outputImage;
}
```

This method takes an input `CIImage` applies the mask to it and returns a `CIImage`. The logic is the same as the previous method. The only difference is that, instead of taking the output of the filter and creating a `CGImageRef`, you just return the `CIImage` object.

Now, just incorporate this new method into the main `amountSliderValueChanged` method:

```
- (IBAction)amountSliderValueChanged:(UISlider *)slider {
    float slideValue = slider.value;

    CIImage *outputImage = [self oldPhoto:beginImage
withAmount:slideValue];

    //New Code!

    if (maskMode) {
        outputImage = [self maskImage:outputImage];
    }

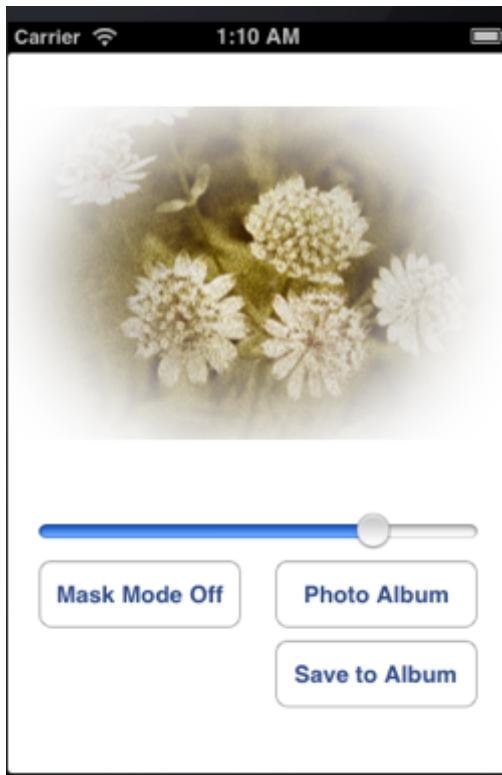
    //Old code
    CGImageRef cgimg = [context createCGImage:outputImage
                                         fromRect:[outputImage
extent]];
}
```

```
UIImage *newImage = [UIImage imageWithCGImage:cgimg scale:1.0
orientation:orientation];
self.imageView.image = newImage;

CGImageRelease(cgimg);
}
```

In the marked code, you check the value of the Boolean first. If necessary, you call the maskImage method passing in the outputImage object (and setting that variable to the returned object).

Easy peasy! Build and run now. You'll be able to turn the mask on and off, and you'll be able to use the slider to increase/decrease the oldPhoto effect.



And that is an example of compositing two filters together for a cool effect!

Combining a filtered image with another

What you'd like to do next is be able to have two layers (your current sepia layer and a new layer) that consist of a chosen image that will appear beneath your current masked image.

From the resources for this chapter, add **bryce.png** to your project. You're going to use this image as your bottom layer image.

Add the following code your class:

```
- (CIImage *)addBackgroundLayer:(CIImage *)inputImage {
    UIImage *backImage = [UIImage imageNamed:@"bryce.png"];
    CIImage *bg = [CIImage imageWithCGImage:backImage.CGImage];
    CIFilter *sourceOver = [CIFilter
        filterWithName:@"CICompositeOver"
        keysAndValues:kCIIputImageKey, inputImage,
        @"inputBackgroundImage", bg, nil];
    return sourceOver.outputImage;
}
```

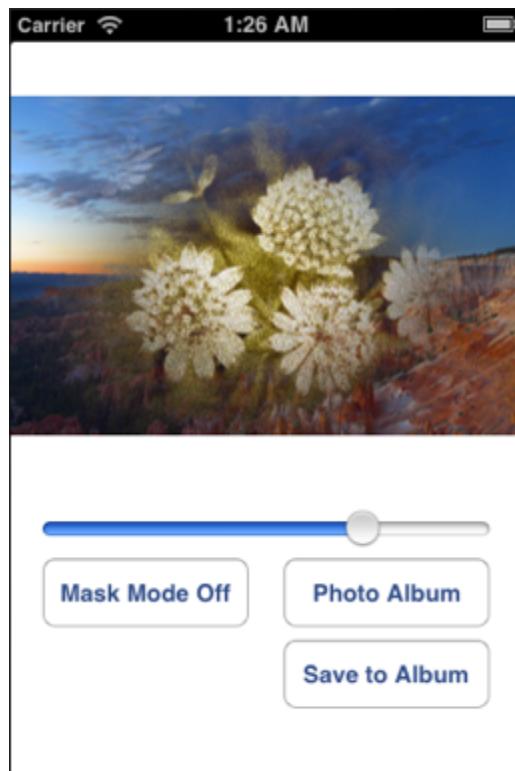
This code should make sense at this point. This method is going to take an input image you supply and composite it with the bryce.png file. The compositing method will simply place the bryce image underneath the supplied image. If the supplied image is not transparent, you won't see bryce at all.

However, when you use your `oldPhoto`'d output image as the input image for this filter, you'll see the bryce in those transparent areas.

Add the following line to both the `amountSliderValueChanged` method right after the `maskMode` if statement:

```
outputImage = [self addBackgroundLayer:outputImage];
```

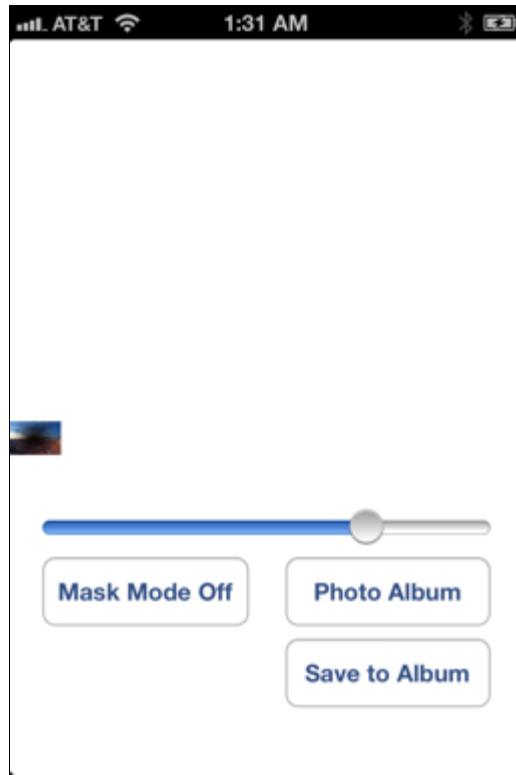
Build and run now:



How easy was that!?

Resizing your input image

One problem you'll run into now is that if you are using images of different sizes, the `CIImage` filter chain you've set up is combining an image of say 400 pixels wide by 300 pixels tall with your image of bryce which is only 320 wide by 213 tall. Go ahead and try using a photo from your album and you'll see this:



What it's doing is removing all of the pixels from the original larger image that don't interact with a corresponding pixel from the mask image. Then the whole thing is scaled to fit inside your `UIImageView` container. You need to resize the incoming image from the photos album so it's the same size as the image of bryce.

You're going to use a `UIImage` category to add a method to resize the `UIImage`. Import the **UIImage+Resize.h/.m** files into your project from the resources for this chapter. Make sure that add to target is checked.

Explaining the Core Graphics drawing code in this method is beyond the scope of this tutorial, but feel free to look through if you're extra curious.

Using it is easy though. First import the header at the top of **ViewController.m**:

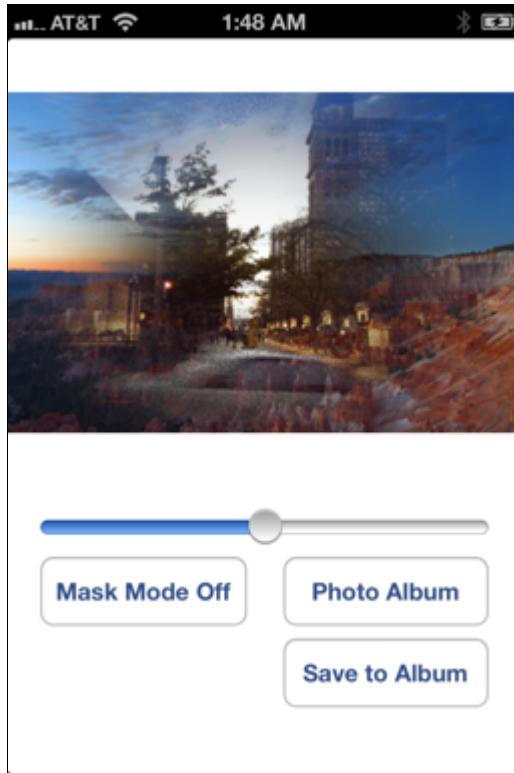
```
#import "UIImage+Resize.h"
```

Then add the following line of code in your `imagePickerController:didFinishPickingMediaWithInfo` method right after the line that sets the `gotImage`:

```
gotImage = [gotImage scaleToSize:[beginImage extent].size];
```

This will fix the problem. You are scaling the image that you've selected from your photo album to the same size as the `beginImage` `CIImage` object. These classes make it easy.

If you run now your photo album images will correctly composite!



Drawing a mask

Masking with a pre-built image is cool, but you know what is even cooler? Drawing the mask dynamically!

You can create a `CIImage` with either raw pixel data or with a `CGImage`. If you draw a `CGImage` using Core Graphics calls, you can pass that in as a `CIImage` for one of your filters. This capability opens up all kinds of possible interaction.

Add the code to dynamically draw a circle mask first:

```
- (CGImageRef)drawMyCircleMask:(CGPoint)location {
    NSUInteger width =
        (NSUInteger)self.imageView.frame.size.width;
    NSUInteger height =
        (NSUInteger)self.imageView.frame.size.height;
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
```

```
NSUInteger bytesPerPixel = 4;
NSUInteger bytesPerRow = bytesPerPixel * width;
NSUInteger bitsPerComponent = 8;
CGContextRef cgcontext = CGBitmapContextCreate(NULL, width,
    height, bitsPerComponent, bytesPerRow, colorSpace,
    kCGImageAlphaPremultipliedLast | kCGBitmapByteOrder32Big);
CGColorSpaceRelease(colorSpace);
CGContextSetRGBFillColor(cgcontext, 1, 1, 0.5, .7);
CGContextFillEllipseInRect(cgcontext,
    CGRectMake(location.x - 25, location.y - 25, 50.0, 50.0));
CGImageRef cgImg = CGBitmapContextCreateImage(cgcontext);
CGContextRelease(cgcontext);
return cgImg;
}
```

This code looks long and scary. Mostly it's just boilerplate that sets up the `CGContext`. A `CGContext` is the data representing the image. In the first five lines you are creating the values you need to set up a `CGContextRef`, namely:

1. **Height.** Just get this from the `imageView`.
2. **Width.** Same as height.
3. **Colorspace.** Defines the data that describes the color.
`CGColorSpaceCreateDeviceRGB()` is what I usually use for iOS projects.
4. **BytesPerPixel.** This is the amount of data in each pixel. Because the data will be RGB + Alpha, each 1 byte, you'll need a total of 4 bytes.
5. **BytesPerRow.** This is just how many bytes are in a full row of pixels, so width times `bytesPerPixel`.
6. **BitsPerComponent.** This mentions the bit depth of each value RGB and Alpha. Each is 8 bits, which is large enough to contain 256 values.

Once you've set up all the variables, you create the `cgcontextRef` using the `CGBitmapContextCreate` function. When you draw with core graphics you always need a context. Sometimes that can be the context that represents what you're seeing on screen, in this case it's a bitmap context, or that data that will usually become an image file.

The important lines are towards the end. You're calling `CGContextSetRGBFillColor` to set your color to yellow (1, 1, 0.5) with a transparency value of 70% (.7). You're setting that for your bitmap context. Next you call the `CGContextFillEllipseInRect` method that actually draws your circle. You're gonna be using a circle as a mask.

You give it an x and a y origin, and then the width and height. You are using the touch point that you passed in for this location. You want the circle to be centered around your touch, so you need to subtract half the width and height (25) from the origin points.

You then use what you've drawn in your context to create a `CGImageRef`. This will be returned to calling object.

Next you need to set this method up to fire. Let's add some touch methods:

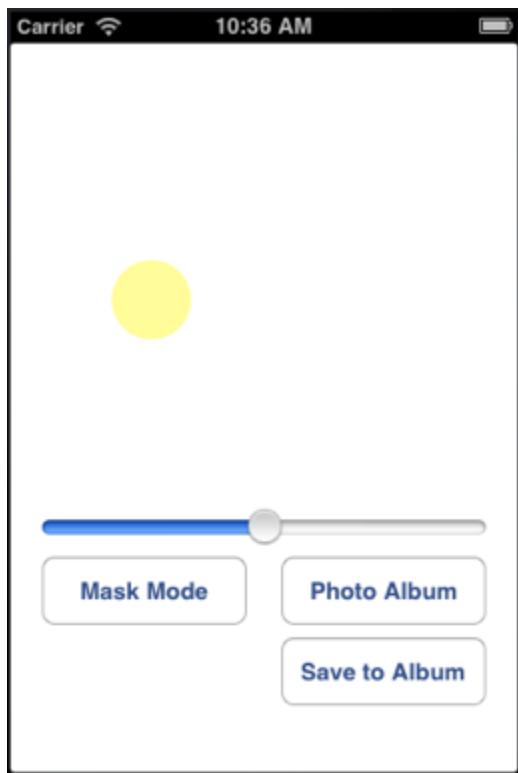
```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    CGPoint loc =
        [[touches anyObject] locationInView:self.imageView];
    if (loc.y <= self.imageView.frame.size.height && loc.y >= 0) {
        loc = CGPointMake(loc.x,
                          self.imageView.frame.size.height - loc.y);
        CGImageRef cgimg = [self drawMyCircleMask:loc];
        UIImage *img = [UIImage imageWithCGImage:cgimg];
        self.imageView.image = img;
    }
}
```

You'll later replicate this method for `touchesMoved`, but for right now this will suffice. You first get the location of any of the touches in your set. You're only interested in touches that reside within the bounds of the `UIImageView` here. So you're checking to see if the touch was inside its frame. You don't need to worry about the `x` bounds, because it stretches the width of the screen.

Next, you convert the coordinate space from `UIView` to Core Graphics by subtracting the `y` coordinate from the height of the `imageView`. Then you create a `CGImageRef` from the method you just created, passing in your `loc` `CGPoint`.

What you're going to do ultimately is pass this `CGImage` into a `CIImage` and `CIFilter`, but for now, let's just convert it to a `UIImage` and display it on screen so you can make sure that everything's working as it should.

If you build and run now you should have the following when you touch the `UIImageView`:



You can draw a yellow circle with a touch!

Now that you have everything you need in place, you should be able to use this new `CGImage` as your `CIIImage` in your masking filter. You want the `CIIImage` that represents the mask to be available throughout the program. So you need to add it to your list of private instance variables. Add this to the `@implementation` variables.

```
CIIImage *maskImage;
```

Next, initialize this variable in `viewDidLoad`:

```
maskImage = [CIIImage imageWithCGImage:[UIImage imageNamed:@"sampleMaskPng.png"].CGImage];
```

Now that you are initializing the mask in the `viewDidLoad` method, you no longer need to create it in the `maskImage` method. Remove these line now:

```
//UIImage *mask = [UIImage imageNamed:@"sampleMaskPng.png"];
//CIIImage *maskImage = [CIIImage imageWithCGImage:mask.CGImage];
```

Now, when you run the `touchesBegan` method, instead of setting the `imageView` image to your drawn `CGImage`, you'll be replacing the `CIIImage` in the filter with it. Here's the new code for that:

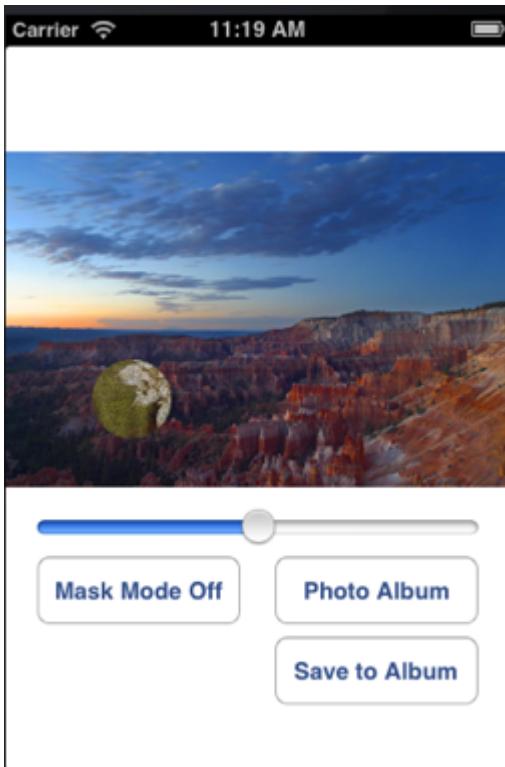
```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    CGPoint loc =
```

```
[touches anyObject] locationInView:self.imageView];
if (loc.y <= self.imageView.frame.size.height && loc.y >= 0) {
    loc = CGPointMake(loc.x, self.imageView.frame.size.height
- loc.y);
    CGImageRef cgimg = [self drawMyCircleMask:loc];
    maskImage = [CIIImage imageWithCGImage:cgimg];
    [self amountSliderValueChanged:self.amountSlider];
}
}
```

Now the `touchesBegan` is doing its work by using other methods. After getting the location of the touch inside the `imageView`, it passes that touch point in to create the circle using the `drawMyCircleMask` method.

Then it uses the circle created to set the `maskImage` to a new `CIIImage` created using the resulting bitmap context. Finally, it calls `amountSliderValueChanged` to perform the entire `CIFilter` chain anew using the new `maskImage` object.

Duplicate the entire `touchesBegan` method and rename it `touchesMoved`. Build and run now. Turn mask mode on and you'll have a little window from the Bryce image to the `oldPhoto` flower image beneath it!



But, you really want more of a “paint” type mask instead of a circle that moves. So, that’s the last thing you’ll do. You’ll want to modify your draw method to keep the

current context around and have an additive model. You'll need to add your `cgcontext` variable to your class and initialize it in your `viewDidLoad` method, then just add circles to it every time you detect a touch.

Let's do that now. First add the `cgcontext` variable to the private instance variables list (in the `@implementation` section):

```
CGContextRef cgcontext;
```

You'll break up your draw image function into two functions, one to set up the `cgcontext` and another to draw into that context.

Here's what those two functions look like, replace the `drawMyCircleMask:` method with these two:

```
- (void)setupCGContext {
    NSUInteger width = (NSUInteger)
        self.imageView.frame.size.width;
    NSUInteger height = (NSUInteger)
        self.imageView.frame.size.height;
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    NSUInteger bytesPerPixel = 4;
    NSUInteger bytesPerRow = bytesPerPixel * width;
    NSUInteger bitsPerComponent = 8;
    cgcontext = CGBitmapContextCreate(NULL, width, height,
        bitsPerComponent, bytesPerRow, colorSpace,
        kCGImageAlphaPremultipliedLast | kCGBitmapByteOrder32Big);
    CGColorSpaceRelease(colorSpace);
}

- (CGImageRef)drawMyCircleMask:(CGPoint)location
reset:(BOOL)reset {
    if (reset) {
        CGContextClearRect(cgcontext, CGRectMake(0, 0, 320,
            self.imageView.frame.size.height));
    }
    CGContextSetRGBFillColor(cgcontext, 1, 1, 1, .7);
    CGContextFillEllipseInRect(cgcontext,
        CGRectMake(location.x - 25, location.y - 25, 50.0, 50.0));
    CGImageRef cgImg = CGBitmapContextCreateImage(cgcontext);
    return cgImg;
}
```

You need to call `setupCGContext` when you initialize the view. However, you can't call it in `viewDidLoad` because it uses the frame of the `imageView` object to set up the size of the `CGContextRef`. In `viewDidLoad` the frame isn't set up yet. So, you need to add a `viewDidAppear` method and call it there, like this:

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self setupCGContext];
}
```

Note you added a second parameter to the `drawMyCircleMask` method. You're going to reset the drawing each time you call `touchesBegan`. If the passed in parameter is set to YES, then you'll call `CGContextClearRect(CGContext, CGRectMake(0, 0, 320, self.imageView.frame.size.height))`, which will clear the drawing.

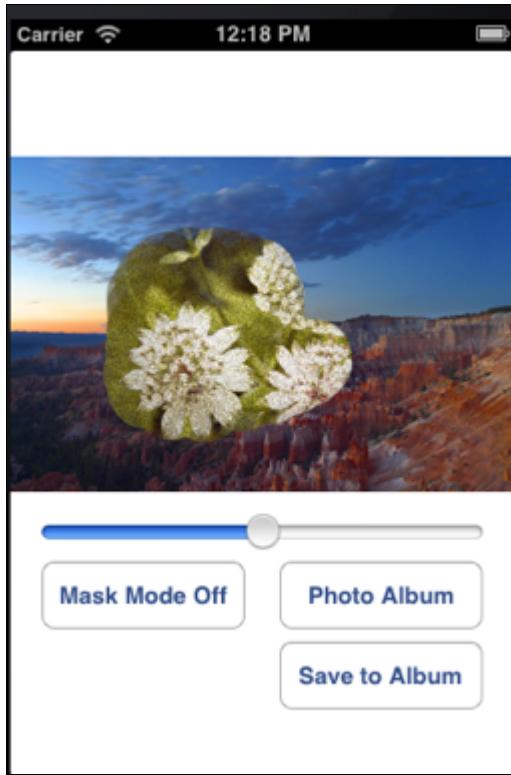
Finally, change the method calls in the `touchesBegan` and `touchesMoved` methods. In `touchesBegan` pass in YES for the `reset` parameter and NO in `touchesMoved`.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    CGPoint loc =
        [[touches anyObject] locationInView:self.imageView];
    if (loc.y <= self.imageView.frame.size.height && loc.y >= 0) {
        loc = CGPointMake(loc.x,
                          self.imageView.frame.size.height - loc.y);

        //Right here!
        CGImageRef cgimg = [self drawMyCircleMask:loc reset:YES];
        maskImage = [CIIImage imageWithCGImage:cgimg];
        [self amountSliderValueChanged:self.amountSlider];
    }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    CGPoint loc =
        [[touches anyObject] locationInView:self.imageView];
    if (loc.y <= self.imageView.frame.size.height && loc.y >= 0) {
        loc = CGPointMake(loc.x,
                          self.imageView.frame.size.height - loc.y);
        //And here!
        CGImageRef cgimg = [self drawMyCircleMask:loc reset:NO];
        maskImage = [CIIImage imageWithCGImage:cgimg];
        [self amountSliderValueChanged:self.amountSlider];
    }
}
```

Build and run, and if all has gone well, you'll have the following mini photo editing/compositing app using Core Image Filters!



Face detection

Another cool feature in Core Image is face detection. The system can detect whether one or more faces within an image and record feature locations of the faces, including a bounding box around the face, the position of each eye, and the position of the mouth. Wow!

Let's implement a function that runs the face detector and outputs the data it gets from the image to the log. You'll need to import an image with a face or faces to your photo album in order for this to work. I suggest using the image at this link to do the detection:

<http://www.stockvault.net/data/s/100377.jpg>

Then add this method to **ViewController.m**:

```
- (void)hasFace:(CIImage *)image {
    CIDetector *faceDetector = [CIDetector
        detectorOfType:CIDetectorTypeFace context:nil options:
        @{@"CIDetectorAccuracy": CIDetectorAccuracyHigh} ];

    NSArray *features = [faceDetector featuresInImage:image];
    NSLog(@"%@", features);
    for (CIFaceFeature *f in features) {
```

```

        NSLog(@"%@", f.leftEyePosition.x,
               f.leftEyePosition.y);
    }
}

```

You instantiate the face detector and give it a type. `CIDetectorTypeFace` is the only type currently. You also pass it a dictionary of options. You can use `CIDectectorAccuracyHigh` or `CIDectectorAccuracyLow`, which is faster. Here you're using high.

This function has one purpose, to create the detector and record the information in the features array. You're passing in your desired `CIIimage` as the argument that will be used for the `featuresInImage` method on the detector.

In this case you're also logging the result. You won't need this later, but to test whether the face detector is working you're using the log statements here.

Now add this line to your method that retrieves photos from your photo album, right before the `[self.amountSliderChanged:self.amountSlider]` line:

```

[self performSelectorInBackground:@selector(hasFace:)
withObject:beginImage];

```

The face detector needs to run on a background thread because it can take a little time to perform its analysis and you don't want to be waiting for it to finish. Go ahead and try this out, selecting a photo with good, front facing faces in it from your photo album. If you use the recommended image, you end up with this result in the console:

```

CoreImageFun :@fattjake's hotspot
All Our Step into
) Step into instruction (Hold Control)
2012-10-17 14:08:59.889 CoreImageFun[12128:5b5f] {
    = <CIFaceFeature: 0x1cd9c578>,
    = <CIFaceFeature: 0x1cdbe2d8>,
    = <CIFaceFeature: 0x1cd95e4d>,
    = <CIFaceFeature: 0x1cd9d9d8>
}

2012-10-17 14:08:59.889 CoreImageFun[12128:5b5f] 45.000000, 110.000000
2012-10-17 14:08:59.891 CoreImageFun[12128:5b5f] 172.000000, 71.000000
2012-10-17 14:08:59.893 CoreImageFun[12128:5b5f] 111.000000, 173.000000
2012-10-17 14:08:59.894 CoreImageFun[12128:5b5f] 213.000000, 142.000000
2012-10-17 14:08:59.896 CoreImageFun[12128:5b5f] 255.000000, 91.000000

```

Note: Orientation is an issue with face detection. If you provide an image that is in an orientation other than `UIInterfaceOrientationUp` you will need to rotate it to that orientation before handing it to the face detector.

Now lets do something with the information you've collected from your face detector. You're going to be creating small boxes that you'll composite with your `inputImage`, so that each feature, mouth, left eye, right eye, each have their own box. This way you can see how well the face detection is working.

These next few methods are a bit long, so you'll walk through it step by step to make sure you understand it well. First add the `createBox` method:

```
- (CIIImage *)createBox:(CGPoint)loc color:(CIColor *)color {
    CIFilter *constantColor = [CIFilter
filterWithName:@"CICvtColor"
keysAndValues:@{"inputColor", color, nil}];
    CIFilter *crop = [CIFilter filterWithName:@"CICrop"];
    [crop setValue:constantColor.outputImage
forKey:kCIInputImageKey];
    [crop setValue:[CIVector vectorWithCGRect:CGRectMake(loc.x -
3, loc.y - 3, 6, 6)] forKey:@"inputRectangle"];
    return crop.outputImage;
}
```

In this first method you are creating a new `CIIImage` with a small box at the center point of the passed in location. You're going to be using these boxes to mark the eyes and mouth of your faces.

The first filter creates a solid color image, the `CICvtColor` takes only one input in the form of a `CIColor`. This filter creates a solid color image with an infinite extent, which means it goes on forever unless you tell it how to crop it.

Cropping is exactly what the next filter does. You're creating a rectangle centered on the input location, `loc`. The crop filter just takes two inputs, the output of the color generator filter, and a `CGRect`.

I'll show you how to instantiate a `CIColor` object next, but it's what you'd expect. You're basically using Core Image filters to draw boxes.

On to the face marking:

```
- (CIIImage *)filteredFace:(NSArray *)features onImage:
(CIIImage *)inputImage {
    CIIImage *outputImage = inputImage;
    for (CIFaceFeature *f in features) {
        if (f.hasLeftEyePosition) {
            CIIImage *box = [self createBox:
CGPointMake(f.leftEyePosition.x, f.leftEyePosition.y)
color:[CIColor colorWithRed:1.0 green:0.0 blue:0.0 alpha:0.7]];

            outputImage = [CIFilter filterWithName:
@"CISourceAttopCompositing" keysAndValues:kCIInputImageKey, box,
kCIInputBackgroundImageKey, outputImage, nil].outputImage;
        }
        if (f.hasRightEyePosition) {
```

```
        CIImage *box = [self
createBox:CGPointMake(f.rightEyePosition.x, f.rightEyePosition.y)
color:[CIColor colorWithRed:1.0 green:0.0 blue:0.0 alpha:0.7]];

        outputImage = [CIFilter filterWithName:
@"CISourceAtopCompositing" keysAndValues:kCIInputImageKey, box,
kCIInputBackgroundImageKey, outputImage, nil].outputImage;
    }
    if (f.hasMouthPosition) {
        CIImage *box = [self
createBox:CGPointMake(f.mouthPosition.x, f.mouthPosition.y)
color:[CIColor colorWithRed:0.0 green:0.0 blue:1.0 alpha:0.7]];

        outputImage = [CIFilter filterWithName:
@"CISourceAtopCompositing" keysAndValues:kCIInputImageKey, box,
kCIInputBackgroundImageKey, outputImage, nil].outputImage;
    }
}
return outputImage;
}
```

This first creates a variable to store the image as you pass it through your filter chains.

Then, for each face, it looks for the presence of left eye, right eye, and mouth features. If it finds any of those present, it uses `createBox` along with a `CISourceAtopCompositing` filter to mark that location.

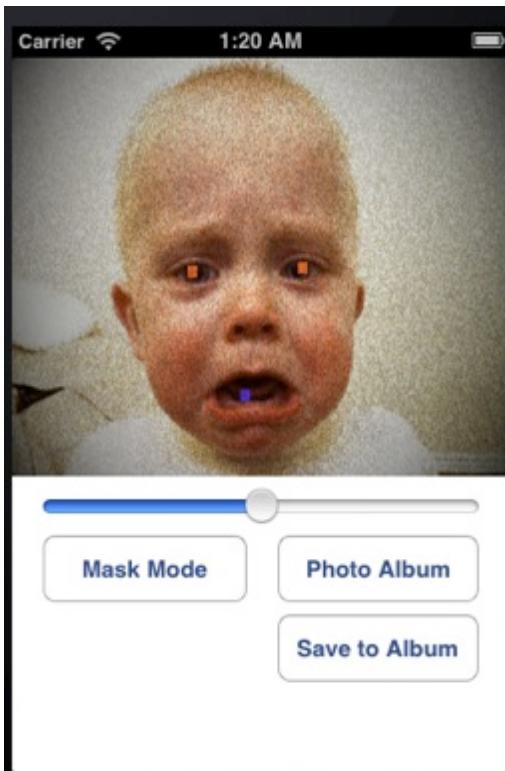
It creates a new image by compositing over the top of the previous image with the new box. It iterates over all features and all faces, until everything has been marked.

Next change the `hasFace:` method by replacing the log statements (including the for loop) with this new code:

```
beginImage = [self filteredFace:features onImage:beginImage];
[self amountSliderValueChanged:self.amountSlider];
```

This will take the `beginImage` (either the flowers or something loaded from the photo album, add markers for any features present, and return the result to the `beginImage` variable. It then calls the `amountSliderValueChanged`, which runs the filter chain again and sets the `imageView` to that new image.

You should have something along these lines:



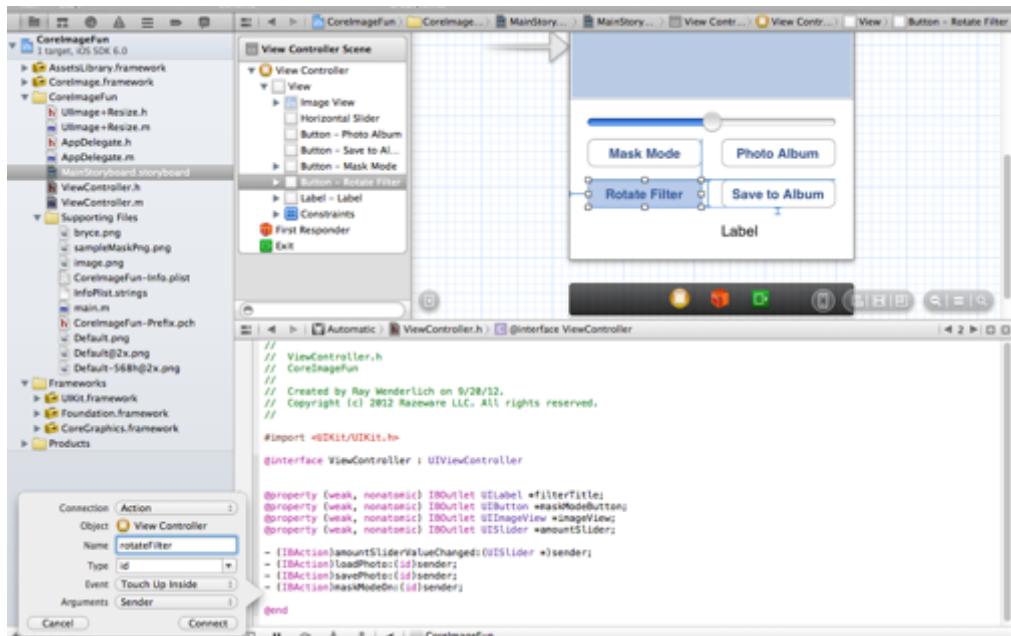
This baby might not be happy, but you should be now that you know how to code face detection! 😊

Looking at other filters

You've looked at a few filters, but this section will do a quick survey and highlight a few more to give you a general sense of what's available. You're going to set up eleven filters with preset input values, and then use a button to rotate through them.

First, add a new button and a label to your **MainStoryboard.storyboard**. The label will let you know the name of the filter you're looking at. Create an **IBOutlet** for the label and call it **filterTitle**, and connect the button to a method called **rotateFilter**.

Your layout should look similar to this:



Then switch to **ViewController.m** and add a few private instance variables like so:

```

NSArray * filtersArray;
NSUInteger filterIndx;

```

You'll need to set up the filters. You'll create a new method that initializes all these filters. Then, you'll call that method in `viewDidLoad`. Stepping through the filters will use them instead of the filter currently active. However moving the slider will reapply the main, sepia tone filter.

Add the following method:

```

-(void)loadFiltersArray {
    CIFilter *affineTransform = [CIFilter filterWithName:
        @"CIAffineTransform" keysAndValues:kCIInputImageKey, beginImage,
        @"inputTransform", [NSValue valueWithCGAffineTransform:
        CGAffineTransformMake(1.0, 0.4, 0.5, 1.0, 0.0, 0.0)], nil];

    CIFilter *straightenFilter = [CIFilter filterWithName:
        @"CIStraightenFilter" keysAndValues:kCIInputImageKey, beginImage,
        @"inputAngle", @2.0, nil];

    CIFilter *vibrance = [CIFilter filterWithName:@"CIVibrance"
        keysAndValues:kCIInputImageKey, beginImage, @"inputAmount", @-
        0.85, nil];

    CIFilter *colorControls = [CIFilter
        filterWithName:@"CIColorControls" keysAndValues:kCIInputImageKey,

```

```
beginImage, @"inputBrightness", @-0.5, @"inputContrast", @3.0,
@"inputSaturation", @1.5, nil];

    CIFilter *colorInvert = [CIFilter
filterWithName:@"CIColorInvert" keysAndValues:kCIInputImageKey,
beginImage, nil];

    CIFilter *highlightsAndShadows = [CIFilter filterWithName:
@"CIEqualize" keysAndValues:kCIInputImageKey,
beginImage, @"inputShadowAmount", @1.5, @"inputHighlightAmount",
@0.2, nil];

    CIFilter *blurFilter = [CIFilter
filterWithName:@"CIGaussianBlur" keysAndValues:kCIInputImageKey,
beginImage, nil];

    CIFilter *pixellate = [CIFilter filterWithName:@"CIPixellate"
keysAndValues:kCIInputImageKey, beginImage, @"inputScale", @10.0,
nil];

    CIFilter *bumpDistortion = [CIFilter
filterWithName:@"CIBumpDistortion" keysAndValues:kCIInputImageKey,
beginImage, @"inputScale", @0.8, @"inputCenter", [CIVector
vectorWithCGPoint:CGPointMake(160, 120)], @"inputRadius", @150,
nil];

    CIFilter *minimum = [CIFilter
filterWithName:@"CIMinimumComponent"
keysAndValues:kCIInputImageKey, beginImage, nil];

    CIFilter *circularScreen = [CIFilter
filterWithName:@"CICircularScreen" keysAndValues:kCIInputImageKey,
beginImage, @"inputWidth", @10.0, nil];

    filtersArray = @[affineTransform, straightenFilter, vibrance,
colorControls, colorInvert, highlightsAndShadows, blurFilter,
pixellate, bumpDistortion, minimum, circularScreen];
    filterIndx = 0;

}
```

You have provided values for many of these filters. However, filters have default values that are used if you don't supply any. Keep that in mind.

I'll cover what each does:

1. **CIAffineTransform** helps you apply a `CGAffineTransform` to an image. These transforms can scale, rotate, and skew an image.
2. **CIStraightenFilter** is used to fix an image take with the camera tilted slightly. You give it an `inputAngle` property to determine the amount of rotation to correct.
3. **CIVibrance** will increase the color intensity.
4. **CIColorControls** allow you to change brightness, contrast, and saturation.
5. **CIColorInvert** will invert the colors. It only takes one parameter, `inputImage`.
6. **CIHighlightShadowAdjust** will allow you to change the values of the highlights and shadows in your image.
7. **CIGaussianBlur** blurs the input image. A blur is a common operation in image processing, but is resource intensive.
8. **CIPixelate** samples the image at square intervals and creates larger pixels from that color.
9. **CIBumpDistortion** warps the image at a location in a circular way.
10. **CIMinimumComponent** Returns a grayscale image from $\min(r,g,b)$.
11. **CICircularScreen** create a circular pattern of lines where the darker the underlying image, the thicker the lines.

Now add this code to your `rotateFilter` method:

```
- (IBAction)rotateFilter:(id)sender {
    CIFilter *filt = [filtersArray objectAtIndex:filterIdx];
    CGImageRef imgRef = [context createCGImagefilt.outputImage
        fromRect:[filt.outputImage extent]];
    [self.imageView setImage:[UIImage imageWithCGImage:imgRef]];
    CGImageRelease(imgRef);
    filterIdx = (filterIdx + 1) % [filtersArray count];
    [self.filterTitle setText:[[filt attributes]
        valueForKey:@"CIAtributeFilterDisplayName"]];
}
```

In this method you're using the `filterIdx` variable to retrieve the filter from the array. Then you go through the regular process of converting a `CIImage` output from a filter into a `UIImage` you can use. Finally you increment the `filterIdx` variable and you also update your `filterTitle` label to show the current filter.

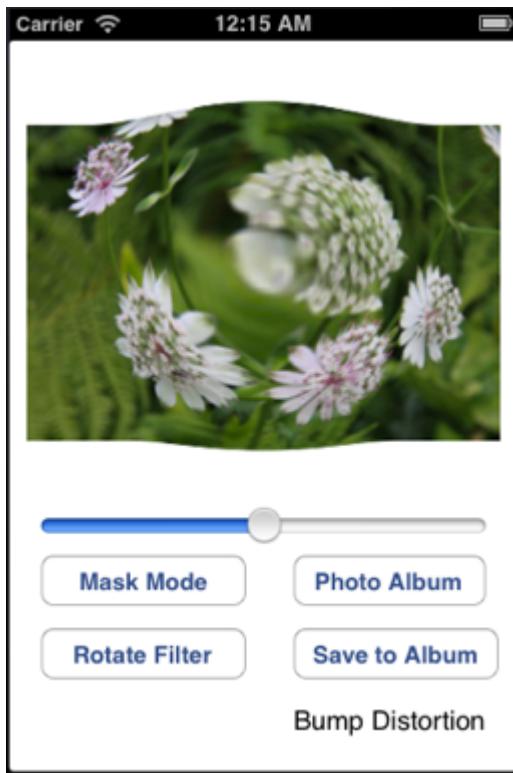
You need to call this method at the end of `viewDidLoad`:

```
[self loadFiltersArray];
```

These new filters will not interact with the slider. So, when you do decide to use the mask mode button or the slider, it will revert you back to the sepia tone filter. You want to update the label in that case. Add the following method to the end of the `changeValue` method:

```
[self.filterTitle setText:@"Sepia Tone Composite"];
```

Go ahead and build and run. You should be able to cycle through a variety of filters to get an idea of some more cool Core Image goodies you could use! If you want to experiment with some of the other Core Image filters, you can just add them to this list.



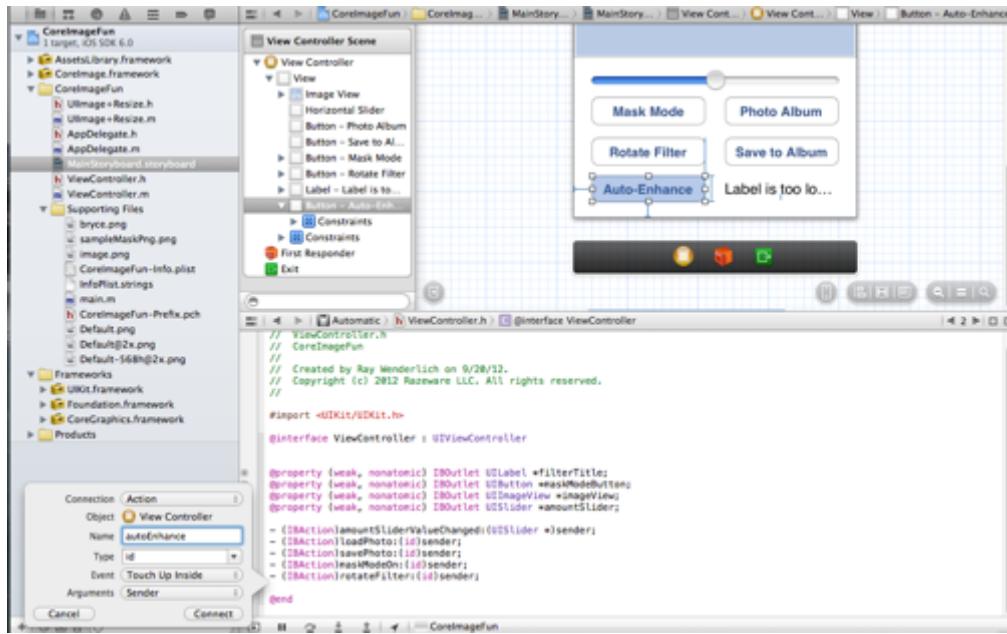
Auto-Enhancement

Apple has another neato feature called auto-enhancement. It's a method that you can call on a `CIImage` that will return an array of filters including things like vibrance, highlights and shadows, red eye reduction, flesh tone, etc.

The array can then be used to apply the filter chain to an image or some of the filters can be turned off.

You're going to implement a method that simply applies all filters to an image. Add a new button titled "Auto Enhance" and attach an IBAction method called `autoEnhance`.

Here's a screenshot of adding the method:



And implement the method as follows:

```
- (IBAction)autoEnhance:(id)sender {
    CIImage *outputImage = beginImage;
    NSArray *ar = [beginImage autoAdjustmentFilters];
    for (CIFilter *fil in ar) {
        [fil setValue:outputImage forKey:kCIInputImageKey];
        outputImage = fil.outputImage;
        NSLog(@"%@", [[fil attributes]
            valueForKey:@"CIAttributeFilterDisplayName"]);
    }
    [filter setValue:outputImage forKey:kCIInputImageKey];
    CGImageRef cgimg = [context createCGImage:outputImage
        fromRect:[outputImage extent]];

    UIImage *newImage = [UIImage imageWithCGImage:cgimg
        scale:1.0 orientation:orientation];
    self.imageView.image = newImage;

    CGImageRelease(cgimg);
}
```

Here you get the array of filters with the `[beginImage autoAdjustmentFilters]` call. Then you iterate through each filter. You first have to set your input image. You set that to `outputImage`, which is the output from the last filter operation. In this way you chain all these filter actions together. Also, you are logging the name of each filter so you can see what the auto-enhancement is doing.

Finally you set the new image as the `inputImage` for your main filter, and call run through the process of drawing a `CGImage` and setting the `UIImageView`.

Build and run and choose an image from your photo library. You should see greater detail and color balance when you press the auto enhance button!

```

}
- (IBAction)rotateFilter:(id)sender {
    CIFilter *filt = [filtersArray objectAtIndex:filterIndex];
    CGImageRef imgRef = [context createCGImagefilt.outputImage fromRect:
    [self.imageView setImage:[UIImage imageWithCGImage:imgRef]];
    CGImageRelease(imgRef);
    //NSLog(@"count %d", [filtersArray count]);
    filterIndex = (filterIndex + 1) % [filtersArray count];
    self.filterTitle setText:[filt attributes] valueForKey:@"CIAtribut
}

- (IBAction)autoEnhance:(id)sender {
    CIImage *outputImage = beginImage;
    NSArray *ar = [beginImage autoAdjustmentFilters];
    for (CIFilter *fil in ar) {
        [fil setValue:outputImage forKey:kCIInputImageKey];
        outputImage = fil.outputImage;
        NSLog(@"%@", [fil attributes] valueForKey:@"CIAtributeFilterDisp
    }
    [filter setValue:outputImage forKey:kCIInputImageKey];

    CGImageRef cgimg = [context createCGImageoutputImage fromRect:[output
    UIImage *newImage = [UIImage imageWithCGImage:cgimg scale:1.0 orientat
    self.imageView.image = newImage;
    CGImageRelease(cgimg);
}

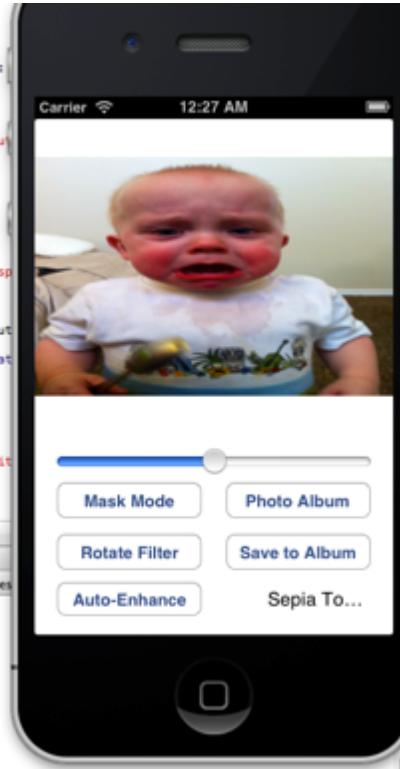
-(CIImage *)maskImage:(CIImage *)inputImage {
    CIFilter *maskFilter = [CIFilter filterWithName:@"CISourceAtopComposit
    kCIInputBackgroundImageKey, maskImage, nil];
    return maskFilter.outputImage;
}

CIAttributeIdentity = "(1 1 1 1)";
CIAttributeType = CIAtributeTypeColor;
};

inputImage = {
    CIAtributeClass = CIImage;
    CIAtributeType = CIAtributeTypeImage;
};

2>18-18 00:27:31.822 CoreImageFun[6404:c07] Vibrance
2>18-18 00:27:31.823 CoreImageFun[6404:c07] Tone Curve
2>18-18 00:27:31.823 CoreImageFun[6404:c07] Highlights and Shadows

```



You can see from the console the list of filters that auto-enhance has chosen to correct this image. This will change from image to image. Pretty easy!

Where to go from here?

You've seen that you can use Core Image filters to quickly apply a variety of effects to images. You can also use them to process real time video.

I covered the basics of how to set up and use core filters, but I hardly scratched the surface of what filters are available. There are filters that do a variety of photo editing tasks, including color, hue, brightness adjustments, scaling, skewing, rotating (affine transform), blurring and sharpening, and the standard set of image editing layer compositing filters (hard light, soft light, burn, etc).

There are more and more filters that will be available to the core image set and you can combine them together to get virtually any effect desired.

Also, as I mentioned in the previous chapter, you should check out our new book *iOS 6 by Tutorials*, that contains a "What's New with Core Image" chapter that

shows you how to dynamically apply Core Image filters to live video for some amazing effects!

Beyond that, there's some good info on Apple's website. For example, here's a list of all the Core Image filters with their functions. Not all of these are available on iOS, mind you!

- <http://developer.apple.com/library/mac/#documentation/GraphicsImaging/Reference/CoreImageFilterReference/Reference/reference.html>

I hope to see you make some cool apps that use Core Image! ☺

Chapter 22: UIViewController Containment

By Ray Wenderlich

Prior to iOS 5, you only had a few official choices for what to use as the root view controller for your app's Window:

- A `UINavigationController`
- A `UITabBarController`
- A `UISplitViewController`
- A plain `UIViewController`

In addition, there was no official way to make view controllers contain other view controllers inside them, like the first three of these controls do.

This was somewhat restrictive though, especially on the iPad. With the large screen, it's often nice to have different view controllers manage different areas. Plus, you might want to make a view controller "almost" like `UISplitViewController`, but with a wider left-hand side, or some other tweaks.

In the old days, people went ahead and wrote code to contain view controllers inside other view controllers anyway, but since there was no official way to do so, there were lots of subtle problems that arose. For example, sometimes the view lifecycle methods such as `viewWillAppear` or view controller rotation callbacks wouldn't be called at the proper times.

But as of iOS 5, those dark days are over! Now Apple has introduced an official API to use when you want to contain a view controller inside another. And as of iOS 6, Apple has introduced a way to do this straight from within the Storyboard editor, without even having to write any code!

In this tutorial, you are going to work on an app that is in the process of being ported from the iPhone to the iPad. The original iPhone version contains several view controllers that were on different screens, but you want to display these different view controllers on the same screen on the iPad.

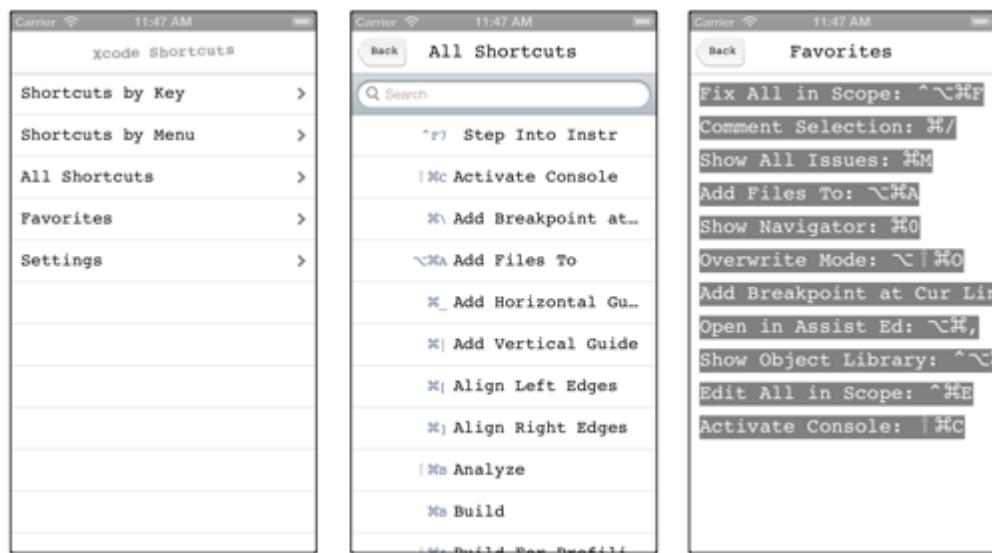
In the process of working on this app, you'll learn how to configure this project to use the new `UIViewController` containment APIs – and why you need them in the first place!

Note: This chapter has changed quite a bit since we first wrote *iOS 5 by Tutorials*. The sample project has been updated to use Storyboards and Modern Objective-C syntax, and the chapters has been simplified to keep the focus more strongly on UIViewController containment. In addition, it covers the new Container View that was introduced in iOS 6.

Introducing Xcode Shortcuts

In the resources for this chapter, you will find a sample project called **ShortcutsStarter**. This is a very simple app that lets you browse the shortcuts available in Xcode and add them to a list of Favorites you are trying to learn:

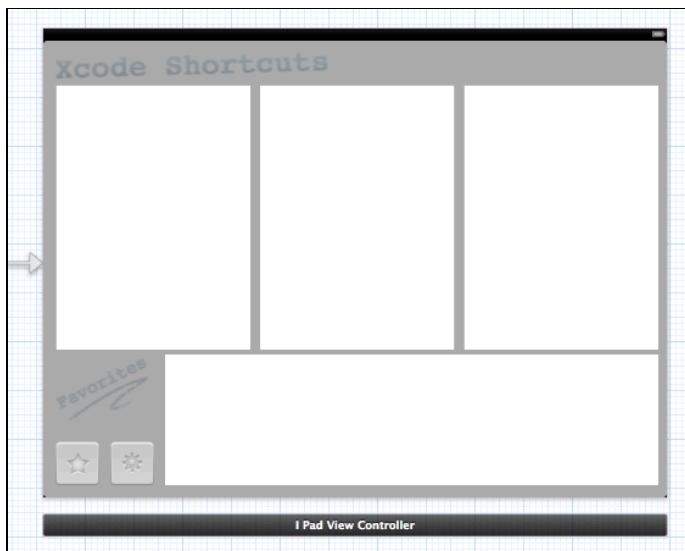
Run the app on your iPhone simulator, and you'll see the following:



As you try out the app, be sure to rotate the simulator left and right, and notice that the app handles orientation changes nicely. This is especially interesting in the Favorites screen, which reorganizes the shortcuts to take advantage of the extra space:



This app is currently in the process of being ported to the iPad. Open up **MainStoryboard-iPad.storyboard**, and you can see that the initial view controller is a screen with different panels where the various view controllers should go:



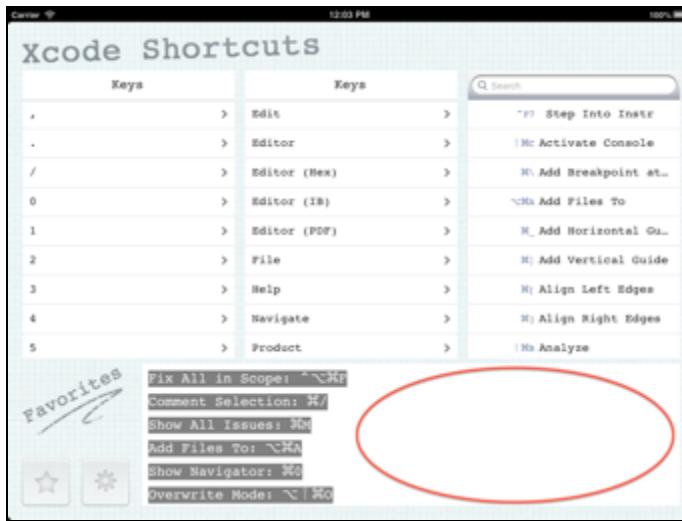
The view controller that corresponds to this is `iPadViewController`. Open **iPadViewController.m** and you'll see that each view is being filled with the contents of a view controller like this:

```
_navControllerLeft = [self.storyboard
    instantiateViewControllerWithIdentifier:@"ShortcutsNav"];
_navControllerLeft.view.frame = self.leftView.bounds;
// ...
[self.leftView addSubview:_navControllerLeft.view];
```

This is the old, pre-iOS 5 and 6 way of doing things – i.e. not using UIViewController containment. Basically in the old days, you would create a view controller, set the bounds of its view appropriately, and then add it as a subview to an existing view.

This actually works, sort of – but can introduce some subtle and hard to track down bugs, which is why you should use `UIViewController` containment instead. To see what I mean, run the app on the iPad simulator.

Make sure that the app starts in portrait mode, then rotate it to landscape. You'll notice that the favorites view does not automatically re-layout its labels to take advantage of the extra space:



Next set a breakpoint in `FavoritesViewController.m`'s `didRotateFromInterfaceOrientation:` method. This method is supposed to be called when a view controller switches orientation, and this view controller implements it to trigger a re-layout to take advantage of any extra space.

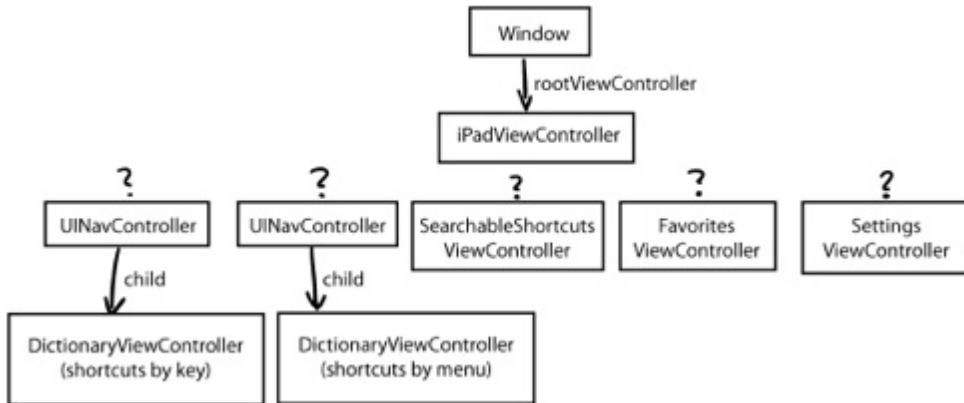
Run the app again, and rotate the app. You'll notice that it never gets called!

Wonder why this is? This leads into a discussion of view controller containment!

View controller containment overview

As you know, views are arranged in a hierarchy – every view has a parent and children. Here's what the view hierarchy looks like in `iPadViewController`:

It turns out that the same way views have parents, view controllers have parents as well. The only problem is, you haven't written any code to let the OS know that the `iPadViewController` should be the parent of your other view controllers, so there is no connection between them!



Because of this missing connection between view controllers, the OS doesn't know that it should pass on certain messages to child view controllers such as `didRotateFromInterfaceOrientation:`.

In fact, this is why you're having the problem you just identified. The favorites view controller does its relayout when the `didRotateFromInterfaceOrientation:` method is called. But since the view controller hierarchy isn't set up properly, this is never forwarded along!

There are workarounds to these issues that people have used in the past (such as manually forwarding on the `didRotateFromInterfaceOrientation:` calls, etc.), but these workarounds often don't catch every edge case and there are still subtle problems. It's better to let the OS handle it for you by setting up the view controller hierarchy properly with the view controller containment APIs!

View controller containment APIs

The view controller containment APIs are actually extremely simple. There are just five methods you need to know about:

- 1. `addChildViewController:`** - To designate a view controller as your child, you call this method. In your case, you want to call this from `iPadViewController` on all of the child view controllers within (the two navigation controllers, the searchable shortcuts view controller, and whichever of the favorites/settings is currently visible). Once the OS knows the hierarchy, it can properly forward the lifecycle and rotation methods.
- 2. `removeFromParentViewController:`** - To remove a view controller from your list of children, you call this method. This is handy when you remove a view controller from the screen, like this app does when you swap between the favorites and settings view controllers.
- 3. `transitionFromViewController:toViewController:duration:options:animations:completion:`** - There's a similar API to this for replacing one view with another that you can see in the current `favoritesTapped:` and `settingTapped:`

methods, but if the views come from view controllers, you should use this one instead so the messages are forwarded properly.

4. **willMoveToParentViewController:** - This is automatically called by `addChildViewController:`, so you don't usually need to call this directly. However, inside a view controller, if you ever need to have certain code happen when a view moves to a parent view controller (or out of one), this is a good method to override.
5. **didMoveToParentViewController:** - It's your responsibility to call this after you call `addChildViewController:`. If there's no animation, you can call it right away, but if there's an animation you should wait until the animation completes.

The most important thing to remember about these is that you need to:

- Call `addChildViewController:` as soon as you add the view controller's view to your window hierarchy
- Call `didMoveToParentViewController:` once it animates in.

This will have the effect of calling `viewWillAppear` on your view controller - even if it's covered up by another view! What `viewWillAppear` really means is the view has made its way into the window's hierarchy, so design with that in mind.

Let's try this out with a simple case – setting up the view controller hierarchy associated with the left view appropriately. Add these two lines to the bottom of **iPadViewController.m**'s `viewDidLoad`:

```
[self addChildViewController:_navControllerLeft];
[_navControllerLeft didMoveToParentViewController:self];
```

Wow, that was easy eh? Build and run, and everything will work as usual – but you can rest peacefully knowing that your left side view controller hierarchy is OK!



Now let's try a slightly more complicated case that shows you how to use some of the other methods – the case where you swap out the favorites view controller and the settings view controller.

For this app, you'll only add the view controller to the hierarchy when it's active/visible, and remove it otherwise. The app starts with the favorites view controller active, so add this to the bottom of `viewDidLoad`:

```
[self addChildViewController:_favorites];
```

```
[_favorites didMoveToParentViewController:self];
```

That's the same as you did before. Next, replace the favoritesButtonTapped: and settingsButtonTapped: with the following:

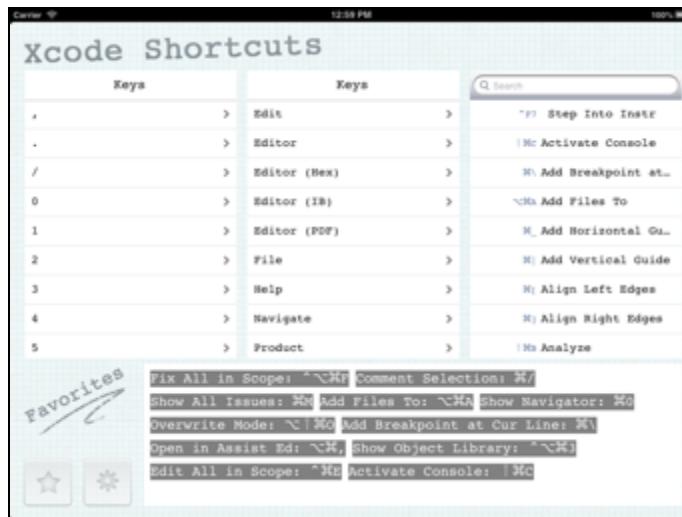
```
- (IBAction)favoritesButtonTapped:(id)sender {
    [[ShortcutsDatabase sharedDatabase] playClick];
    // 1
    [self addChildViewController:_favorites];
    // 2
    [self transitionFromViewController:_settings
        toViewController:_favorites
        duration:0.5
        options:
        UIViewAnimationOptionTransitionFlipFromBottom
        animations:^{
            // 3
            [_settings.view removeFromSuperview];
            _favorites.view.frame = self.bottomView.bounds;
            [self.bottomView addSubview:_favorites.view];
        } completion:^(BOOL finished) {
            // 4
            [_favorites didMoveToParentViewController:self];
            [_settings removeFromParentViewController];
        }];
}

- (IBAction)settingsButtonTapped:(id)sender {
    [[ShortcutsDatabase sharedDatabase] playClick];
    [self addChildViewController:_settings];
    [self transitionFromViewController:_favorites
        toViewController:_settings
        duration:0.5
        options:
        UIViewAnimationOptionTransitionFlipFromBottom
        animations:^{
            [_favorites.view removeFromSuperview];
            _settings.view.frame = self.bottomView.bounds;
            [self.bottomView addSubview:_settings.view];
        } completion:^(BOOL finished) {
            [_settings didMoveToParentViewController:self];
            [_favorites removeFromParentViewController];
        }];
}
```

These two methods are almost exactly the same – except one flips from settings to favorites, and the other the opposite. Let's look at the `favoritesButtonTapped:` method step by step:

1. You're about to display the favorites view controller, so add it as a child view controller to get it into the hierarchy.
2. Instead of calling the old `transitionWithView:duration:options:animations:completion:` method, you call this new variant instead that deals with two view controllers. It allows you to specify the two view controllers you're switching between and details about the animation.
3. In the animation block you make your manipulations to the view hierarchy – i.e. removing the setting view controller's view and adding in the favorites view. Note that you have to manage both the view hierarchy and the view controller hierarchy.
4. Finally, when the animation completes, that is where you have to call `didMoveToParentViewController:` on the favorites view controller that just appeared, and `removeFromParentViewController` on the settings view controller that just disappeared.

Pretty easy, eh? Build and run the app and perform the same test you did earlier – start the app in portrait mode and rotate to landscape. You'll see now the favorites view controller properly resizes! This is because `didRotateToInterfaceOrientation:` is now called because the view controller hierarchy is properly set up.



Container views

Now you know how to use the view controller containment APIs programmatically, but it would be annoying if you had to always had to manage everything in code like this. Wouldn't it be better if somehow you could set up container views in the Storyboard editor without having to write any code?

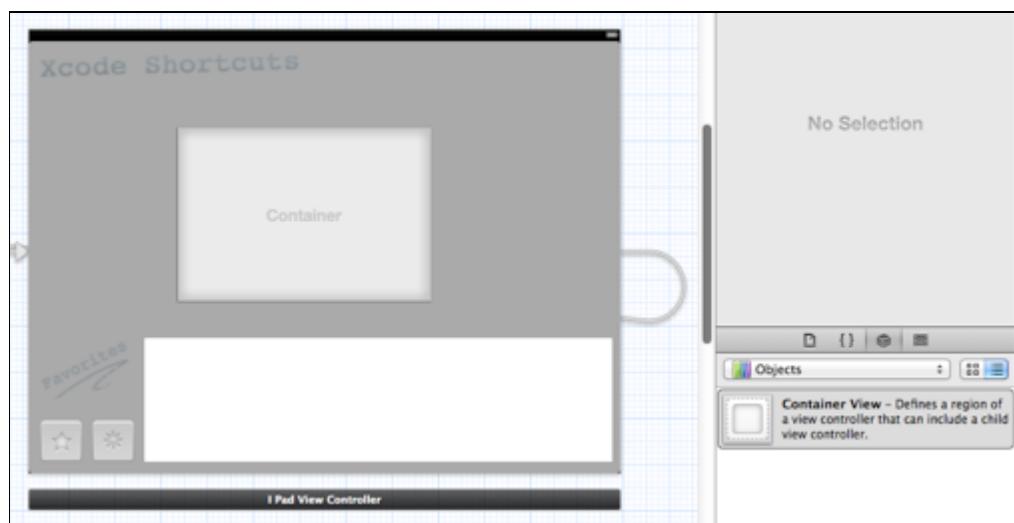
As of iOS 6, now you can, the the new Container Views! These are special views you can drag into your Storyboards, that can configure to display the contents of another view controller in that spot. It saves you a ton of code, and makes your storyboard easier to understand too.

Let's modify the project to use the new iOS 6 Container views for the top three views instead of manually loading the view controllers and adding them as subviews.

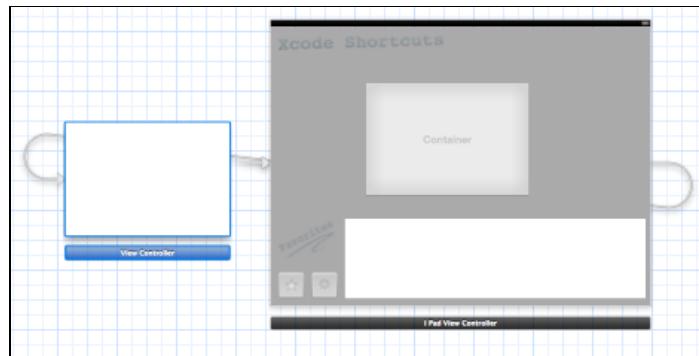
Open **MainStoryboard-iPad.storyboard** and delete the 3 top view placeholders like so:



Then in your object library panel, look for the new Container View, and drag it onto the view controller (doesn't matter where at this point):



You'll notice that it has automatically made a special segue to embed a plain view controller inside that space. You don't need that, so select the plain view controller and delete it.

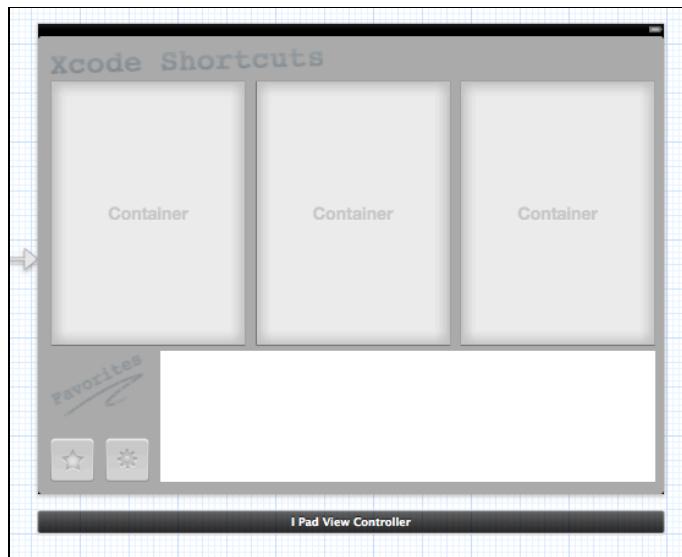


After deleting the plain view controller, select the Container View and in the Size Inspector set X=21, Y=73, Width=319, and Height=433.

Now repeat this for two more Container Views. Drag them in, delete the default view controller it adds, and set their positions/sizes like the following:

- Middle: X=356, Y=73, Width=319, Height=433.
- Right: X=691, Y=73, Width=319, Height=433.

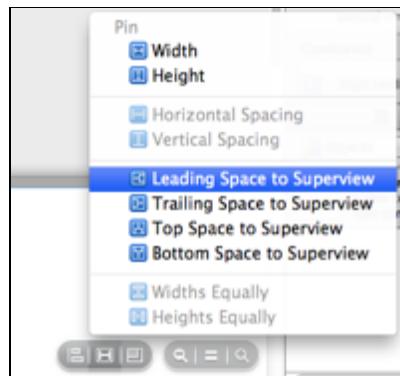
At this point, your view controller should look like this:



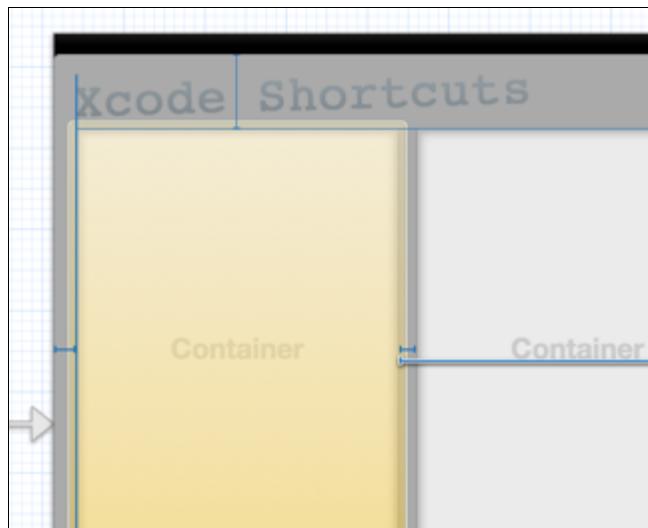
Next it's time for some Auto-Layout practice! Your goal is to configure the auto layout settings for the three container views so they stay the same width, but grow or shrink to the available width/height in portrait and landscape.

To do this, perform the following steps:

- Select the leftmost container view, and from menu at the bottom right of the Storyboard select Pin\Leading Space to Superview.

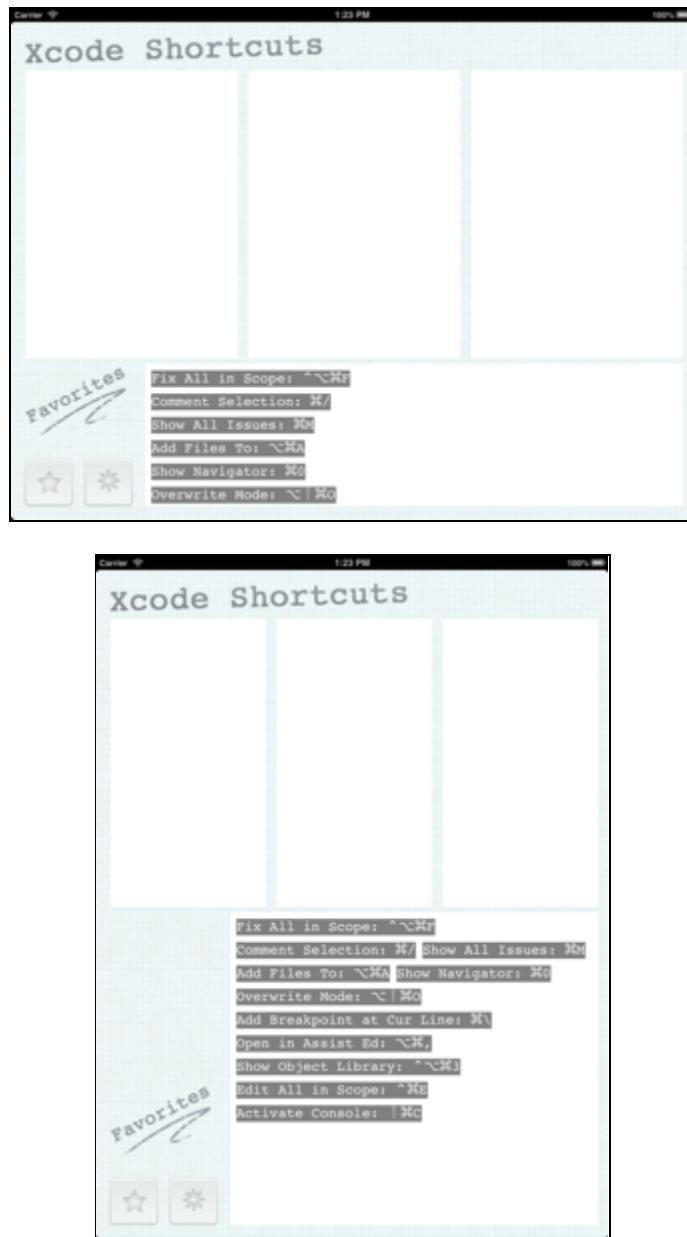


- Select both the left and middle container views at the same time (you can select two views by clicking the first, holding down command, and then clicking the second). Then from the menu select Pin\Horizontal Spacing.
- Select both the middle and right container views, and again select Pin\Horizontal Spacing.
- Select the right container view, and select Pin\Trailing Space to Superview.
- Select all three container views and select Pin\Widths Equally.
- Click on each view controller one at a time and verify that there is a line in-between the small gap between each view controller and its predecessor/successor (or the main window itself). If something is missing, try re-adding it.
- Next click on each view controller and delete any constraints that are not the aforementioned “gap constraints.” For example, in this screenshot you don’t need the highlighted constraint:



- When you’re done, click each container view and make sure it’s clean and just the needed “gap constraints” are there. Also make sure at least one of the container views has a constraint that connects it to the top of the main view, and at least one has a constraint that connects it to the bottom view.

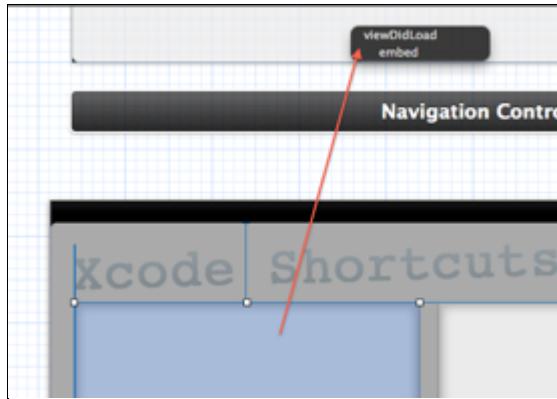
When you think you've got it, build and run and verify that it rotates and resizes OK in both portrait and landscape:



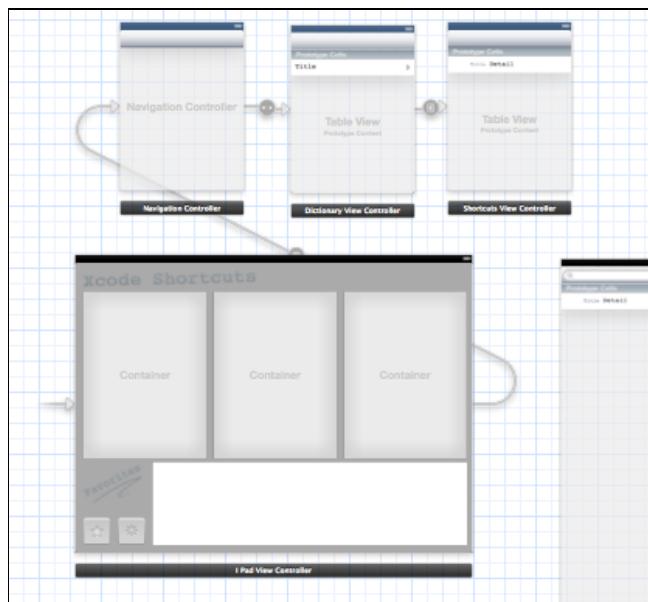
Note: If you get stuck, check out Chapter 3 in *iOS 6 by Tutorials*, "Beginning Auto Layout."

Now that you have your container views in the right spots, let's hook them up to the appropriate view controllers!

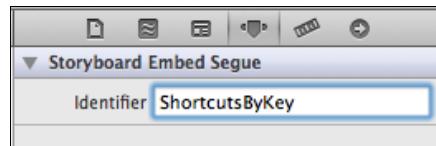
This is incredibly easy. Select the first container view, and control-drag to the Navigation controller right above it, and select **embed** from the drop down:



You'll notice that after you do this, it automatically resizes the navigation controller and its child view controllers to be the same size as the container view. Pretty cool, eh?



You'll also notice a line connecting the container and its child view controller – it created a segue for you. You're going to need to use this in code, so select the segue and give it an identifier "ShortcutsByKey":



Now repeat this for the other two container views:

- Connect the middle container view to the navigation controller as well, but name the segue "ShortcutsByMenu."
- Connect the right container view to the `SearchableShortcutsViewController`, and name the segue "AllShortcuts."

Build and run, and you'll see that something shows up in the right spots now – however everything's empty:



This is because before you passed some initial data to these view controllers – but now that you're creating the view controllers completely within the Storyboard editor, you haven't had a chance to pass the initial data.

But as you may have guessed, that's exactly what you'll do next! ☺

Setting up embedded view controllers

If you've done much work with the Storyboard editor, you'll know that when you want to pass data from one view controller to another, you can override `prepareForSegue:` to get a pointer to the destination view controller and set data before it is displayed.

Note: Not ringing any bells? Check out Chapter 4 and 5 in this book, "Beginning and Intermediate Storyboards".

You'll be pleased to know that you can use the exact same technique for Container Views! Simply add this new method to `iPadViewController.m`:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue  
sender:(id)sender {  
    if ([segue.identifier isEqualToString:@"ShortcutsByKey"]) {  
        [[ShortcutsDatabase sharedDatabase] playClick];  
        UINavigationController * navController =  
            (UINavigationController *)  
            segue.destinationViewController;  
        DictionaryViewController * dictionaryViewController =
```

```
        navController.viewControllers[0];
        dictionaryViewController.navigationController.title =
            @"Keys";
        dictionaryViewController.dict =
            [ShortcutsDatabase sharedDatabase].shortcutsByKey;
    } else if ([segue.identifier
        isEqualToString:@"ShortcutsByMenu"]) {
        [[ShortcutsDatabase sharedDatabase] playClick];
        UINavigationController * navController =
            (UINavigationController *)
            segue.destinationViewController;
        DictionaryViewController * dictionaryViewController =
            navController.viewControllers[0];
        dictionaryViewController.navigationController.title =
            @"Menus";
        dictionaryViewController.dict =
            [ShortcutsDatabase sharedDatabase].shortcutsByMenu;
    } else if ([segue.identifier
        isEqualToString:@"AllShortcuts"]) {
        [[ShortcutsDatabase sharedDatabase] playClick];
        SearchableShortcutsViewController *
            shortcutsViewController =
            (SearchableShortcutsViewController *)
            segue.destinationViewController;
        shortcutsViewController.navigationItem.title =
            @"All Shortcuts";
        shortcutsViewController.shortcutsDict =
            [ShortcutsDatabase sharedDatabase].shortcutsByKey;
    }
}
```

If you've overridden `prepareForSegue:` before, this should look familiar to you. It checks to see what segue is being performed, pulls out the view controller that is being displayed, and sets up the initial values.

As a final cleanup step, you might want to delete the old instance variables for `_navControllerLeft`, `_navControllerMiddle`, and `_allShortcuts` as well as the code that deals with them in `viewDidLoad`, since you are not using those anymore.

And that's it – build and run, and your app now uses view controller containment for the top three view controllers – with barely any code!



Where to go from here?

As you can see, using view controller containment is pretty easy these days, between the new iOS 5 APIs and the iOS 6 container views.

If you want to learn more about container views in iOS 6, check out Chapter 21 in *iOS 6 by Tutorials*, "What's New with Storyboards", which goes a bit deeper into the topic.

Using view controller containment will allow you to create some neat user interfaces and re-use your existing view controllers between apps easily.

And the best part is you're no longer officially restricted to using the built-in container view controllers such as `UINavigationController`, `UITabBarController`, or `UISplitViewController` in your apps. Now you can design your own without having to worry about subtle issues, and the sky's the limit!

Chapter 23: Working with JSON

By Marin Todorov

JSON is a simple human readable format that is often used to send data over a network connection.

For example, if you have an array of three strings, the JSON representation would simply be:

```
[ "test1", "test2", "test3" ]
```

That is actually pretty similar to declaring a new array in modern Objective-C:

```
@[@"test1", @"test2", @"test3"]
```

If you have a Pet object with member variables `name`, `breed`, and `age`, the JSON representation would simply be:

```
{"name" : "Dusty", "breed": "Poodle", "age": 7}
```

Again very, very similar to the corresponding Objective-C code:

```
@{@"name" : @"Dusty", @"breed": @"Poodle", @"age": @7}
```

It's that simple, which is why it's so easy and popular to use. For the full spec, which can be read in just a couple minutes, check out <http://www.json.org>.

The reason JSON is important is that many third parties such as Google, Yahoo, or Kiva make web services that return JSON formatted data when you visit a URL with a specified query string. If you write your own web service, you'll also probably find it really easy to convert your data to JSON when sending to another party.

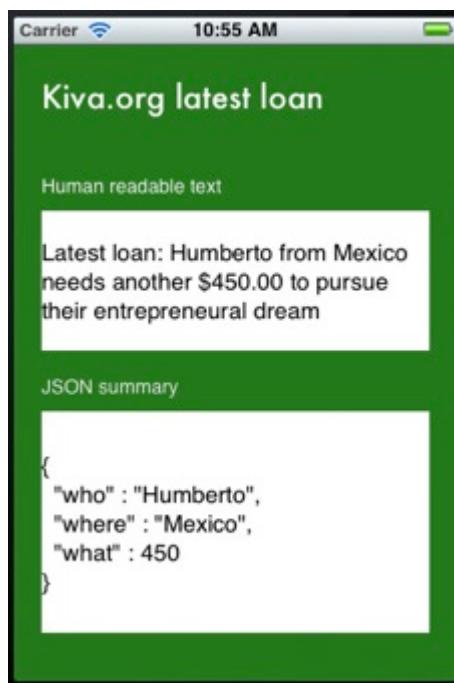
JSON and iOS 5

If you've had to parse JSON in your iOS apps in the past, you've probably used a third party library such as the JSON library.

Well with iOS 5, needing to use a third-party library for JSON parsing is a thing of the past. Apple has finally added a JSON library in Cocoa and I must say I personally like it very much!

You can turn objects like `NSString`, `NSNumber`, `NSArray` and `NSDictionary` into JSON data and vice versa super easily. And of course no need to include external libraries - everything is done natively and super fast.

In this chapter you are going to get hands-on experience with the new native JSON support. You are going to build a simple application, which will connect to the Internet and consume JSON service from the Kiva.org web site. It will parse the data, show it in human readable format on the screen, and then build back a different JSON.



Later on in the chapter you are to create a class category, which will give you ideas how to integrate JSON support more tightly into Cocoa and your own classes. Having the possibility to turn your own classes data into JSON could really help you persist data structures online, exchange data between applications, or anything that requires your classes to be able to serialize and persist data, which can be sent over http, email, etc.

Getting Started

Open up Xcode and from the main menu choose **File\New\New Project**. Choose the **iOS\Application\Single View Application** template, and click *Next*. Name the product **KivaJSONDemo**, select **iPhone** for the Device family, and make sure just the **Use Automatic Reference Counting** checkbox is checked, make sure

Use Storyboards checkbox is checked, click *Next* and save the project by clicking *Create*.

Disable the landscape mode (you're not going to design all the Auto Layout constraints in this tutorial to handle orientation changes) from the project settings that popup after you create the project:



First you are going to do a little clean up - open up **ViewController.m** file and replace everything inside with this:

```
#define kBgQueue dispatch_get_global_queue(  
DISPATCH_QUEUE_PRIORITY_DEFAULT, 0) //1  
  
#define kLatestKivaLoansURL [NSURL URLWithString:  
@"http://api.kivaws.org/v1/loans/search.json?status=fundraising"]  
//2  
  
#import "ViewController.h"  
  
@interface ViewController()  
@end  
  
@implementation ViewController  
@end
```

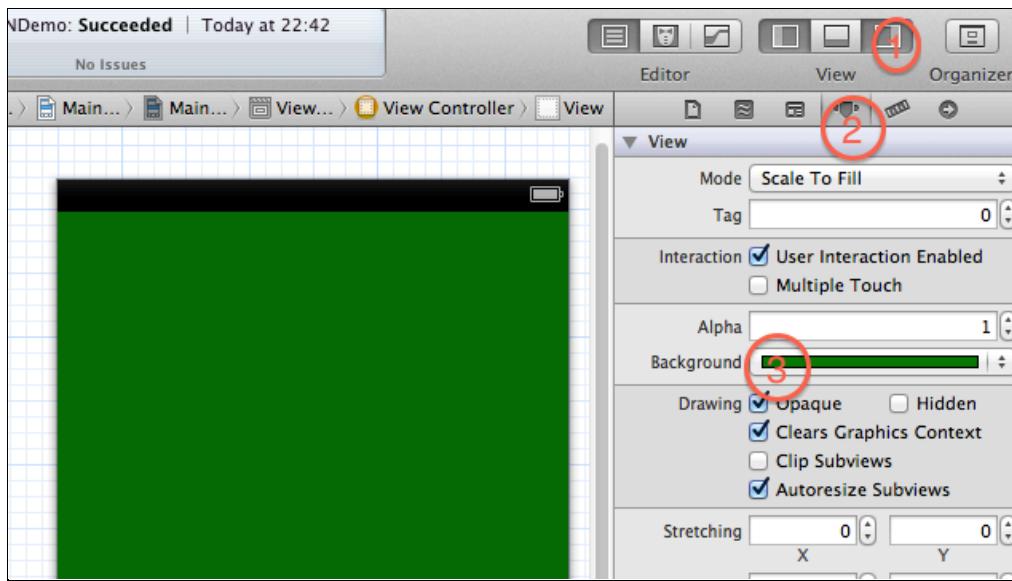
In the first line of code you define a macro that gives you back a background queue. (It is nice having a `kBgQueue` shortcut for that – keeps writing code simpler.)

In the second line of code you create a macro named `kLatestKivaLoansURL`, which returns you an `NSURL` pointing to this URL [<http://api.kivaws.org/v1/loans/search.json?status=fundraising>].

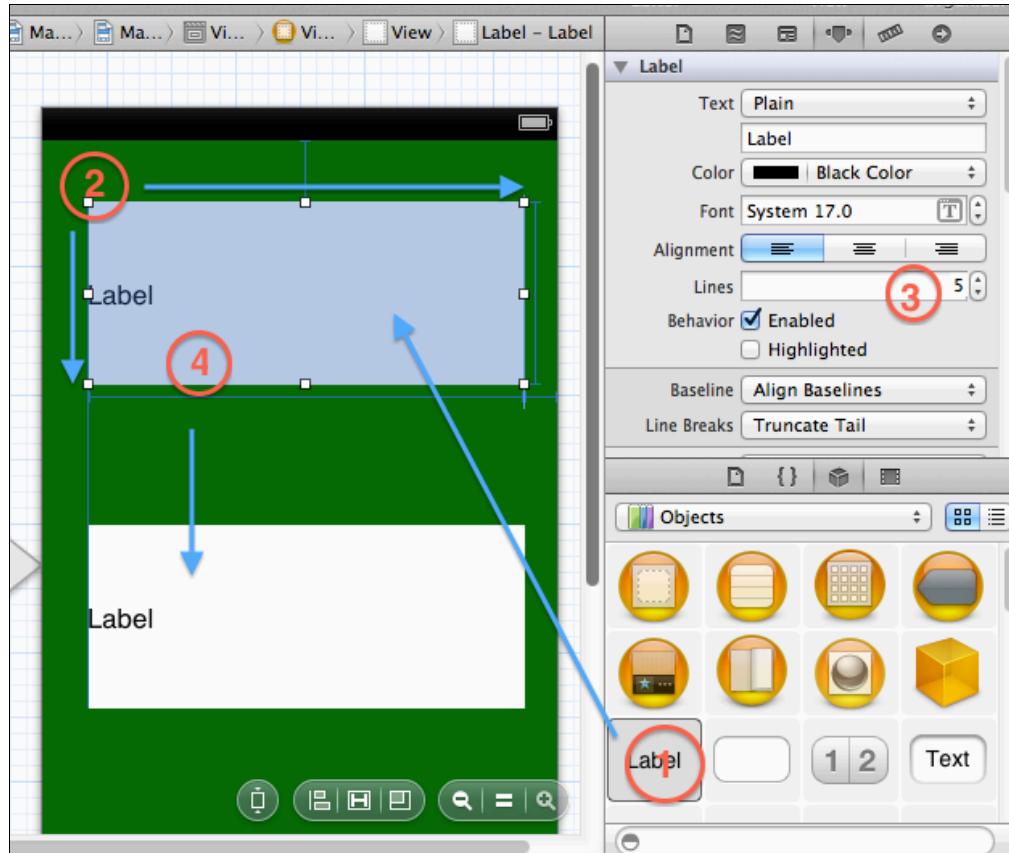
Go ahead and visit this URL in your browser - you'll see [Kiva.org](http://kiva.org)'s list of currently fundraising loans in JSON format. You are going to use this API to read the list of loans, take the latest one and show the information on the screen. (No worries – the JSON data you see in the browser window is optimized for size, so it probably looks like gibberish.)

Let's make a little detour and design the application's UI in Interface Builder real quick.

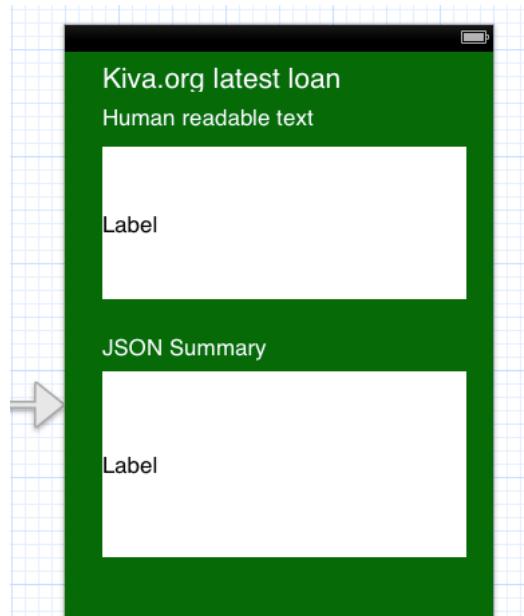
Open **MainStoryboard.storyboard** up in the Project Navigator. This app is supposed to be positive and life changing, so you need to do something about that background! Select the view controller of the only screen in the storyboard, and from the Utilities bar (1), make sure you have the Attributes Inspector open (2), and set the Background to a nice green color (3).



Now grab a label from the Object library and drop it inside the already open view (1). Resize the label so it fits about 4 lines of text and takes almost the screen's width (2). Then from the Attributes Inspector make the following changes: set "Background" to white, "Color" to black, set "Lines" to "5" (3). Click on the label to make sure it's selected and then press Cmd+C, Cmd+V to clone the label (4). Finally arrange the two labels like on the screenshot:



To polish up the interface add 3 more labels and finish the UI so it looks like this:



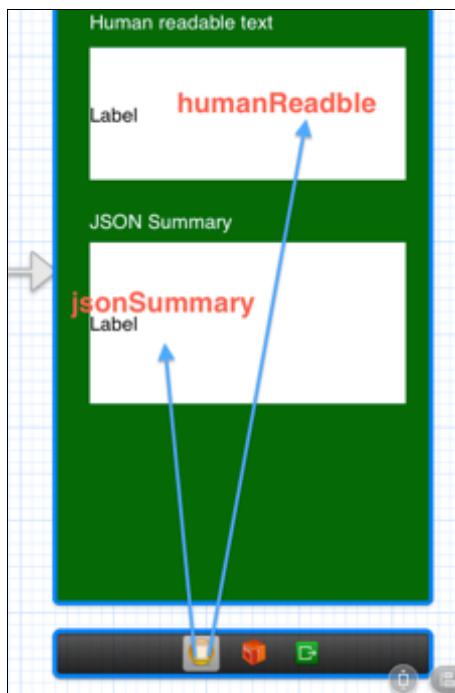
The only thing left is to connect the labels to a couple of `IBOutlet`s in your class.

Switch to **ViewController.m** and add two instance variables inside the interface like so:

```
@interface ViewController () {  
    IBOutlet UILabel* humanReadable;  
    IBOutlet UILabel* jsonSummary;  
}  
@end
```

Then open up **MainStoryboard.storyboard** again. Control-drag from **View Controller** onto the 1st 5-line label and from the popup menu and choose **humanReadable**.

Again while holding the "ctrl" key drag with the mouse from **View Controller** onto the 2nd 5-line label and from the popup menu and choose **jsonSummary**.



That concludes the project setup - you're ready to start coding!

Parsing JSON from the Web

The first thing you need to do is download the JSON data from the web. Luckily, with GCD you achieve this in one line of code! Add the following to **ViewController.m**:

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

```
dispatch_async(kBgQueue, ^{
    NSData* data = [NSData dataWithContentsOfURL:
        kLatestKivaLoansURL];

    [self performSelectorOnMainThread:@selector(fetchedData:)
        withObject:data waitUntilDone:YES];
});
}
```

Remember how earlier you defined `kBQqueue` as a macro, which gives a background queue?

Well this bit of code makes it so that when `viewDidLoad` is called, you run a block of code in this background queue to download the contents from the Kiva loans URL.

When `NSData` has finished fetching data from the Internet you call `performSelectorOnMainThread:withObject:waitUntilDone:` so you can update the application's UI. You still haven't written `fetchedData:` yet but will do so shortly.

Remember that it is only OK to run a synchronous method such as `dataWithContentsOfURL` in a background thread, otherwise the GUI will seem unresponsive to the user.

Also, remember that you can only access UIKit objects from the main thread, which is why you had to run `fetchedData:` on the main thread.

Note: You might wonder why I preferred to use `performSelectorOnMainThread:withObject:waitUntilDone:` over dispatching a block on the main thread? It's a personal preference really and I have two reasons:

1) I'm all for the greatest readability of a piece of code. For me, `[self performSelectorOnMainThread:...]` makes it easier to spot what's going on in that piece of code.

2) I'm a symmetry freak! I find that Xcode doesn't handle text indentation well when you use `dispatch_async()`, so purely visually the code is not so pleasant to look at. You might have other preferences, so yes - if you prefer `dispatch_async(dispatch_get_main_queue(), ^(){...});` go for it!

So, when the data has arrived the method `fetchedData:` will be called and the `NSData` instance will be passed to it. In your case the JSON file is relatively small so you're going to do the parsing inside `fetchedData:` on the main thread. If you're parsing large JSON feeds (which is often the case), be sure to do that in the background.

So next add the `fetchedData` method to the file:

```

- (void)fetchedData:(NSData *)responseData {
    //parse out the json data
    NSError* error;
    NSDictionary* json = [NSJSONSerialization
        JSONObjectWithData:responseData //1

        options:kNilOptions
        error:&error];

    NSArray* latestLoans = json[@"loans"]; //2
    NSLog(@"loans: %@", latestLoans); //3
}

```

This is it - the new iOS 5 JSON magic!

Basically iOS 5 has a new class named `NSJSONSerialization`. It has a static method called `JSONObjectWithData:options:error` that takes an `NSData` and gives you back a Foundation object - usually an `NSDictionary` or an `NSArray` depending what do you have at the top of your JSON file hierarchy.

In Kiva.org's case at the top there's a dictionary, which has a key with list of loans. You get an `NSArray` – `latestLoans`, which is the `loans` key in the top JSON dictionary.

Finally in line //3, you dump `latestLoans` to the console, so you can make sure everything's OK. Hit Run and check Xcode's console to see the result:

```

id = 877228;
"template_id" = 1;
};
"loan_amount" = 550;
location =
{
    country = Nicaragua;
    "country_code" = NI;
    geo =
    {
        level = town;
        pairs = "12.435556 -86.879444";
        type = point;
    };
    town = Leon;
};
name = "Luz Marina Hernandez Ordoñez";
"partner_id" = 96;
"posted_date" = "2011-09-24T03:50:05Z";
sector = Agriculture;
status = fundraising;
use = "to buy pigs.";

```

Not bad for 3 lines of code, eh? :]

Parsing Options

I'd like to talk just a bit more about `NSJSONSerialization's` `JSONObjectWithData:options:error:` method. It's one of these Apple APIs, which

understand and do everything by themselves, but you still can configure a bit its behavior.

Notice in the code that I chose to pass for the parameter `options` a value of `kNilOptions`. `kNilOptions` is just a constant for 0 - I find its name very descriptive though, so I always prefer it over just the value of 0 as a method parameter.

However you can pass other values or even a bit mask of values to combine them. Have a look at what you got as options when you're converting JSON to objects:

- **NSJSONReadingMutableContainers**: The arrays and dictionaries created will be mutable. Good if you want to add things to the containers after parsing it.
- **NSJSONReadingMutableLeaves**: The leaves (i.e. the values inside the arrays and dictionaries) will be mutable. Good if you want to modify the strings read in, etc.
- **NSJSONReadingAllowFragments**: Parses out the top-level objects that are not arrays or dictionaries.

So, if you're not only reading, but also modifying the data structure from your JSON file, pass the appropriate options from the list above to `JSONObjectWithData:options:error:`.

Displaying on the Screen

You are going to continue by showing the latest loan information on the screen. At the end of "fetchedData:" method add these few lines of code:

```
// 1) Get the latest loan
NSDictionary* loan = latestLoans[0];

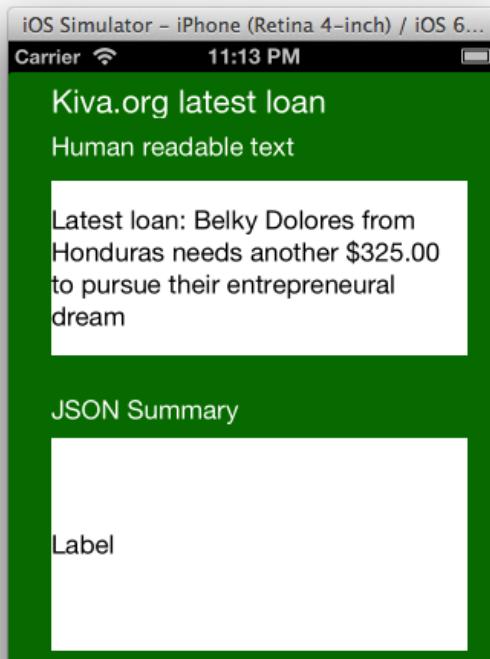
// 2) Get the funded amount and loan amount
NSNumber* fundedAmount = loan[@"funded_amount"];

NSNumber* loanAmount = loan[@"loan_amount"];
float outstandingAmount = [loanAmount floatValue] -
    [fundedAmount floatValue];

// 3) Set the label appropriately
humanReadable.text = [NSString stringWithFormat:@"Latest loan: %@
from %@ needs another $%.2f to pursue their entrepreneurial dream",
    loan[@"name"],
    loan[@"location"][@"country"],
    outstandingAmount
];
```

The `latestLoans` array is a list of dictionaries, so (1) you get the first (and latest) loan dictionary and (2) you fetch few values about the loan. Finally (3) you set the text of the 1st label in the UI.

OK! Let's have a look - hit Run and see what comes up:



It may take a while until your device fetches the JSON feed (depending on your current connection speed), but just wait few seconds and you will see the text appearing in the top label.

Of course the information you see will be different as Kiva adds loans constantly - but it's clear that you have achieved what was wanted: you have parsed the JSON data and visualized some human readable info.

Generating JSON Data

Now let's do the opposite. From the `loan` dictionary that you now have you will build some JSON data, which you will be able to send over to a server, another app, or do with it whatever else you want.

Add this code to the end of the `fetchedData:` method:

```
//build an info object and convert to json
NSDictionary* info = @{@"who": loan[@"name"],
 @"where": loan[@"location"][@"country"],
 @"what": [NSNumber numberWithFloat: outstandingAmount]};
```

```
//convert object to data
NSData* jsonData = [NSJSONSerialization dataWithJSONObject:info
options:NSJSONWritingPrettyPrinted error:&error];
```

Here you build an `NSDictionary` called `info` where you store the loan information as `who`, `where`, and `what` in different keys and values.

Then you call `dataWithJSONObject:options:error:` the opposite to the JSON API you just used before. It takes in an object and turns it into JSON data.

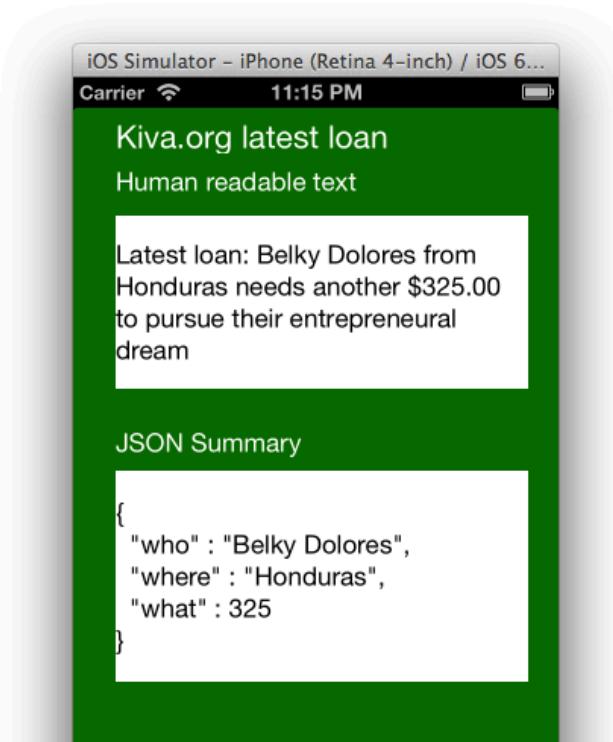
For the options parameter there's only one possible value -

`NSJSONWritingPrettyPrinted`. If you want to send the JSON over the Internet to a server use `kNilOptions` as this will generate compact JSON code, and if you want to see the JSON use `NSJSONWritingPrettyPrinted` as this will format it nicely.

So, at this point your task of turning `info` into JSON is finished, but you can't be sure before you see that it is actually so. Let's show the JSON into the second UI label. Add this final line of code to `fetchedData`:

```
//print out the data contents
jsonSummary.text = [[NSString alloc] initWithData:jsonData
encoding:NSUTF8StringEncoding];
```

By initializing an `NSString` with `initWithData:encoding:` you easily get the text representation of your JSON data and you show it straight inside the `jsonSummary` label. Hit Run and:



Integrating Objects and JSON

Imagine if `NSDictionary`, `NSArray`, `NSString`, and `NSData` had methods to convert to and from JSON data - wouldn't that be great?

Oh, but wait - what you are using it's Objective-C, so you can actually extend foundation classes with methods of our own! Let's do an example with `NSDictionary` and see how useful that could be.

Open **ViewController.m**, and add this category just above the `@implementation`:

```
@interface NSDictionary(JSONCategories)
+(NSDictionary*)dictionaryWithContentsOfJSONURLString:
    (NSString*)urlAddress;
-(NSData*)toJSON;
@end

@implementation NSDictionary(JSONCategories)
+(NSDictionary*)dictionaryWithContentsOfJSONURLString:
    (NSString*)urlAddress
{
    NSData* data = [NSData dataWithContentsOfURL: [NSURL
        URLWithString: urlAddress] ];
    __autoreleasing NSError* error = nil;
```

```

    id result = [NSJSONSerialization JSONObjectWithData:data
        options:kNilOptions error:&error];
    if (error != nil) return nil;
    return result;
}

-(NSData*)toJSON
{
    NSError* error = nil;
    id result = [NSJSONSerialization dataWithJSONObject:self
        options:kNilOptions error:&error];
    if (error != nil) return nil;
    return result;
}
@end

```

As there's nothing new that I didn't speak about so far in this tutorial I won't go over the code line by line.

Basically, you add 2 methods to `NSDictionary`: one `dictionaryWithContentsOfJSONURLString`: which gets an `NSString` with a web address (it's often easier to work with URLs as text, not as `NSURL` instances), does all the downloading, fetching, parsing and whatnot and finally just returns an instance of a dictionary (or `nil` in case of an error) - ain't that pretty handy?

In the category there's also one more method – `toJSON`, which you call on an `NSDictionary` instance to get JSON data out of it.

So with this category fetching JSON from the web becomes as easy as:

```

NSDictionary* myInfo =[NSDictionary
    dictionaryWithContentsOfJSONURLString:
    @"http://www.yahoo.com/news.json"];

```

And of course on any of your `NSDictionary` objects you can do:

```

NSDictionary* information =
@{@"orange":@"apple",@"banana":@"fig"};

NSData* json = [information toJSON];

```

Pretty cool and also quite easily readable code. Now of course you can also extend `NSMutableDictionary` with the same `dictionaryWithContentsOfJSONURLString`: method, but in there you'll have to pass `NSJSONReadingMutableContainers` as options - so hey, `NSMutableDictionary` could be initialized with JSON too, and it'll hold mutable data. Cool!

Where to Go From Here?

At this point, you have hands-on experience with the awesome new iOS5 JSON reading and writing APIs, and are ready to start using this in your own apps!

Before wrapping-up I'd like to mention just few more methods from the `NSJSONSerialization` class.

```
BOOL isTurnableToJson = [NSJSONSerialization isValidJSONObject:  
object]
```

As you might guess, `isValidJSONObject:` tells you whether you can successfully turn a Cocoa object into JSON data.

Also I presented to you the 2 methods to read and write JSON from/to `NSData` objects, but you can do that also on streams - with `JSONObjectWithStream:options:error:` and `writeJSONObject:toStream:options:error:`, so do have a look at the class documentation.

If you want to keep playing around with JSON, feel free to extend the demo project with the following features:

- Modify the demo project to use the JSON categories, like we discussed above
- Develop further JSON categories for `NSArray`, `NSString`, etc
- Think about how cool it'd be if your own classes had a `toJSON` method - so you can easily persist them on your web server! Make an implementation on a test class to see if you can get it working!

Chapter 24: UIKit Particle Systems

By Marin Todorov

You've probably seen particle systems used in many different iOS apps and games for explosions, fire effects, rain or snow falling, and more. However, you probably saw these types of effects most often in games, because UIKit didn't provide a built-in capability to create particle systems - until iOS 5, that is!

Now with iOS 5, you can use particle systems directly in UIKit to bring a lot of exciting new eye-candy to your apps. Here are a few examples of where particle systems could be useful:

- **UIKit games:** Yes, you can make games with plain UIKit (and some types of games work really well there, particularly card games and the like). But now, you can make them even better with explosions, smoke, and other goodies!
- **Slick UI effects:** When your user moves around an object on the screen it can leave a trail of smoke, why not?
- **Stunning screen transitions:** How about presenting the next screen in your app while the previous one disappears in a ball of fire?

Hopefully you've got some cool ideas of what you might want to use UIKit particle systems for. So let's go ahead and try it out!

In this tutorial, we're going to develop an app called "Draw with fire" that lets you (you guessed it) draw with fire on the screen.

I'll take you along the process of creating the particle systems and having everything set on the screen, and you can develop the idea further into your own eye-candied drawing application. When the app is ready, you'll be able to use it to draw a nice question mark of fire - like this one:



The New Particle APIs

The two classes you will need to use in order to create particle systems are located in the **QuartzCore** framework and are called `CAEmitterLayer` and `CAEmitterCell`.

The general idea is that you create a `CAEmitterLayer`, and add to it one or more `CAEmitterCells`. Each cell will then produce particles in the way it's configured.

Also, since `CAEmitterLayer` inherits from `CALayer`, you can easily inject it anywhere in your UIKit hierarchy!

I think the coolest thing about the new UIKit particle systems is that a single `CAEmitterLayer` can hold many `CAEmitterCells`. This allows you to achieve some really complicated and cool effects. For example, if you're creating a fountain you can have one cell emitting the water and another emitting the vapor particles above the fountain!

Getting Started

Fire up Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Single View Application** template and click *Next*. Enter **DrawWithFire** for the product name, enter **DWF** for the class prefix, select iPhone for the Device Family, and make sure that **Use automatic reference counting**

and **Use Storyboards** are checked (leave the other checkboxes unchecked). Click *Next* and save the project by clicking *Create*.

Select your project and select the **DrawWithFire** target. Open the **Build Phases** tab and open the **Link Binary With Libraries** fold. Click the plus button and double click **QuartzCore.framework** to add Quartz drawing capabilities to the project.

You will start the project by creating a custom `UIView` class, which will have `CAEmitterLayer` instance as its layer. You can actually achieve this very easy by overwriting the `+ (Class)layerClass` method of the `UIView` class and returning a `CAEmitter` class. Pretty cool – you'll see the code in a second.

Create a new file with the **iOS\ Cocoa Touch\ Objective-C class** template, name the class **DWFParticleView**, and make it a subclass of **UIView**.

Open **DWFParticleView.m** and replace it with the following:

```
#import "DWFParticleView.h"
#import <QuartzCore/QuartzCore.h>

@implementation DWFParticleView
{
    CAEmitterLayer* fireEmitter; //1
}
-(void)awakeFromNib
{
    //set ref to the layer
    fireEmitter = (CAEmitterLayer*)self.layer; //2
}

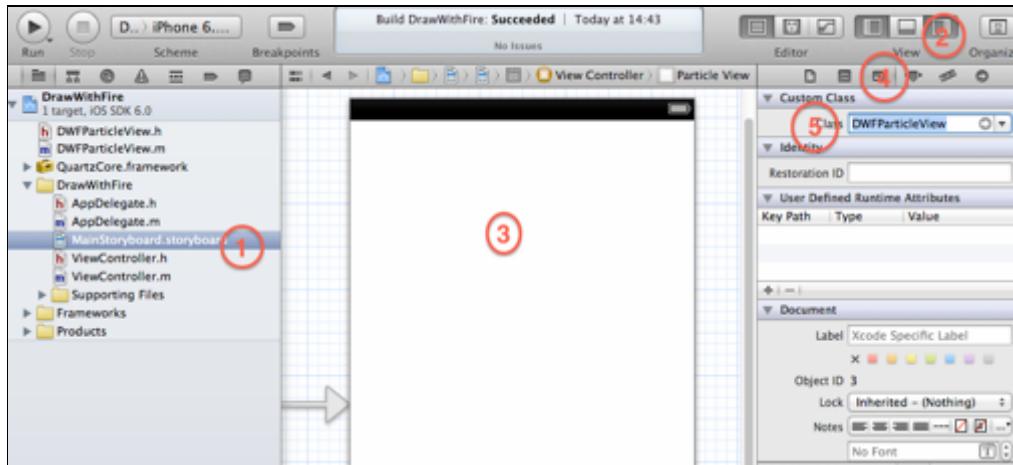
+ (Class) layerClass //3
{
    //configure the UIView to have emitter layer
    return [CAEmitterLayer class];
}

@end
```

Let's go over the initial code:

1. You define a single private instance variable to hold our `CAEmitterLayer`.
2. In `awakeFromNib` you set `fireEmitter` to be the view's `self.layer`. You store it in the `fireEmitter` instance variable, because you are going to set a lot of parameters on this later on.
3. `+ (Class)layerClass` is the `UIView` method that tells `UIKit` which class to use for the root `CALayer` of the view. For more information on `CALayers`, check out the "[Introduction to CALayer Tutorial](#)" on [raywenderlich.com](#).

Next let's add our view controller's root view to `DWFParticleView`. Open up `DWFViewController.xib` and perform the following steps:



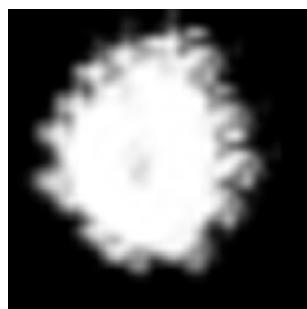
1. Make sure the Utilities bar is visible (the highlighted button on the image above should be pressed down).
2. Select the gray area in the Interface builder - this is the view controller's root view.
3. Click the Identity Inspector tab.
4. In the Custom class panel enter `DWFParticleView` in the text field.

At this point we have the UI all set - good job!

Let's add some particles to the picture.

A Particle Examined

In order to emit fire, smoke, waterfalls and whatnot you'll need a good PNG file to start with (your particle image). You can make it yourself in any image editor program; have a look at the one I did for this tutorial (it's zoomed in and rendered on a dark background so you can actually see the shape)



My particle file is 32x32 pixels in size, it's a transparent PNG file and I just used a little bit funkier brush to draw randomly with white color. For particles is best to use

white color as the particle emitter will take care to tint the provided image to the colors you would like to have. It's also good idea to make particle image semi-transparent as the particle system can blend particles together by itself (you can figure out how it works by just trying out few different image files).

So, you can create a particle of your own or just use the image from this chapter's resources, but just be sure to add it to your Xcode project and have it named **Particles_fire.png**.

Let's Start Emitting!

It's time to add the code to make our `CAEmitterLayer` do its magic! Open **DWFParticleView.m**, and add the following code at the end of `awakeFromNib`:

```
//configure the emitter layer
fireEmitter.emitterPosition = CGPointMake(50, 50);
fireEmitter.emitterSize = CGSizeMake(10, 10);
```

This sets the position of the emitter (in view local coordinates) and the size of the particles to spawn.

Next, add some more code to the bottom of `awakeFromNib` to add a `CAEmitterCell` to the `CAEmitterLayer` so we can finally see some particles on the screen!

```
CAEmitterCell* fire = [CAEmitterCell emitterCell];
fire.birthRate = 200;
fire.lifetime = 3.0;
fire.lifetimeRange = 0.5;
fire.color = [[UIColor colorWithRed:0.8 green:0.4 blue:0.2
alpha:0.1] CGColor];
fire.contents = (id)[[UIImage imageNamed:
@"Particles_fire.png"] CGImage];
[fire setName:@"fire"];

//add the cell to the layer and we're done
fireEmitter.emitterCells = @[fire];
```

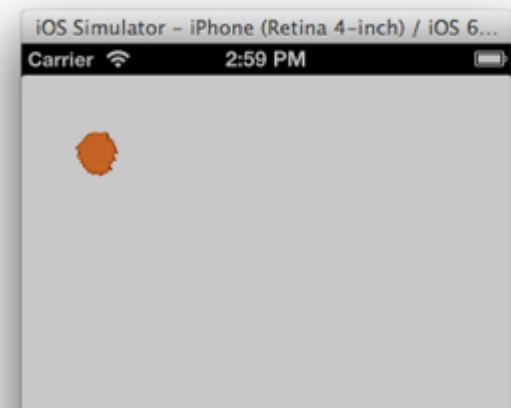
You create a cell instance and set up few properties. Then you set the `emitterCells` property on the layer, which is just an `NSArray` of cells. The moment `emitterCells` is set, the layer starts to emit particles!

Let's see what the emitter cell properties above do. Let's go over these one by one:

- **birthRate**: The number of emitted particles per second. For a good fire or waterfall you need at least few hundred particles, so we set this to 200.
- **lifetime**: The number of seconds before a particle should disappear. You set this to 3.0.

- **lifetimeRange**: You can use this to vary the lifetime of particles a bit. The system will give each individual a random lifetime in the range [lifetime - lifetimeRange, lifetime + lifetimeRange]. So in the case above case, a particle will live from 2.5 to 3.5 seconds.
- **color**: The color tint to apply to the contents. You set an orange color.
- **contents**: The contents to use for the cell, usually a `CGImage`. You set it to the particle image file's contents.
- **name**: You can also set a name for the cell in order to look it up and change its properties at a later point in time.

Run the app, and check out our new particle effect!



Well it works, but isn't as cool as you might've actually wanted.

You might barely even be able to tell it's doing something, it just looks like an orange splotch! Let's change this a bit to make the particle effect more dynamic. Add this code just before calling `setName:` on the cell:

```
fire.velocity = 10;
fire.velocityRange = 20;
fire.emissionRange = M_PI_2;
```

Here you set the following new properties on the `CAEmitterCell`:

- **velocity**: The particles' velocity in points per second. This will make your cell emit particles and send them towards the right edge of the screen
- **velocityRange**: This is the range by which the velocity should vary, similar to `lifetimeRange`.
- **emissionRange**: This is the angle range (in radians) in which the cell will emit. `M_PI_2` is 45 degrees (and since this is a range, it will be +/- 45 degrees).

That should give those particles some direction during their lifetime! :] Compile and run to check out the progress:



OK, this is better – you are not far from getting there! If you want to better understand how these properties affect the particle emitter - feel free to play and try tweaking the values and see the resulting particle systems.

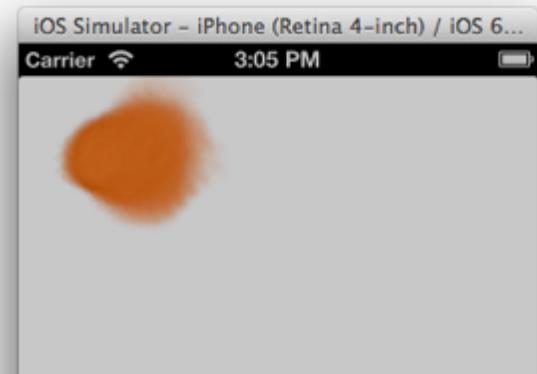
Add two more lines to finish the cell configuration:

```
fire.scaleSpeed = 0.3;  
fire.spin = 0.5;
```

Here we set two more properties on the CAEmitterCell:

- **scaleSpeed**: The rate of change per second at which the particle changes its scale. You set this to 0.3 to make the particles grow over time.
- **spin**: Sets the rotation speed of each particle. You set this to 0.5 to give the particles a nice spin.

Hit Run one more time:

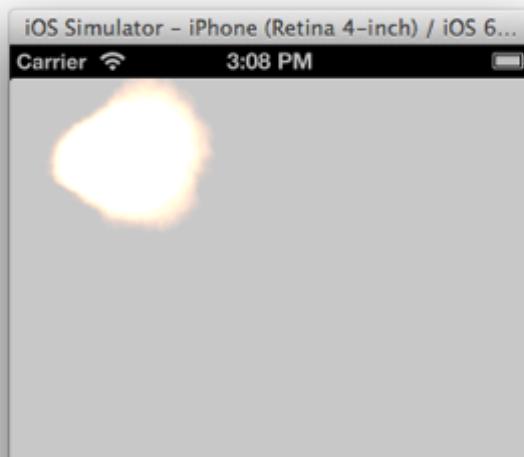


By now we have something like a rusty smoke - and guess what? `CAEmitterCell` has a lot more properties to tweak, so the sky is the limit here. But you are going to leave it like that and go on with setting some configuration on the `CAEmitterLayer`. Directly after setting `fireEmitter.emitterSize` in the code add this line:

```
fireEmitter.renderMode = kCAEmitterLayerAdditive;
```

This is the single line of code, which turns the rusty smoke effect into a boiling ball of fire.

Prepare yourself for a surprise. Hit Run and check the result:



Note: If you forgot to set gray background for the view controller, you won't be able to see anything. If you didn't do that before – now it's the time.

What's happening? The additive render mode basically tells the system not to draw the particles one over each other as it normally does, but to do something really cool: if there are overlapping particle parts - their color intensity increases! So, in the area where the emitter is - you can see pretty much a boiling white mass, but at the outer ranges of the fire ball - where the particles are already dying and there's less of them, the color tints to its original rusty color. Awesome!

Now you might think this fire is pretty unrealistic - indeed, you can have much better fire by playing with the cell's properties, but we need such a thick one because we're going to draw with it. When you drag your finger on the device's screen there's relatively few touch positions reported so we'll use a thicker ball of fire to compensate for that.

Play With Fire!

Now you finally get to play with fire (even though you've been told not to your entire life!) :]

To implement drawing on the screen by touching you will need to change the position of the emitter according to the user's touch.

First declare a method for that in **DWFParticleView.h**:

```
-(void)setEmitterPositionFromTouch: (UITouch*)t;
```

Then implement the method in **DWFParticleView.m**:

```
-(void)setEmitterPositionFromTouch: (UITouch*)t
{
    //change the emitter's position
    fireEmitter.emitterPosition = [t locationInView:self];
}
```

This method gets a touch as a parameter and sets the `emitterPosition` to the position of the touch inside the view - pretty easy.

Next you will need an outlet for the view so you can tweak it from the view controller. Open **DWFViewController.h** and replace the code with the following:

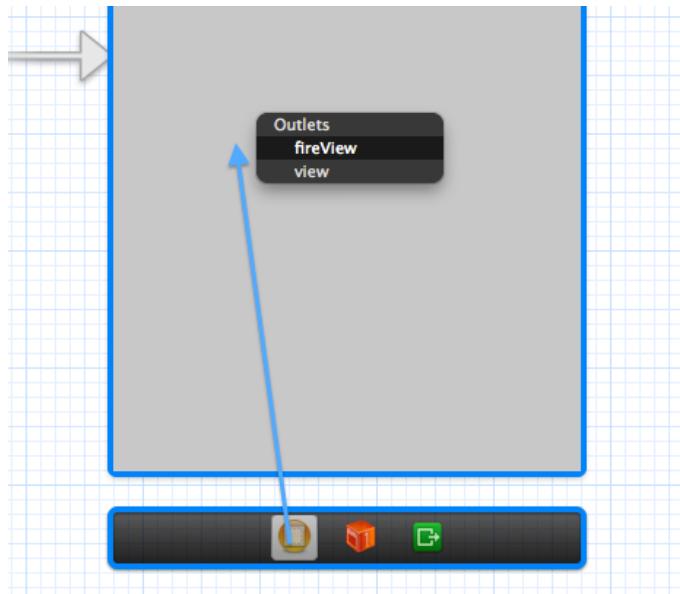
```
#import <UIKit/UIKit.h>
#import "DWFParticleView.h"

@interface DWFViewController : UIViewController
{
    IBOutlet DWFParticleView* fireView;
}

@end
```

You import the custom view class and declare an instance variable for the `DWFParticleView`.

Next open **DWFViewController.xib** and control-drag from the **View Controller** to the root **view**, and choose **fireView** from the popup:



Now you can access the emitter layer from the view controller. Open **DWFViewController.m**, remove all the boilerplate methods from the implementation, and add this instead:

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    [fireView setEmitterPositionFromTouch: [touches anyObject]];
}
```

Now hit Run - touch and drag around and you'll see the emitter moving and leaving a cool trail of fire! Try dragging your finger on the screen slower or faster to see the effect it generates.



If you move your finger faster – you will probably see a trail of fire spots. On the other hand if you drag your finger slower and with a bit of pressure – you will draw nice fire lines!

Dynamically Modifying Cells

The last topic for today will be modifying cells in an emitter layer dynamically. Right now the emitter emits particles all the time, and that's not really giving the feel to the user that they're really drawing on the screen. Let's change that by emitting particles only when there's a finger touching the screen.

Start by changing the `birthRate` of the cell at creation time to "0" in **DWFParticleView.m**'s `awakeFromNib` method:

```
fire.birthRate = 0;
```

If you run the app now, you should see an empty screen. Good! Now let's add a method to turn on and off emitting. First declare the method in **DWFParticleView.h**:

```
- (void)setIsEmitting:(BOOL)isEmitting;
```

Then implement it in **DWFParticleView.m**:

```
- (void)setIsEmitting:(BOOL)isEmitting
{
    //turn on/off particles
    [fireEmitter setValue:
        [NSNumber numberWithInt: isEmitting?200:0]
    forKeyPath:@"emitterCells.fire.birthRate"];
}
```

Here you use the `setValue:forKeyPath:` method so you can modify a cell that's already been added to the emitter by the name you've set earlier. You use `"emitterCells.fire.birthRate"` for the keypath, which translates to: *the birthRate property of the cell named fire, found in the emitterCells array.*

Finally you will need to turn on the emitter when a touch begins and turn it off when the user lifts their finger. Inside **DWFViewController.m** add:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    [fireView setEmitterPositionFromTouch: [touches anyObject]];
    [fireView setIsEmitting:YES];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    [fireView setIsEmitting:NO];
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    [fireView setIsEmitting:NO];
}
```

Compile and run the project, and watch out - you're playing with fire! :]



Where to Go From Here?

If you've enjoyed this tutorial, there's a lot more you can play around with from here! You could:

- Experiment with different particle image files.
- Go through `CAEmitterCell` reference docs and have a look at all its properties.
- Add functionality to render the image on the screen to a file.
- Capture the drawing as a video file.

Chapter 25: Using the iOS Dictionary

By Marin Todorov

In iOS 5, Apple has introduced a new system-wide dictionary service. And the best part is they have made a new API that allows you to make use of this in your own apps!

Now when I say dictionary, I actually mean a reference dictionary for definitions of words. This means Apple can provide a definition for a word, but they don't actually provide you with a list of all dictionary words.

So if you need a list of words, you'll still have to build this yourself. On the other hand - if you're creating a multiplayer scrabble game and you want your players to be able to lookup if a given word exists and what it means - you're all set!

Here's what the new dictionary looks like in iOS 5:



In this tutorial you are going to experiment with the new dictionary APIs by creating a very simple word game. The user will be presented with a picture and 3 words and he needs to choose the correct word for the picture. When they make their choice, they will be presented with the dictionary entry for the correct word.

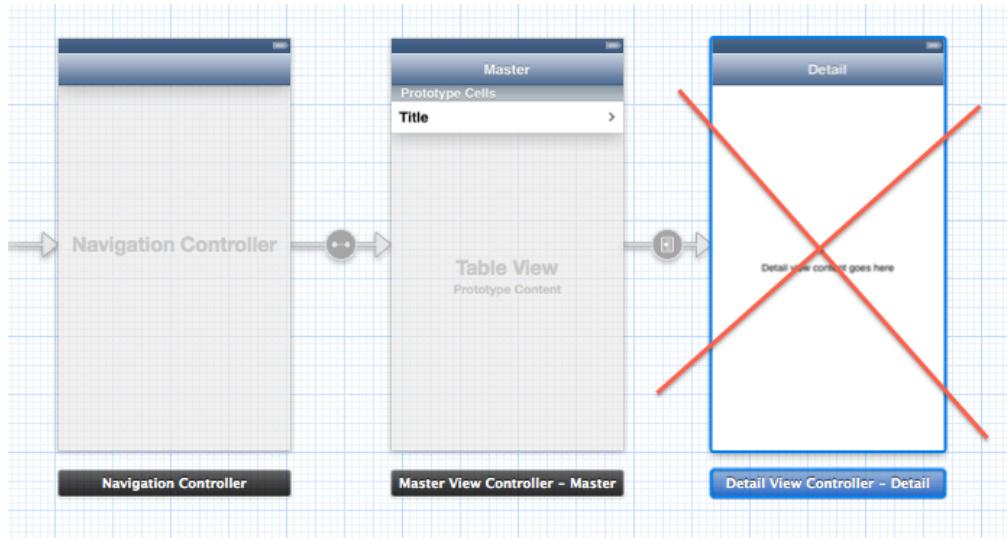
Getting Started

Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Master-Detail Application** template (as this will set up a table view controller for free), and click Next. Enter **GuessTheWord** for the product name, enter **GTW** for the class prefix, select iPhone for the Device Family, and make sure that **Use automatic reference counting** and **Use Storyboards** are checked (leave the other checkboxes unchecked). Click *Next* and save the project by clicking *Create*.

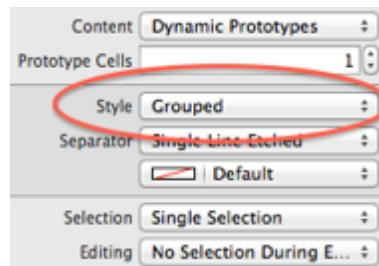
You will show the pictures and the possible answers in the table view, and when the user taps a cell you will push a new reference dictionary view on top. Thus we won't need the Details view controller Xcode that has created for you by default. So go ahead and delete the following files:

- GTWDetailViewController.h
- GTWDetailViewController.m

Open up **MainStoryboard.storyboard** and delete the detail view controller:



Next zoom in on the master view controller and click with the mouse on the empty space around the label "Table View" to select the table. Select as table style "Group" from the drop box:



That's about all the interface setup we'll need to do for the project. Good job!

Adding the Implementation

Open **GTWMasterViewController.m** and replace all the boilerplate code with:

```
#import "GTWMasterViewController.h"

//Interface declarations
@interface GTWMasterViewController()
{
    int curWord;
    NSArray* objects;
}
```

```

        NSArray* answers;
    }
@end

//Implementation declarations
@implementation GTWMasterViewController

@end

```

In this code you just declare three variables you will need in order to implement the game. Let's see what they will do:

- **curWord** – it's the index of the current word to guess from the list of objects.
- **objects** – is a list of words – essentially objects the user will see on the screen and than guess what they are.
- **answers** – a list of possible answers for each of the words in the **objects** array.

As soon as the application launches and the first view controller is loaded you will need to initialize the lists from above. Let's add the code to do that:

```

-(void)awakeFromNib
{
    self.title = @"Guess the word";

    objects = @[@"backpack", @"banana", @"hat", @"pineapple"];

    answers = @[
        @{@"backpack": @"bag", @"chair": @"orange", @"strawberry": @"banana", @"hut": @"head", @"poppler": @"apple", @"pineapple": @"pineapple"};
        curWord = 0;
}

```

Here you do some setup:

- **self.title** - sets the navigation bar title.
- **objects** - is the word list for our game.
- **answers** - are the lists of possible answers for each of the words in `self.objects`.
- **curWord** - is the index of the current word in `self.objects` (the current word the player sees in the game.)

Good job so far!

Setting Up the Table

Next you're about to add few methods to configure the table view. You will need one section with a large header on top where you are going to show the image of the current word to guess. Also you'll need three rows in the table where you're going to show the possible answers. Add these methods to the implementation body:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1; //we show 1 word on each screen
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return 3; //we show 3 possible answers
}

-(CGFloat)tableView:(UITableView *)tableView
heightForHeaderInSection:(NSInteger)section
{
    return 150.0; //the image of the word is 150px high
}
```

In the header of the only section you're going to be showing images called "object_banana.png", "object_hat.png" and so forth. The images you are going to use are created by Vicki Wenderlich and released under Creative Commons license. You'll find the 4 images in the resources for this chapter, so go ahead and add them to your project.



Let's add the code, which will create the table cells:

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

```
UITableViewController *cell = [tableView  
dequeueReusableCellWithIdentifier:CellIdentifier];  
cell.textLabel.text = answers[curWord][indexPath.row];  
cell.textLabel.textColor = [UIColor blackColor];  
  
return cell;  
}
```

So far so good! You can as well hit Run and see what pops up in the Simulator:



Cool! Now with just one more piece of code you're going to also show the current word as a picture on top. Just add this method:

```
- (UIView *)tableView:(UITableView *)tableView  
viewForHeaderInSection:(NSInteger)section  
{  
    NSString* imgName = [NSString  
        stringWithFormat:@"object_%@.png", objects[curWord] ];
```

```
UIImageView* img = [[UIImageView alloc] initWithImage:  
    [UIImage imageNamed:imgName] ];  
img.frame = CGRectMake(0, 0, 150, 150);  
img.contentMode = UIViewContentModeScaleAspectFit;  
return img;  
}
```

Inside `tableView:viewForHeaderInSection` you create a `UIImageView` and pass it to the table view as the section header's view. Easy peasy! Let's hit Run and...



Awesome!

Showing the Dictionary

You will make the Dictionary view controller appear when the player taps a word in the table. In order to have this going you'll need a `tableView:didSelectRowAtIndexPath:` method.

When the player taps a row, you'll get the row index and compare the text of the selected cell to the current picture we're showing. If they match, you'll set the text color to green and if not - to red. Finally, you'll show the dictionary definition of the answer the player gave, so they can figure out their mistake, or read more about the word they've successfully guessed.

So add this method to the implementation:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString* word = objects[curWord];

    UITableViewCell* cell = [tableView
cellForRowAtIndexPath:indexPath];
    [cell setSelected:NO];

    if ([word compare: cell.textLabel.text]==NSOrderedSame) {
        //correct
        cell.textLabel.textColor = [UIColor greenColor];
    } else {
        //incorrect
        cell.textLabel.textColor = [UIColor redColor];
    }

    [self performSelector:@selector(showDefinition:)
withObject:cell.textLabel.text afterDelay:1.5];
}
```

Let's go over this bit-by-bit:

- `curWord` is the index of the current word in `objects` so we fetch the current word in the `NSString` instance `word`.
- Next you fetch the tapped table cell and de-select it.
- If `word` equals `cell.textLabel.text` you set the label's color to green, otherwise to red.
- You do a delay of 1.5 seconds so the player can notice the color change, and then you call `showDefinition:` to show the dictionary.

OK! The only thing left is to actually show the word definition, so add the `showDefinition:` method as follows:

```
- (void)showDefinition:(NSString*)word
{
    UIReferenceLibraryViewController* dictionaryView =
    [[UIReferenceLibraryViewController alloc] initWithTerm: word];
```

```
[self presentViewController:dictionaryView animated:YES completion:nil];

//also move to next word
if (curWord+1 < objects.count) {
    curWord++;
} else {
    curWord = 0;
}
}
```

Here you initialize the `UIReferenceLibraryViewController` by calling `initWithTerm:` and passing to it an `NSString` with the word you'd like to lookup. That'll open the given word's definition in the view controller and it'll be ready to be presented on the screen.

Presenting the dictionary is the same as doing that with any other view controller - you just call `presentViewController:animated:completion:` on your navigation controller, or you present it via popup on the iPad.

The first time you show the dictionary view controller, there will be a little delay - apparently some initialization goes behind the scenes; keeping that in mind you might want to have a spinner or any other HUD showing up as it may actually take up to few seconds.

The rest of the method deals with the game logic. Once you present a word definition- the player should advance in the game, i.e. he should see the next word in the `objects` list. If he reaches the end of the word list - the game will just start from the beginning.

Time to run the app and give it try!



In case you have iOS6.0

Unfortunately in the 6.0 release of iOS the built-in dictionary is somewhat broken. (A number of people have confirmed that in the Apple forums.)

If you have a fresh iOS 6.0 on your device you will most probably see this screen, no matter which word you tap in the app:

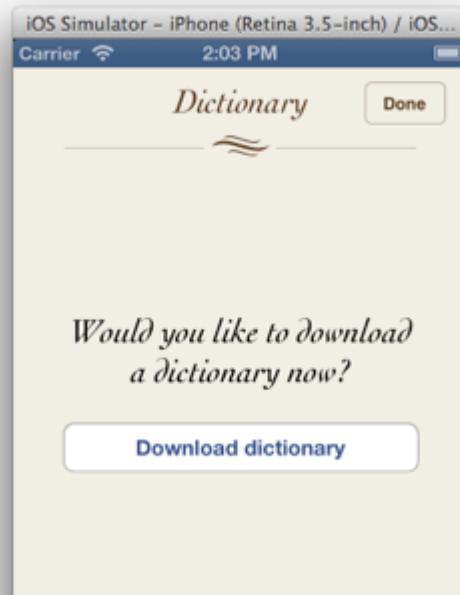


Note: If you are running a newer iOS and the dictionary is working skip this section.

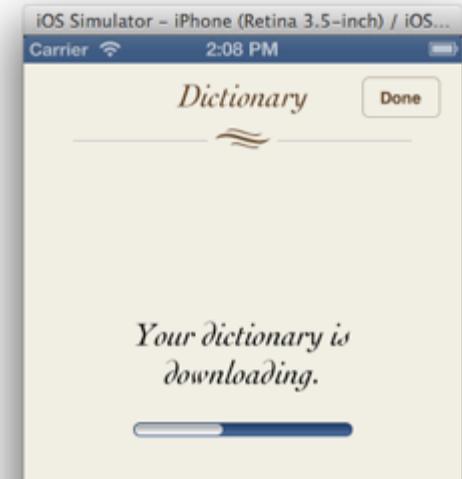
The Dictionary is not installed by default on iOS6.0, and there is no way to install it from the Settings app. There is a workaround for you if you want to go through this chapter.

Open up the Settings app. Choose: General > International > Language > British English. This will restart your device and set the language to British English (no worries, you will still be able to understand everything and you need this change just temporarily.)

Now run the project again from Xcode. This time you should see the following popping up when you select a word:



Tap on "Download dictionary" and the reference dictionary will be downloaded from Apple servers.



The definition of the word you initially tapped will pop up by itself when the download is finished. Now you can set the language back to English and still use the dictionary you just downloaded.

Let's go on

It looks like the game works, but does it really? The word doesn't really change after you close the dictionary view...

You still don't change the table contents anywhere in the code, so let's do that. You will just use the `viewWillAppear:` method on the table view controller - this way when you close the dictionary `viewWillAppear:` will be automatically invoked and you can reload the table data in there. Add this final method to the implementation body:

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

`[self.tableView reloadData]` reloads the table data and since you've already changed the `curWord` index when presenting the dictionary, now the next word and the new answers will appear on the screen. Cool!

One last thing about the dictionary API - if you would like to first check if the given word exist, or only do that without showing any UI this is how you can check:

```
BOOL wordIsFound = [UIReferenceLibraryViewController
    dictionaryHasDefinitionForTerm: @"byword"];
```

Where To Go From Here?

Using the new dictionary view controller is pretty simple - it only has two methods and we already discussed them! But I'm sure you can come up with new and fresh ways how to integrate the dictionary into your apps. Think about:

- Validating words entered by the app user towards the dictionary.
- Providing definition reference of selected words in a text.
- Integrating the dictionary in your multiplayer Scrabble game.

Chapter 26: New AddressBook APIs

By Marin Todorov

The AddressBook framework on iOS allows your app to access the user's contacts in the Address Book app. Using it is a great way to make your app more connected and more social.

In this chapter, you will learn about two major new features added to the AddressBook framework in iOS 5.

The first new feature is a new multi-value field added into the Person record in the Address Book that allows you to store contacts' social profiles. It includes predefined constants for the most popular social networks: Facebook, Flickr, Game Center, LinkedIn, MySpace and Twitter.

The second new feature is the new capability to bulk import and export contacts via vCard records. vCard is a plain text format which allows you to transfer contact data easily over mail, http, etc.

For more information on the format, check out the Wikipedia entry:
<http://en.wikipedia.org/wiki/VCard>

Introducing the Social Agent



In this tutorial you are going to develop a simple cloud connected app (and when I'm saying cloud I don't mean iCloud, but the web - it's fancier that way), to help you socialize with more people.

Imagine you're starting a new class at the community college and you don't know any of your course-mates. Luckily the college has put online a students directory and gave you credentials to access it (based on the courses you attend). So the app you are going to develop will connect to an online service, give you a list of names, and allow you to choose which ones to import into your address book.

In this tutorial I took the role of the college's IT department, and I've created a simple directory service. It provides you with two things:

1. [directory index](#): Gives you back a plist format file with the peoples' contact data.
2. [vCard data](#): You supply a list of IDs and it gives you back vCard with their contact data.

The service is just quickly hacked for testing purposes of course. You don't need to write or understand the web service code to go through this tutorial (you can just use the one I already created), but if you're particularly interested feel free to take a look at the PHP code [here](#).

Getting Started

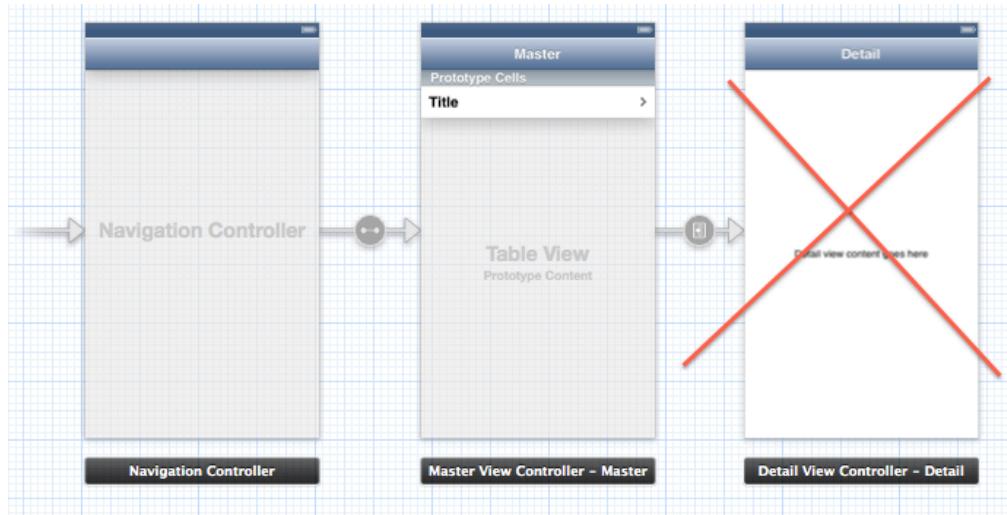
Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Master-Detail Application** template (as this will set up a table view controller for free), and click **Next**. Enter **SocialAgent** for the product name, enter **SA** for the class prefix, select iPhone for the Device Family, and make sure that **Use automatic reference counting** and **Use Storyboards** are checked (leave the other checkboxes unchecked). Click **Next** and save the project by clicking **Create**.

Select your project and select the SocialAgent target. Open the **Build Phases** tab and open the **Link Binary With Libraries** fold. Click the plus button and double click **AddressBook.framework** to add AddressBook capabilities to the project.

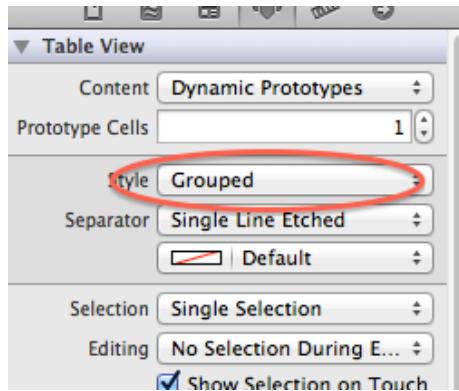
You will use the table view controller that Xcode created, but you don't need the Details view controller. So - time for a little cleanup. Delete the following files from the Project Navigator:

- SADetailViewController.h
- SADetailViewController.m

Now open **MasterStoryboard.storyboard**. First delete the detail view controller (the most right one):



Then zoom in on the master view controller and click on the table view, which appears in Interface Builder. Make sure the Attributes Inspector is open on the right (1) and from the **Style** combo box choose **Grouped** (2).



Now you are ready to start coding!

Preparing the Table View

The next step is to prepare the table view to be able to display the contact information that you are going to pull from the web directory.

Open up **SAMasterViewController.m** and replace the contents of the file with the following:

```
#import "SAMasterViewController.h"

#import <AddressBook/AddressBook.h> //1

#define kBgQueue dispatch_get_global_queue
(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0) //2

#define kContactsList [NSURL URLWithString: @"http://www.touch-
code-magazine.com/services/ContactsDemo/] //3

@interface SAMasterViewController()
{
    NSArray* contacts; //4
}
-(void)importContacts; //5
@end

@implementation SAMasterViewController

-(void)importContacts {
}

@end
```

You start very humbly, but things are gonna get crazy in just a wee while - have patience, young padawan! Let me go over the code we have just now:

1. Imports the Address Book framework header.
2. You use `dispatch_get_global_queue()` from Grand Central Dispatch here to get a background processing queue. You are going to use that queue so you can fetch the data from the Internet in the background, instead of blocking the app's UI.
3. This is just a handy reference to the `NSURL` for the web directory service.
4. This contacts instance variable will serve as data source for the table view.
5. You will use `importContacts` later on - stay tuned!

You still have a few more things to get out of the way before you can get to the interesting code. Add these methods to the class to implement our table view data source:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView*)tableView
{
    return 1; //we'll need only 1 section
}

-(NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return contacts.count; //contacts is our data source
}
```

Note that the table view only has one section (`numberOfSectionsInTableView:`) and it'll have as many rows as objects you have in the contacts `NSArray` instance variable (`tableView:numberOfRowsInSection:`).

Let's have a quick look at what contact data our service returns. For each person it will give you the following fields:

- name (first name)
- family name (last name)
- telephone number
- Facebook
- Skype

So, you are going to show the name and the family name in each cell in the table. Let's do that by adding a `tableView:cellForRowIndexPath:` method to the class:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
```

```
UITableViewController *cell = [tableView  
dequeueReusableCellWithIdentifier:CellIdentifier];  
  
cell.textLabel.text = [NSString stringWithFormat:@"%@ %@",  
    contacts[indexPath.row][@"name"],  
    contacts[indexPath.row][@"family"]];  
  
return cell;  
}
```

This is a pretty standard code. It dequeues a new cell and then sets the cell's text to be the name and family name of the current record from `contacts`.

And that's all the setup code you need to write, now we can move onto the new Address Book APIs! If you hit Run in Xcode you will see an empty table view controller, which means we are ready to start consuming the directory service!

Consuming the Directory Cloud Service

The plan for the app is to read the contacts from the server as soon as the app is started, then load the list of people into the contacts array, and finally reload the table view.

You will do that in `awakeFromNib`, so go ahead and add this code inside **SAMasterViewController.m**:

```
-(void)awakeFromNib  
{  
    self.title = @"Social Agent"; //1  
    self.tableView.allowsMultipleSelection = YES; //2  
  
    //3  
    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem  
alloc] initWithTitle:@"Import selected"  
style:UIBarButtonItemStyleDone target:self  
action:@selector(importContacts)];  
  
    //read directory here  
}
```

1. You set the navigator bar title to "Social Agent".

2. iOS 5 contains a new feature allowing multiple table cells to be selected at the same time. You will allow the user to select all the contacts they want imported from the server, and then tap the import button to bulk import them in one shot!

3. You add a navigator bar button that calls the `importContacts` method.

The only thing left is that placeholder that says: "**read directory here**". How can we accomplish that? Well, we could (!) write the following code:

```
contacts = [NSArray arrayWithContentsOfURL: kContactsList];
[self.tableView reloadData];
```

Kids, do not try this at home :]

This is the wrong way of retrieving data from the web, because you will be blocking the main thread, and it will make the application seem unresponsive. You might not notice this in development, but if something goes wrong with the network connection your app could seem frozen for a considerable length of time, and if it takes long enough the OS might even shut down your app.

Instead of blocking the main thread, it's best to run the code on a background thread. Luckily with GDC that's pretty easy. Add this code after the "**read directory here**" comment:

```
dispatch_async(kBgQueue , ^{
    contacts = [NSArray arrayWithContentsOfURL: kContactsList];
    dispatch_async(dispatch_get_main_queue(), ^{
        [self.tableView reloadData];
    });
});
```

So what does that code do? First you call `dispatch_async` and you pass it a background queue (lookup `kBgQueue` again at the top of the file) and a block. This block will be executed in the background. It gets the contents of the cloud service, parses it out (expecting a plist file format) and then builds an `NSArray` instance from the fetched data.

So in just one line of Objective-C you call a service on the web, get the response, parse the response and then create an `NSArray`. Hooray for Cocoa touch!

But that's not all. After the contacts list is fetched, there's another call to `dispatch_async`. This time though you pass `dispatch_get_main_queue()` as first parameter - this will make sure the block you supply to it will be executed on the main thread where you can update the application's UI. And you do just that - simply call `[self.tableView reloadData]`; and the table view reads the list of people from `contacts`.

Hit Run in Xcode and let's see the results!



That was really easy wasn't it? You can also try out the new feature of having multiple rows selected while you're at it.

Let's develop just a bit more on this - let's make the screen title show how many rows are selected.

To do this, you're going to use an API on `UITableView` called `indexPathsForSelectedRows`. This method returns an `NSArray` instance filled with `indexPath` items - one for each selected row.

You're going to just count how many items it returns and update the title. So add these two methods to **SAMasterViewController.m** for when the user selects or deselects a row:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSArray* selected = [self.tableView
        indexPathsForSelectedRows];
    self.title = [NSString stringWithFormat:
        @"%i items", selected.count];
}

-(void)tableView:(UITableView *)tableView
didDeselectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSArray* selected = [self.tableView
        indexPathsForSelectedRows];
```

```
    self.title = [NSString stringWithFormat:  
        @"%i items", selected.count];  
}
```

Run your app, and you'll see the selected count in the title bar like so:



Importing vCards into the AddressBook

Right now the user can see the directory list, select the contacts they want to import, and tap the "Import selected" button, so everything is done except the actual import.

This is where you are going to work with the AddressBook framework and it's going to get messy, because it's a C-based API. Luckily you have me on your side. Let's go!

Let's first take the selected indexes and prepare them in a text string ready to be sent to the web service. Find the empty `importContacts` method and add the following between the curly brackets:

```
NSArray* selectedCells = [self.tableView  
indexPathsForSelectedRows]; //1  
  
if (!selectedCells) return; //2  
  
NSMutableString* ids = [NSMutableString stringWithString:@""];  
for (NSIndexPath* path in selectedCells) {  
    [ids appendFormat:@"%@%i,", path.row]; //3
```

```

}

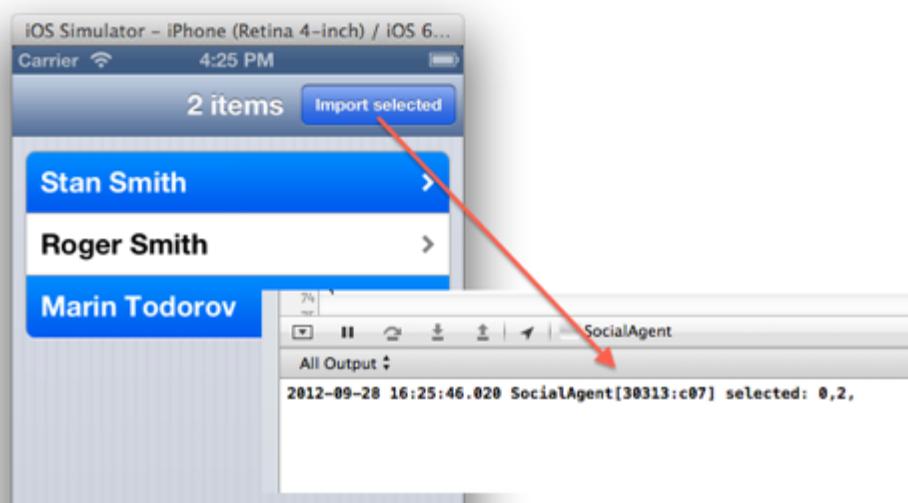
 NSLog(@"%@", selected: %@", ids); //4

```

Let's go over the code quickly:

1. `[self.tableView indexPathsForSelectedRows]` returns back an `NSArray` with the selected `indexPaths`.
2. If there are no selected rows, `selectedCells` will be `nil`, so you return in that case.
3. Next in a simple loop you append all selected indexes to the `ids` mutable string.
4. Finally, an `NSLog` dumps `ids` to the console; just to be sure everything's working correctly.

If you run the app and give it a try, you'll see the list of IDs in the console. Note it ends on a comma - the web service expects it to be like that:



Great! Next you're going to:

1. Invoke the web service again and get vCard data to import for the selected people.
2. Cycle through the people you've just imported and get their Facebook usernames, so the app can display them to the user.
3. Save the new records to the address book and show an alert that the import is finished.

You are going to go step by step. Let's start by adding the following code at the end of `importContacts`:

```

dispatch_async(kBgQueue , ^{
    NSString* request = [NSString stringWithFormat:@"%@%@", 

```

```

        kContactsList, ids]; //1

NSData* responseData = [NSData dataWithContentsOfURL:
    [NSURL URLWithString:request]]; //2
//parse vCard data

//define few constants

//loop over people and get their facebook

//save to addressbook

//show done alert

});

```

Again you consume the web service with just 2 lines of code:

1. The url is the same service URL, but with a "?" and the list of IDs appended to it. `dataWithContentsOfURL:` reads the given URL and returns the response as an `NSData` object.

Then you just have comment placeholders for everything I mentioned you are about to implement next.

Parsing the vCard Data

Replace the "**parse vCard data**" comment with the following code:

```

ABAddressBookRef addressBook =
ABAddressBookCreateWithOptions(NULL, nil); //1

ABRecordRef record = ABPersonCreate(); //2
NSArray* importedPeople = (__bridge_transfer NSArray*)
ABPersonCreatePeopleInSourceWithVCardRepresentation( record,
(__bridge CFDataRef)responseData); //3
CFBridgingRelease(record); //4

```

1. `ABAddressBookCreate()` gives you back an instance of the Address Book. It returns an `ABAddressBookRef`, which comes retained through the bridge.
2. `ABPersonCreate()` creates an empty address book record. You need an empty record for the next line of code.
3. `ABPersonCreatePeopleInSourceWithVCardRepresentation()` is the new iOS 5 function that parses vCard data and returns you an `NSArray` of people. The first

parameter is the `ABRecordRef` to use as its source - i.e. the instance of `AddressBook` record you just created. The second parameter is the `NSData` of the vCard text. Actually, it needs a `CFDataRef` - so you have to cast it as such through the bridge.

4. `CFBridgingRelease()` releases an object, which has been retained through the bridge.

Did you notice the bridge transfer syntax (`__bridge_transfer NSArray*`)? This is because the function returns a retained result, so while you're casting it to `NSArray*` you also need to "transfer" its retain count to ARC - so it'll take care to automatically release it later on for you.

Note also that `addressBook` as well as `record` are references to objects (C style) and ARC does not really do automatic reference counting for them. So you need to release them manually by explicitly calling `CFBridgingRelease()`.

You're not going to go into much more detail about how the memory management aspect of this works in this chapter so we can keep the focus on the new Address Book APIs, but if you're confused by any of this, be sure to read the Automatic Reference Counting chapters earlier in the book, where we go over the topic in greater detail.

Here's the single greatest tip how to deal with all the new bridge memory management with ARC: If you're ever in doubt - choose from Xcode's main menu: Product/Analyze. Even if ARC can't do automatic memory management- Xcode's static analyzer still knows about Cocoa conventions of function names and sees you're using functions, which have "Create" in their names. So if you do that right now the analyzer will show you that you have a potential memory leak. So that's a pretty damn good clue how to fix your code:

```
98 NSData* responseData = [NSData dataWithContentsOfURL:  
99     [NSURL URLWithString:request]]; //2  
100 //parse vCard data  
101 ABAddressBookRef addressBook = ABAddressBookCreate(); //1 ⚠ Unused variable 'addressBook' ③  
102 ABRecordRef record = ABPersonCreate(); //2 ⚠ Unused variable 'importedPeople' ②  
103 NSArray* importedPeople = (_bridge_transfer NSArray*)  
104 ABPersonCreatePeopleInSourceWithVCardRepresentation( record, (_bridge CFDataRef)responseData;  
105 CFBridgingRelease(record); //4
```

Notice it marked `addressBook` as a potential leak, but record is just fine because we already have called `CFBridgingRelease()` on it.

Defining Some Constants

Now replace the "**define few constants**" comment with:

```
//define few constants
__block NSMutableString* message = [NSMutableString
stringWithString: @"Contacts are imported to your Address Book.
Also you can add them in Facebook: "|i
```

```
NSString* serviceKey =  
    (NSString*)kABPersonSocialProfileServiceKey;  
  
NSString* facebookValue =  
    (NSString*)kABPersonSocialProfileServiceFacebook;  
  
NSString* usernameKey =  
    (NSString*)kABPersonSocialProfileUsernameKey;
```

First you define a message that you'll display in the alert later on. It's a mutable string, so you can append to it the Facebook usernames of the imported people.

Next you copy the values of few constants into `NSStrings`. You're doing that because you have an `NSArray` with the imported people, and using `NSStrings` to access different properties inside is much easier (and less messy).

Now in order to better understand what you're going to do you'll need an insight into an AddressBook record. Let's consider one person, who has several social services profiles in his record. Have a look at the schema below:

<i>NAME</i>	<i>DAVID</i>										
<i>SOCIAL PROFILE</i>	<table border="1"><tr><td><i>TYPE: FACEBOOK</i></td></tr><tr><td><i>USERNAME: STAN.CIA.SMITHHHH</i></td></tr><tr><td><i>PROFILE URL: WWW.FACEBOOK....</i></td></tr><tr><td><i>ETC.</i></td></tr><tr><td> </td></tr><tr><td><i>TYPE: FLICKR</i></td></tr><tr><td><i>USERNAME: FLICKR.SMITHHHH</i></td></tr><tr><td><i>PROFILE URL: WWW.FLICK....</i></td></tr><tr><td><i>ETC.</i></td></tr><tr><td> </td></tr></table>	<i>TYPE: FACEBOOK</i>	<i>USERNAME: STAN.CIA.SMITHHHH</i>	<i>PROFILE URL: WWW.FACEBOOK....</i>	<i>ETC.</i>		<i>TYPE: FLICKR</i>	<i>USERNAME: FLICKR.SMITHHHH</i>	<i>PROFILE URL: WWW.FLICK....</i>	<i>ETC.</i>	
<i>TYPE: FACEBOOK</i>											
<i>USERNAME: STAN.CIA.SMITHHHH</i>											
<i>PROFILE URL: WWW.FACEBOOK....</i>											
<i>ETC.</i>											
<i>TYPE: FLICKR</i>											
<i>USERNAME: FLICKR.SMITHHHH</i>											
<i>PROFILE URL: WWW.FLICK....</i>											
<i>ETC.</i>											

One Address Book record can contain various data. Some of this data is simple, like Name - it's "Bill" or "David", and so forth.

Other data is a wee bit more complicated. For example, the social profile field's value is a list of all social profiles of the person, and each profile is a dictionary of keys and values. One of these keys is the name of the service - Facebook, Twitter, etc. So what you need to do is iterate over the imported people, get their social profile item, iterate over all profiles stored inside and find the Facebook one.

Getting the Facebook Username

Replace the "loop over people and get their facebook" comment with this code:

```
for (int i=0;i<[importedPeople count];i++) { //1
    ABRecordRef personRef = (_bridge ABRecordRef)
        importedPeople[i];
    ABAddressBookAddRecord(addressBook, personRef, nil);

    //2
    ABMultiValueRef profilesRef = ABRecordCopyValue( personRef,
        kABPersonSocialProfileProperty);
    NSArray* profiles = (_bridge_transfer NSArray*)
        ABMultiValueCopyArrayOfAllValues(profilesRef);

    //3
    for (NSDictionary* profile in profiles) { //4
        NSString* curServiceValue = profile[serviceKey]; //5
        if ([facebookValue compare: curServiceValue]
            == NSOrderedSame) { //6

            [message appendFormat: @"%@, ", profile[usernameKey]];
        }
    }

    //7
    CFBridgingRelease(profilesRef);
}
```

Here you loop over the list of imported people and for each record you do the following:

1. Store the current person from the list into the `personRef` variable, and you call `ABAddressBookAddRecord()` to add the record to the address book.
2. Get the list of their social profiles into the `profilesRef` multi value structure, and by using `ABMultiValueCopyArrayOfAllValues` you end up with a nice `NSArray` of "profiles" to work with.
3. You loop over "profiles" and you get an `NSDictionary` with the data of each profile.
4. You store the type of social service into `curServiceValue` by accessing the key `serviceKey` (you defined this constant earlier).
5. You compare the Apple defined constant for Facebook with the current service type.
6. If a Facebook profile was found, you add the username to our message string.

7. Finally you call `CFBridgingRelease()` for `profilesRef` so you don't leak memory.

OK - that was a lot of code to go through, but in the end it is not so hard - just review the earlier diagram and explanations if you are a bit confused.

Let's kind of detour for a moment and have a look at what predefined kinds of social services there are. Apple defines the following constants for the `kABPersonSocialProfileServiceKey` key in each profile:

- **kABPersonSocialProfileServiceFacebook**: for Facebook profiles (you used this one!)
- **kABPersonSocialProfileServiceTwitter**: for Twitter profiles
- **kABPersonSocialProfileServiceMyspace**: for Myspace profiles
- **kABPersonSocialProfileServiceFlickr**: for Flickr profiles
- **kABPersonSocialProfileServiceLinkedIn**: for LinkedIn profiles
- **kABPersonSocialProfileServiceGameCenter**: for GameCenter profiles And further - in each profile you could find the following pieces of useful information:
- **kABPersonSocialProfileServiceKey**: the kind of profile (have a look at the list above)
- **kABPersonSocialProfileURLKey**: the URL of the profile
- **kABPersonSocialProfileUsernameKey**: the username

Saving to the Address Book

When you save people to the address book (as you did just above there) - the info is actually stored in a temporary copy of the address book. So, since at this point you're finished working with the address book instance - let's wrap up by saving and releasing it. Replace the "**save to addressbook**" comment with:

```
ABAddressBookSave(addressBook, nil);
CFBridgingRelease(addressBook);
```

`ABAddressBookSave()` saves the changes permanently to the device's address book, and finally you release the reference we got from the bridge.

Showing the Done Alert

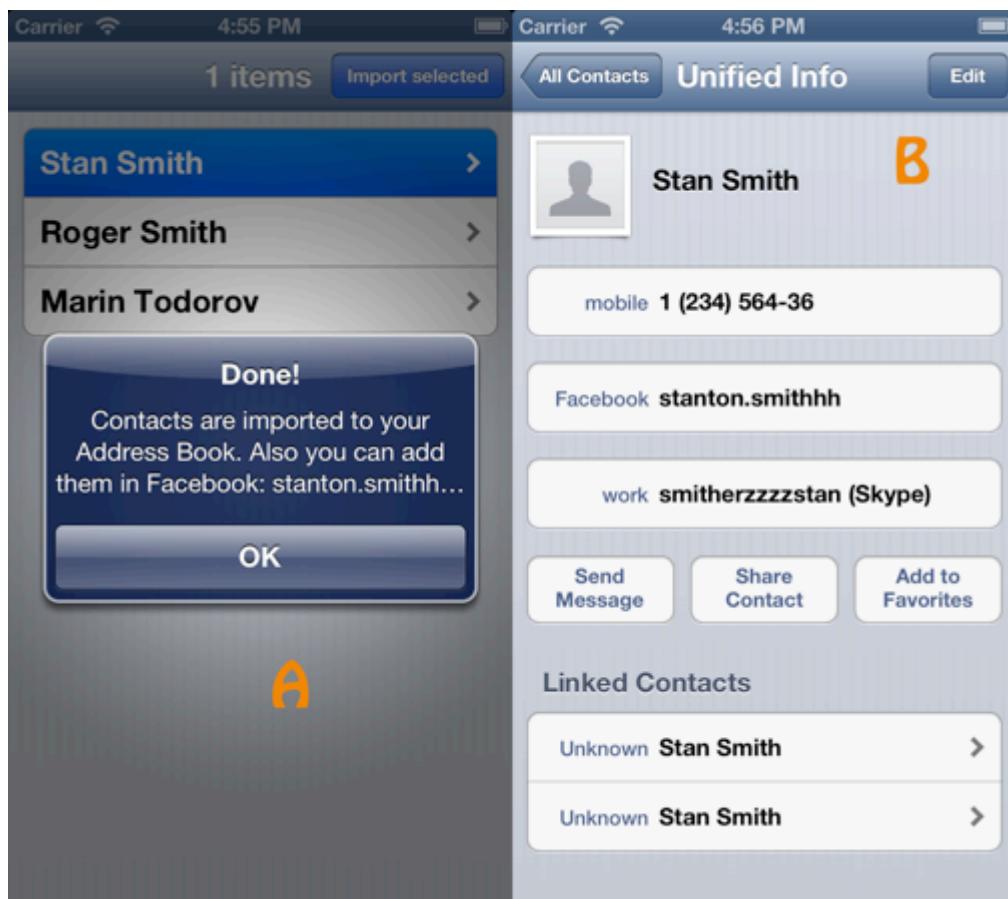
After all the work is done (in a background queue - whoa this feels like ages ago) you would like to show the user the message and let them know that the desired contacts are successfully imported.

In order to work with the application's UI you'll need to switch to the main queue. So, find the "**show done alert**" comment and replace it with this code:

```
dispatch_async(dispatch_get_main_queue(), ^{
    [[UIAlertView alloc]
        initWithTitle: @"Done!"
        message: message
        delegate: nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil] show];
});
```

There's nothing new here - just calling `dispatch_async()` as you've done it before, and instructing it to show an `UIAlertView` from the main queue. Let's give it a test!

As you can see on exhibit "A" the app successfully finishes the import and shows the alert, and visible on exhibit "B" is that the information is available immediately in the "Contacts" app:



And that's a wrap for this chapter! Congrats on making it through the entire bridging memory management and address book code goodness!

Where To Go From Here?

If you want to continue experimenting with the new Address Book APIs and this project, you could extend this in many possible ways:

- Implement contact groups for the people you import from the cloud, so they don't get mixed with the user's other contacts.
- Develop further a real web service, possibly one accessible via authentication.
- Produce cool mash-ups like "The Flickr photo stream of your contacts".

Chapter 27: New Location APIs

By Marin Todorov

Geocoding has been present in the iOS SDK for a while now, but as with several other frameworks Apple has made some improvements to the API so they are easier to work with.

In iOS 5, forward and reverse geocoding are part of the CoreLocation framework with the new class `CLGeocoder`. Placemarks are now handled by the new class `CLPlacemark`. It is way better having all the relevant classes together and leaving the MapKit framework to handle only functionality purely related to working with maps.

Furthermore testing location aware apps is becoming much easier and more powerful with Xcode 4.2. While going through the tutorial project I'm going to show you the new tools available to test your app with different pre-defined locations and how to test with your own custom locations too. Hooray for Xcode 4.2!

In this tutorial you are going to build an app called BeerAdvisor. When started it will fetch the user's location and show them their current address and location on a map. Once the app detects that the user enters a location known as a good area for drinking beer, it will alert the user that it's beer time!

Getting Started

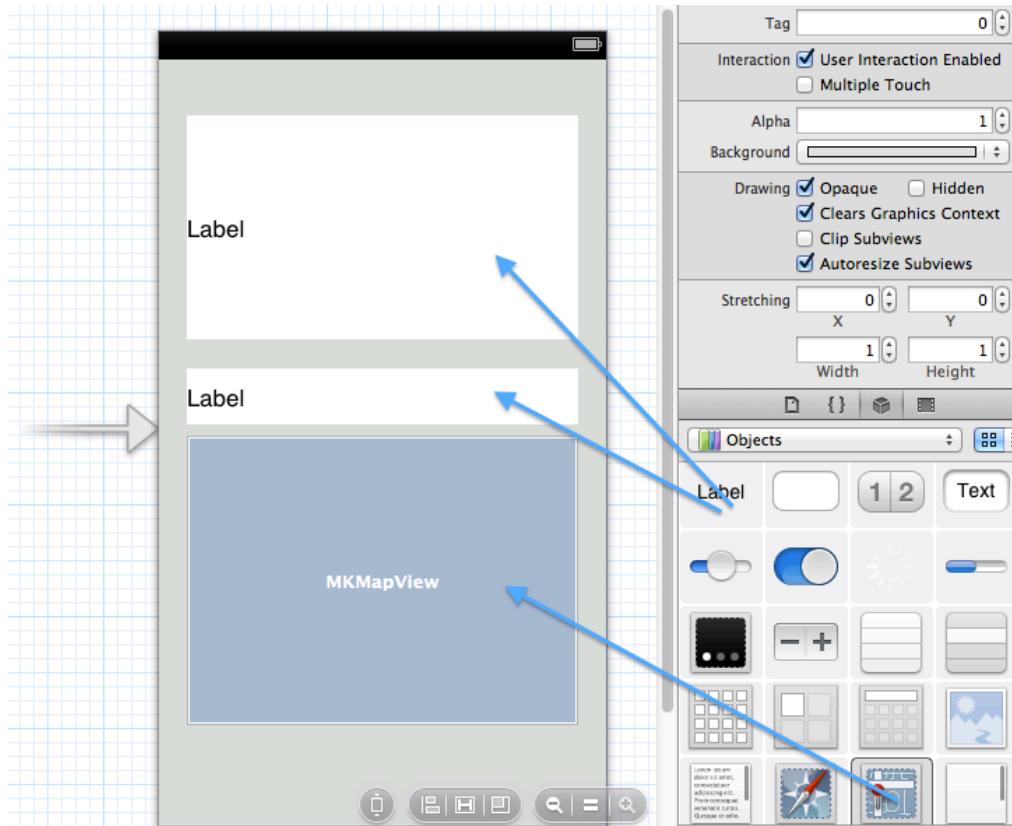
Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Single View Application** template, and click *Next*. Enter **BeerAdvisor** for the product name, enter **BA** for the class prefix, select iPhone for the Device Family, and make sure that **Use automatic reference counting** and **Use Storyboards** are checked (leave the other checkboxes unchecked). Click *Next* and save the project by clicking *Create*.

Select your project and select the BeerAdvisor target. Open the **Build Phases** tab and open the **Link Binary With Libraries** fold. Click the plus button and double click **CoreLocation.framework** to add Core Location capabilities to the project.

You'll also be using a map to show the user's location, so click again the plus button and double click **MapKit.framework**.

Open up **MainStoryboard.storyboard**. You are going to set up the interface to contain a map, a label to show the current coordinates, and a label to show the current address.

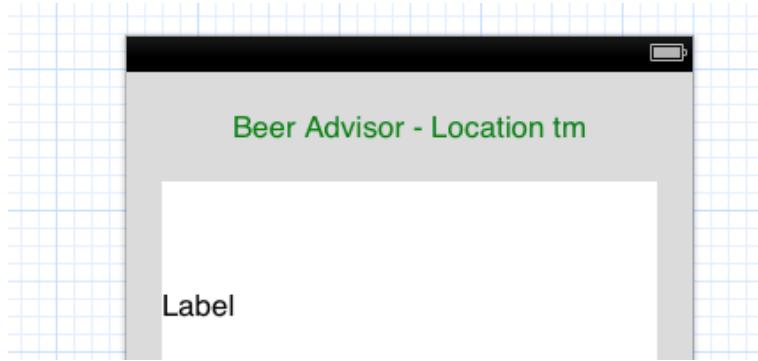
Drag two labels and one map view from the Objects library onto the application's view and position them and resize them as shown below:



In the Attributes Inspector set "Background" for both labels to white and "Color" to black. Select the top label and set the Lines to 5 in the Attributes Inspector. This should be enough for almost all kinds of address formatting.

For the second label set the **Autoshrink** property to "Minimum Font Size". For the minimum font size enter "9".

One final touch - let's add a heading to the application (beautify as you find fit):



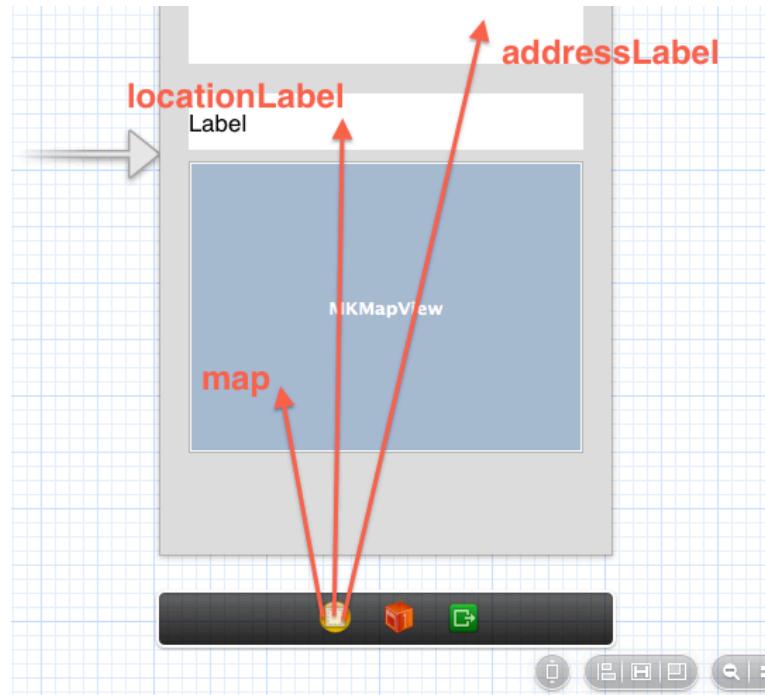
You will need few outlets to the UI elements we just created. Open up **BAViewController.m** and replace the interface definition at the top with the following:

```
#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>

@interface BAViewController : UIViewController <CLLocationManagerDelegate>
{
    IBOutlet MKMapView* map;
    IBOutlet UILabel* locationLabel;
    IBOutlet UILabel* addressLabel;
    CLLocationManager* manager;
}
@end
```

Next, open again **MainStoryboard.storyboard** and make the necessary connections:

- Control-drag from **View Controller** to the **top label**, and choose **addressLabel** from the popup.
- Control-drag from **View Controller** to the **bottom label**, and choose **locationLabel** from the popup.
- Control-drag from **View Controller** to the **map view**, and choose **map** from the popup.



OK - you're all connected! Let's move on to some coding.

Reverse Geocoding With Core Location

First the application will fetch the user's current location and show them where they are in a human readable form.

Tracking user's location is done the same way as pre-iOS 5.0. You will use the `manager` class variable that you declared in the interface of the `BAViewController` class to track the device location. You also declared in the interface part that your class would conform to the `CLLocationManagerDelegate` protocol so you can set it as a delegate to the location manager.

Let's start by turning the location tracking on.

When the app starts you would like to have changes in location reported to you. So let's hook up to `viewDidLoad` and fire up the location manager in there. Replace the boilerplate declaration of `viewDidLoad` inside the implementation body with this code:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    //start updating the location

    manager = [[CLLocationManager alloc] init];
    manager.delegate = self;
```

```
[manager startUpdatingLocation];
}
```

You create a new instance of `CLLocationManager` and store it in the `manager` instance variable. Then you set the delegate to `self` (the view controller) and finally call `startUpdatingLocation` so the location manager starts reporting location changes to the delegate.

Now to be good iOS developer, add also a `viewDidUnload` method, which will explicitly tell the manager to stop the location updates:

```
-(void)viewDidUnload
{
    [manager stopUpdatingLocation];
}
```

The `CLLocationManagerDelegate` protocol defines a method `locationManager:didUpdateToLocation:fromLocation:` for reporting locations, so you'll need to implement that in your view controller.

When you test this in the iPhone Simulator there's new locations reported constantly, so I'll implement a little condition to make the app react only to new locations.

```
- (void)locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation *)newLocation
fromLocation:(CLLocation *)oldLocation
{
    if (newLocation.coordinate.latitude != oldLocation.coordinate.latitude) {
        [self revGeocode: newLocation];
    }
}
```

Do you notice that the method takes `newLocation` and `oldLocation` parameters? This fact helps you track how much the user has moved, and in the case above - if the latitude didn't change you just don't fire up the reverse geocoder. You can never save too much CPU and battery! :]

Next let's add the `revGeocode` method where we will finally do some geocoding with the new iOS 5 `CLGeocoder` class.

```
-(void)revGeocode:(CLLocation*)c
{
    //reverse geocoding demo, coordinates to an address
    addressLabel.text = @"reverse geocoding coordinate ...";
```

```
CLGeocoder* gcrev = [[CLGeocoder alloc] init];

[gcrev reverseGeocodeLocation:c completionHandler:
 ^(NSArray *placemarks, NSError *error) {

    CLPlacemark* revMark = [placemarks objectAtIndex:0];
    //turn placemark to address text

}];

}
```

As you see it is incredibly easy to get an address for a location. But there are few more things to talk about.

First of all the `CLGeocoder` class is a one-shot object. What that means is that you are to make only one forward or reverse geocoding call with one instance of `CLGeocoder`. So don't keep `CLGeocoder` instances in class instance variables or do any other type of persisting. You create one by calling `alloc` and `init`, make your call, and leave ARC to destroy it at the proper time.

`reverseGeocodeLocation:completionHandler:` makes a reverse geocoding call to the Apple's servers and invokes the block you pass when there's a response. Check `error` to see if the call was successful or not (we're skipping that to keep the tutorial shorter). In your block code iterate over the `placemarks` array to get all the possible addresses the server returned for the coordinates you provided it.

You get the first result returned from the server and store it in `revMark`. The new for iOS 5 `CLPlacemark` class provides very extended information about a location. Let's have a quick look at the properties you can access in a `CLPlacemark`:

- **`CLLocation* location`**: The location including coordinates of the placemark.
- **`NSDictionary* addressDictionary`**: The address representation of the location.
- **`NSString* ISOcountryCode`**: The country code - "US", "FR", "DE", etc.
- **`NSString* country`**: The country name.
- **`NSString* postalCode`**: The postal code of the area.
- **`NSString* locality`**: Usually the city or town.
- **`NSString* subLocality`**: Usually the neighborhood or area.
- **`NSString* thoroughfare`**: Usually the street name.
- **`NSString* subThoroughfare`**: Usually the street number.
- **`CLRegion* region`**: A geographic region associated with the location.
- **`NSString* inlandWater`**: If the location is on water - the name of the sea, river, or lake.
- **`NSString* ocean`**: If the location is in an ocean - the name of the ocean.

- **NSArray* areasOfInterest:** A list of points of interest nearby the location. (Say, how cool is that?)

For your project you are going to use only the address dictionary, but getting all sorts of info out of `CLPlaceMark` is just as easy. So, let's get the address and show it in our label. Find the "**turn placemark to address text**" comment and replace with the following code:

```
NSArray* addressLines =
    revMark.addressDictionary[@"FormattedAddressLines"];

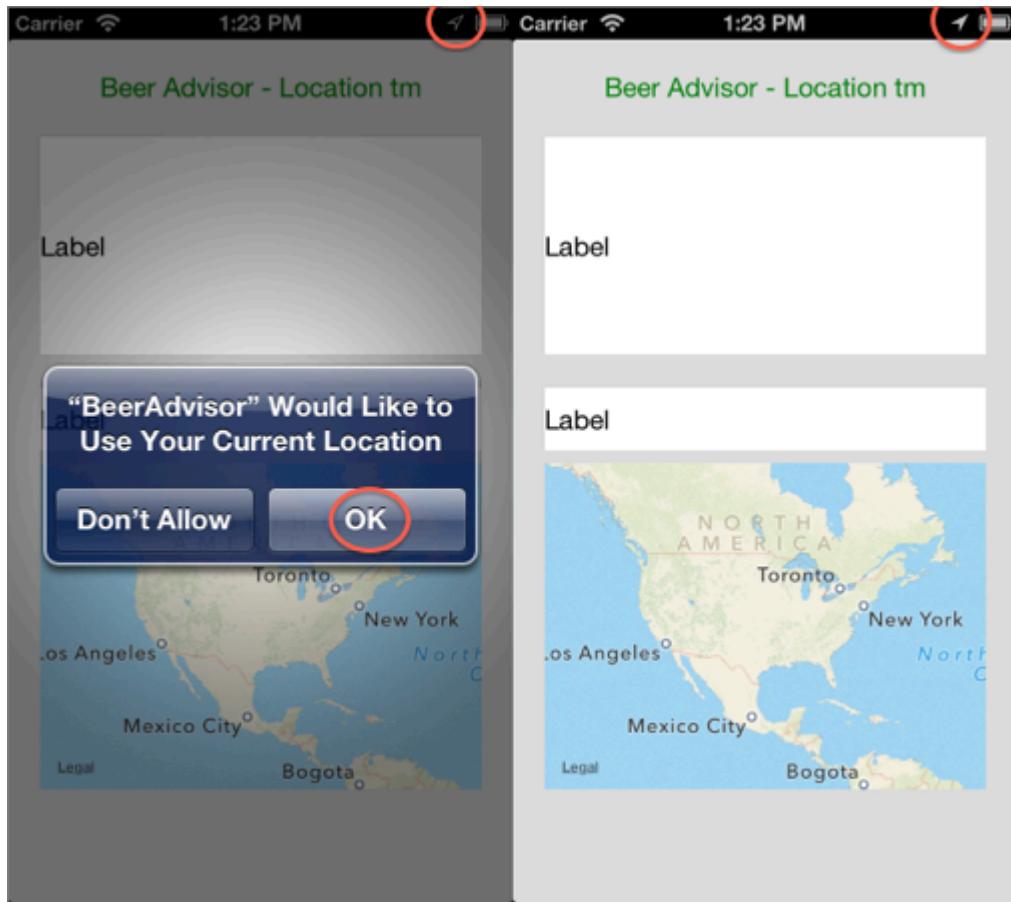
NSString* revAddress =
    [addressLines componentsJoinedByString: @"\n"];

addressLabel.text = [NSString stringWithFormat:
    @"Reverse geocoded address: %@", revAddress];

//now turn the address to coordinates
```

In the `addressDictionary` there's a key called `FormattedAddressLines` and it contains the formatted address text. It actually is an `NSArray` instance and for each text line in the address there's a single `NSString` element in the array. You use `componentsJoinedByString:` to join the text lines and store them in `revAddress`.

Last step - show the address on the screen: just set the `text` property of the address label to what you've got from the `CLPlaceMark`. Let's test, hit Run in Xcode:



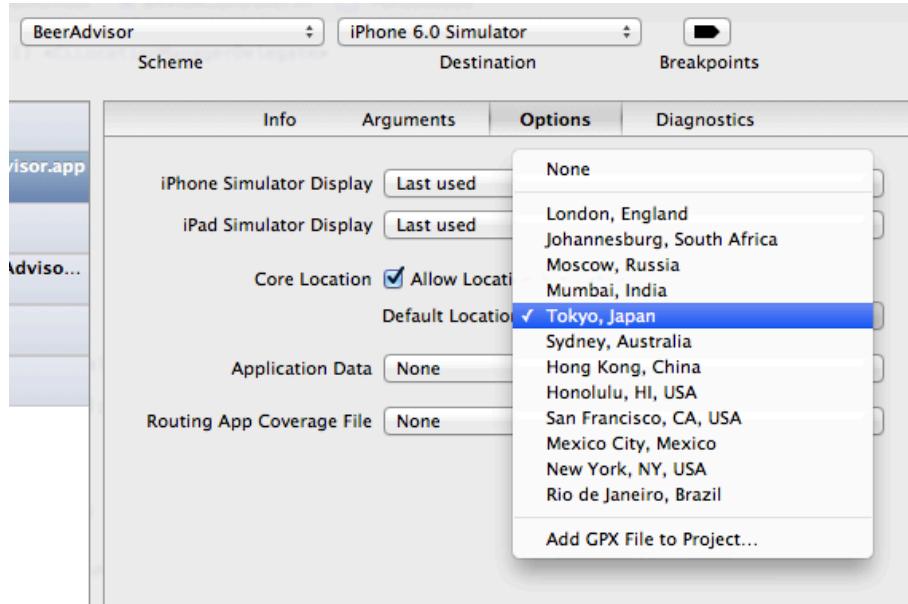
As usual, since the iOS doesn't have permission for the app to use location data, the Simulator asks for permission. Once you tap "OK" the app goes on running. However - that's not exactly what we expected - is it? Well the Simulator does not simulate location changes - this is what is wrong. We can easily fix that with the new tools Xcode 4.2 provides us with. Let's do it!

Location in the Simulator - Part 1

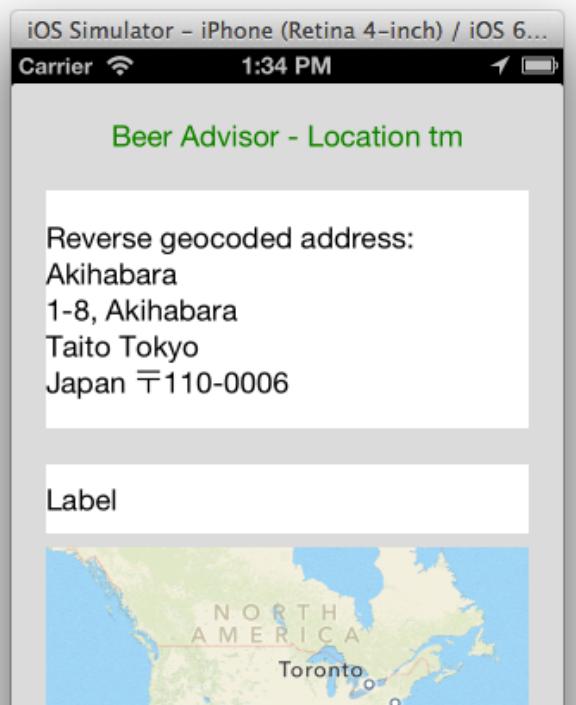
With Xcode 4.2 Apple also brings location simulation to the set of awesome iOS development tools.

One way to control the simulated location is to edit your project scheme. This way every time you run that scheme the set location will be simulated. Let's do that!

From Xcode's menu choose **Product\Edit scheme...** and sure enough the scheme properties dialogue pops up. Now open the tab saying **Options** and make sure **Allow Location Simulation** is checked. Then have a look at **Default location** and choose **Tokyo, Japan** from the pre-defined by Apple list of locations.

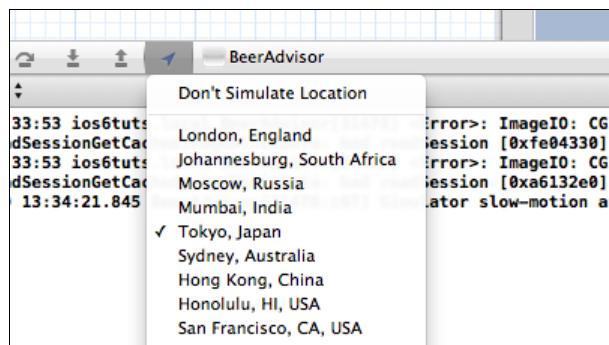


Now click "OK" and hit again the Run button:



So, the selected location's address shows up inside the address label, and hey! There's the purple arrow next to the battery indicator, which shows the application's receiving location data! Great!

Also another cool feature - look at the debugging toolbar in Xcode (just under the code editor). There's a new button in Xcode 4.2 with the location arrow - when it's blue it means that it simulates location to the application being run. Also when you click it, a list with location pops up - so you can actually change the currently simulated location at runtime. Try that - choose several different location and watch the application react and show the addresses.



Forward Geocoding

Next you are going to take the current address, convert it to coordinates, and use those to show to the user where he is on the map on the screen.

Open **BAViewController.m**, find the comment "**now turn the address to coordinates**" and replace it with:

```
[self geocode: revAddress];
```

So, once you get the address from the server, you will immediately going to make another call to convert it to coordinates.

And here's the initial method body to add inside the implementation of the class:

```
- (void)geocode:(NSString*)address
{
    locationLabel.text = @"geocoding address...";
    CLGeocoder* gc = [[CLGeocoder alloc] init];

    //2
    [gc geocodeAddressString:address completionHandler:
     ^(NSArray *placemarks, NSError *error) {
        //3
        if ([placemarks count]>0) { //4
            CLPlacemark* mark = (CLPlacemark*)placemarks[0];
            double lat = mark.location.coordinate.latitude;
            double lng = mark.location.coordinate.longitude;
        }
    }];
}
```

```
//5 show the coords text
locationLabel.text = [NSString stringWithFormat:
    @"Coordinate\nlat: %@, long: %@", 
    [NSNumber numberWithDouble: lat],
    [NSNumber numberWithDouble: lng]];
//show on the map

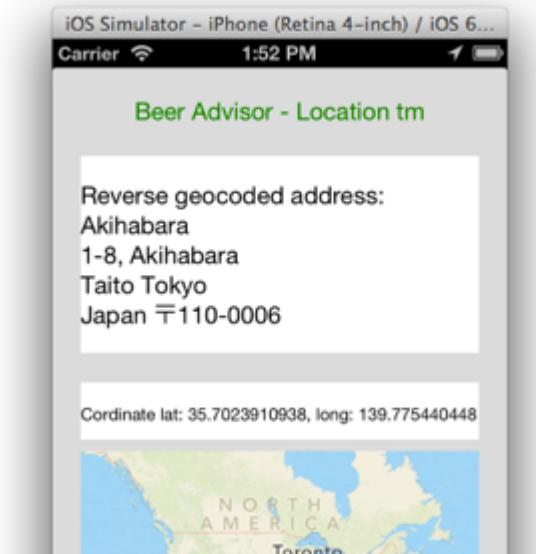
}
};

}
```

This is pretty similar to what you did before:

1. Lets the user know there's a geocoding call in progress.
2. `geocodeAddressString:completionHandler:` works the same way as the reverse geocoding call, it just takes a string as input. It also returns an array of placemark objects, so handling the result goes the same way as what you did before.
3. Unlike passing coordinates to the geocode call, passing an address might actually not return results - if the address cannot be recognized you won't get any placemarks as a result. (Also that's why is good to check the error parameter).
4. You get the coordinates from the first `CLPlacemark`.
5. Finally you show the coordinates inside the location label in the UI.

You can hit Run and see your current latitude and longitude!



OK - you're almost done, the final touch is to show the returned placemark on the map. There's nothing new in how you do that compared to iOS 4.x, so I'm not

going to discuss it in detail. First of all you need a custom location annotation class, so let's go with the most minimal class to do the job.

Create a new File, select the **iOS\Cocoa Touch\Objective-C class** template, and enter **BAAnnotation** for the class and **NSObject** for the Subclass.

Then open **BAAnnotation.h** and replace its contents with the following:

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

@interface BAAnnotation : NSObject <MKAnnotation>

@property (nonatomic) CLLocationCoordinate2D coordinate;

-(id)initWithCoordinate:(CLLocationCoordinate2D)c;

@end
```

You define a class, which conforms to the `MKAnnotation` protocol, so you can use it to add placemarks to a MapKit map view. The only required property is the `coordinate` property. It's also handy to be able to initialize the annotation in one shot, so implement a custom initializer `initWithCoordinate:`. Next open **BAAnnotation.m** and replace its contents with the following:

```
#import "BAAnnotation.h"

@implementation BAAnnotation

-(id)initWithCoordinate:(CLLocationCoordinate2D)c
{
    if (self = [super init]) {
        self.coordinate = c;
    }
    return self;
}
@end
```

This simple annotation implementation just features a custom initializer, which assigns the coordinate to the class property. Now you are ready to show the location on the map.

Open **BAViewController.m** and at the top of the code import the new annotation class:

```
#import "BAAnnotation.h"
```

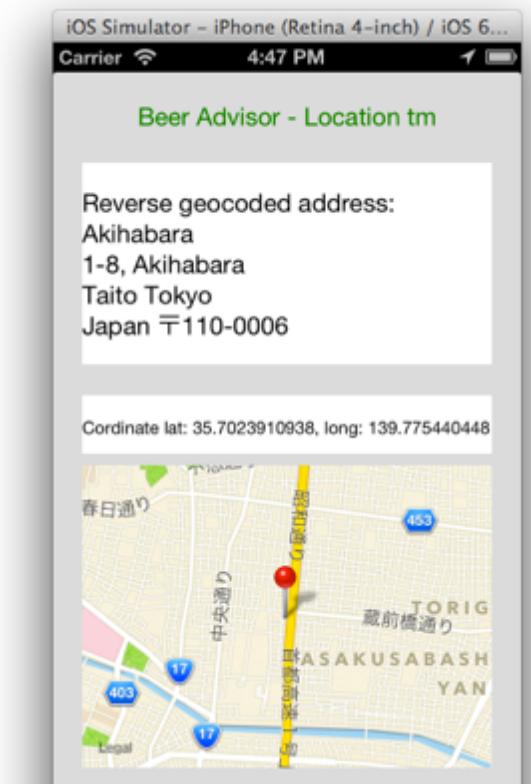
Now find the "**show on the map**" comment and replace it with this code to animate the map to the current location:

```
//1
CLLocationCoordinate2D coordinate;
coordinate.latitude = lat;
coordinate.longitude = lng;
//2
[map addAnnotation:[[BAAnnotation alloc]
    initWithCoordinate:coordinate] ];
//3
MKCoordinateRegion viewRegion =
    MKCoordinateRegionMakeWithDistance(coordinate, 1000, 1000);
MKCoordinateRegion adjustedRegion = [map
    regionThatFits:viewRegion];

[map setRegion:adjustedRegion animated:YES];
```

1. First you build a `CLLocationCoordinate2D` with the latitude and longitude you've just got back from the server.
2. In one shot you create a new `BAAnnotation` and add it to the map.
3. You build a region around the coordinate and set it to the map.

Now you can hit again the Run button and see the result:



At this point you have the app forward and reverse geocode with CoreLocation!

Monitoring for a Given Area

Monitoring for a given region is supported prior iOS 5.0, but now there's a new API called `startMonitoringForRegion:`.

The "Beer Advisor" app will have only one predefined region. This is the Kreuzberg neighborhood in Berlin - an area known for a lot of bars where tons of foreigners hang out.



1st of May in Kreuzberg by [Alexandre Baron](#)

So, at the end of `viewDidLoad` add the following code to start monitoring whether the user enters Kreuzberg:

```
//monitor for the Kreuzberg neighbourhood
//1
CLLocationCoordinate2D kreuzbergCenter;
kreuzbergCenter.latitude = 52.497727;
kreuzbergCenter.longitude = 13.431129;
//2
CLRegion* kreuzberg = [[CLRegion alloc]
    initCircularRegionWithCenter: kreuzbergCenter radius: 1000
    identifier: @"Kreuzberg"];
//3
[manager startMonitoringForRegion: kreuzberg];
```

1. You build a `CLLocationCoordinate2D` with the coordinate of the area's center more or less (I just sampled it by hand in Google maps.)
2. You create kreuzberg `CLRegion` that has the chosen center and a radius of 1000 meters (about 0.6 miles).
3. `startMonitoringForRegion:` will tell the location manager to let you know when the user arrives inside the given region.

Let's also add this code at the end of `viewDidUnload:`

```
for (CLRegion* region in manager.monitoredRegions) {  
    [manager stopMonitoringForRegion: region];  
}
```

Instead of keeping an instance variable to our region and then stop monitoring for it when the view unloads, we'll just iterate over `monitoredRegions` and disable all. This is ready for when you are going to start adding more regions.

There's one more final thing to do: react when the user enters into Kreuzberg! The location manager will call `locationManager:didEnterRegion:` on its delegate when that happens. Let's just show an alert:

```
- (void)locationManager:(CLLocationManager *)manager  
    didEnterRegion:(CLRegion *)region  
{  
    [[[UIAlertView alloc] initWithTitle:@"Kreuzberg, Berlin"  
        message:@"Awesome place for beer, just hop in any bar!"  
        delegate:nil  
        cancelButtonTitle:@"Close"  
        otherButtonTitles: nil] show];  
}
```

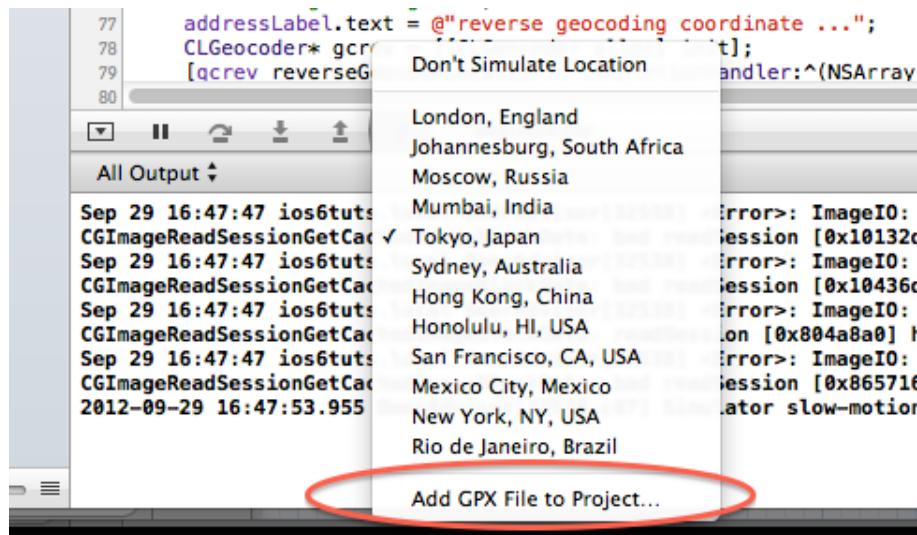
If you would like to show in the alert which region the user entered - do check out the `region` parameter. For now we're checking for just one region, so we're just showing an alert with preset text.

At this point a question comes to mind. Apple doesn't seem to be big fans of Kreuzberg - they didn't find necessary to include it in their predefined locations list, so we have a problem how to test this functionality without flying over to Berlin! (Which is also a great idea, but maybe it's better to first finish the tutorial and then head to some Berlin fun!)

Location in the Simulator - Part 2

Luckily Apple allows adding your own custom locations to simulate, so beer lovers across the world can rejoice!

While running the project, click again on the button in the debug toolbar and have a look at the list of locations. Do you notice that there's an option at the bottom called "Add GPX file to project...?"

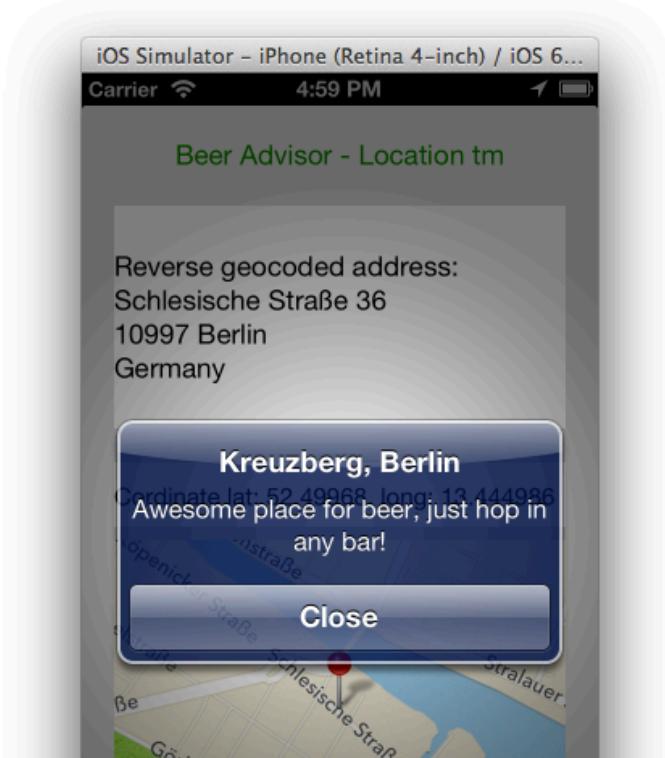


GPX files are plain text files containing XML markup and contain GPS data. If you're interested in the GPX file format, you can have a look here:

http://en.wikipedia.org/wiki/GPS_eXchange_Format

You can find the GPX file I generated as an asset file of this tutorial, it's called "**cake.gpx**". I used <http://gpx-poi.com> - a very simple tool that allows you to drop a pin on the world map and get a GPX file with the coordinates. I generated a GPX of a place in Kreuzberg (it's a bar called "Cake", thus the filename :]) and we're going to use it to showcase custom simulated locations.

Run the application and choose "Add GPX file to project..." from the locations list. In the File Open dialogue locate "**cake.gpx**" from the chapter resources and add it to the project. This will also change the simulated location to Kreuzberg, Berlin, Germany and you should see this on the screen:



Awesome! Now that you added the GPX file to the project it is always available in the simulated locations list.

Furthermore - you can also test locations on the device, just use the locations list as you did with the Simulator - pretty awesome! However ... (there's always a but, isn't there?) monitoring for regions doesn't work on the device with simulated locations. So ... if you test on the Simulator you'll get the Kreuzberg alert, but if you do that on the device you won't.

Where to Go From Here?

If you want to keep playing around with maps and locations, there's more fun stuff you can try!

- You can extend the demo project with further interesting beer regions (perhaps in your own home town!) If you do, please share them with us ;]
- Pull and visualize more interesting data from `CLPlaceMark`.
- Have a look at the heading information that the location manager supplies.

Chapter 28: New Game Center APIs

By Marin Todorov

iOS 5 introduces a ton of great new features into Game Center. Of course the biggest of these is the new Turn-Based Gaming API - but we already have an entire chapter on this earlier in the book!

Besides Turn-Based Gaming, there are also few other new APIs and it will be really unfair if they did go unnoticed. In this tutorial I'm going cover the most important of them.

First of all, Apple has included a new class in the GameKit framework called `GKNotificationBanner`. "A-ha!", those of you who have used Game Center before may be thinking, "I can use this to display text to the user!" Indeed you can - it allows you to display a banner similar to the one users see when they log into Game Center, but with your text on it. So, great! - you can keep the user messages concise and in style!

Secondly, as you may have noticed if you've played around with Game Center on iOS 5, you can now upload a photo of yourself to your Game Center profile. This is actually pretty cool - when you get a random opponent from the Internet, turn based gaming or not, it's cool to see who are you going to beat at their own game, right? Now there's a new method on the `GKPlayer` class that will get you the player's photo in the specified size - small or large.

Finally, there's a new property on the `GKAchievement` class. Apple's docs (up to now) were instructing us to let the player know in whatever way we prefer that he snagged a Game Center achievement. In practice, this turned out to be a pain because every app had to implement their own achievement notification banner and every app looked different. With iOS 5, Apple came up with a way to have a Game Center notification appear automatically when an achievement is reported as completed 100%, in a standard style. Absolutely cool!

In order to keep the tutorial to a sane length we won't create a sample game to put the new functionalities in context, but I'll just take you through using the new APIs in a simple app.

To follow this tutorial, you'll need to register the project's bundle id with iTunes Connect and activate Game Center. So if you don't have an iOS developer account you might not finish the demo project, but you can still read through!

Getting Started

Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Single View Application** template, and click *Next*. Enter **GameCenterDemo** for the product name, enter **GCD** for the class prefix, select iPhone for the Device Family, and make sure that **Use automatic reference counting** and **Use Storyboards** are checked (leave the other checkboxes unchecked). Click *Next* and save the project by clicking *Create*.

Select your project and select the GameCenterDemo target. Open the **Build Phases** tab and open the **Link Binary With Libraries** fold. Click the plus button and double click **GameKit.framework** to add Game Center capabilities to the project.

There's one final step of the setup: Setting up Game Center for your demo project, unfortunately this is out of the scope of this tutorial, but here's what you need to do in short:

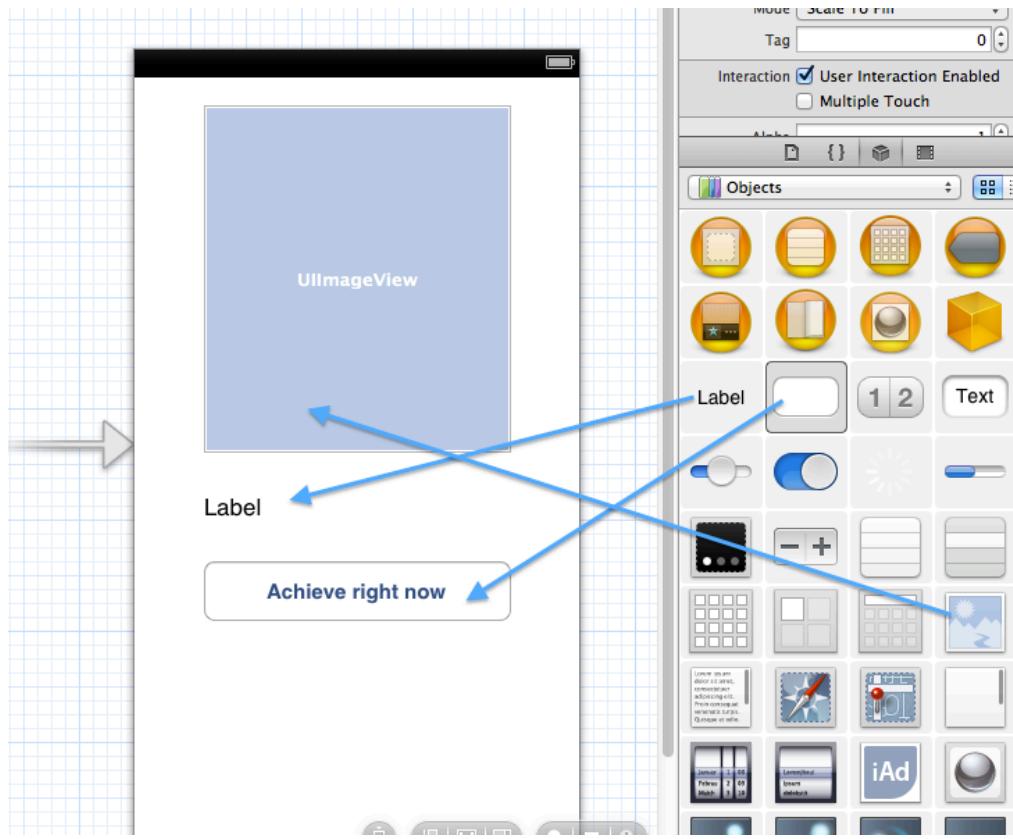
1. Create an App ID for your app in the iOS Provisioning Portal.
2. Log in to iTunes Connect, register a new application for your new App ID, and activate Game Center for it.
3. Also create one achievement, and give it an Achievement ID of "writinggamecenterapichapter".
4. Change the demo project bundle id to the bundle id you just registered for Game Center in iTunes Connect.
5. You may need to click on your new app, find the Game Center section, and click Enable for this version, and move to the Ready to Upload stage.

If you have any troubles following through this setup, refer back to the Turn-Based gaming chapter that goes through this in more detail.

Once you're done open up **MainStoryboard.storyboard**. For the complete demo project you are going to need:

1. A UIImageView to show the current player's photo.
2. A UILabel to show the logged player's handle.
3. A UIButton that will report the completion of an achievement to Game Center.

So go ahead and drag these 3 elements into the application's view, which you have open in Interface Builder at this point. Lay the elements down as shown below:



Double click the button and enter as text "Achieve right now". Select the label and in the Attributes inspector on the right select in the combo box "Background" the value "White", and for "Alignment" choose the Center aligned button (the one in the middle).

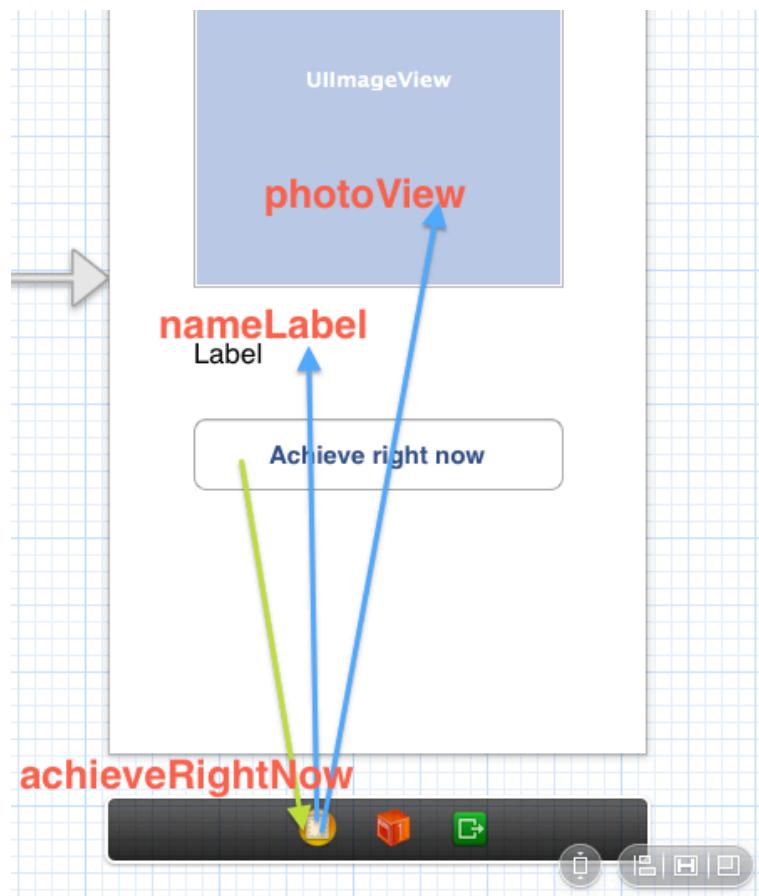
Let's add the outlets and the action needed for the interface to work. Open up **GCDViewController.m** and replace the boilerplate @interface code with:

```
@interface GCDViewController : UIViewController
{
    IBOutlet UIImageView* photoView;
    IBOutlet UILabel* nameLabel;
}
-(IBAction)achieveRightNow;
@end
```

You have two instance variables, which are outlets to our UI elements. `photoView` will help you show the player's picture in the UI, and `nameLabel` will show the player's Game Center handle. `achieveRightNow` will be invoked when the button is pressed.

Let's connect the UI elements and go on further with coding. Switch back to **MainStoryboard.storyboard** and control-drag from **View Controller** to the **UIImageView** and from the popup select **photoView**. Similarly, control-drag from

View Controller to the **UILabel** and from the popup select **nameLabel**. Last but not least, control-drag from the **UIButton** to **View Controller**, and choose **achieveRightNow**.



Good! The interface is all connected, and we can start coding.

Loading the Current Player's Photo

You want to make the app log the player into Game Center, fetch their photo and show it. You'll start by adding the code to log the player in.

Open **GCDViewController.m** and import the GameKit classes - at the very top of the file:

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

Next find **viewDidLoad** and replace it with a new implementation:

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

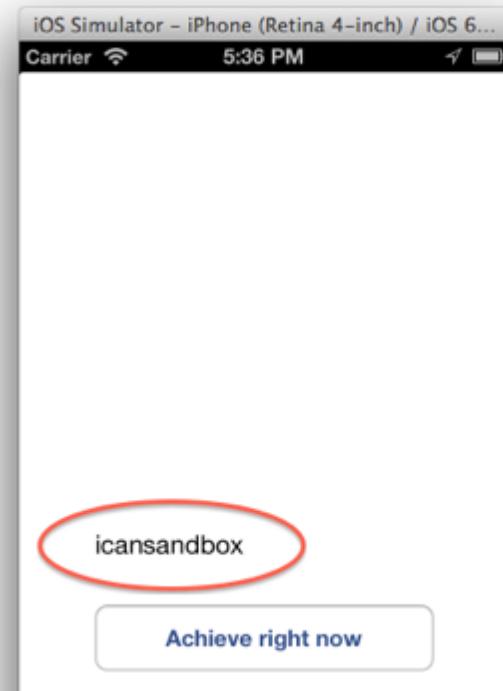
```
if ([GKLocalPlayer localPlayer].authenticated == NO) { //1
    [[GKLocalPlayer localPlayer] setAuthenticateHandler:
     ^(UIViewController *controller, NSError *error) {

        dispatch_async(dispatch_get_main_queue(), ^(void) { //3
            nameLabel.text = [GKLocalPlayer localPlayer].alias; //4
            //show the user photo
            //show logged notification
        });
    }];
}
```

Here's what this code does, section by section:

1. You first check if the local player is already authenticated with Game Center.
2. If not - you call `setCompletionHandler:` and you pass a block of code to execute when logged in.
3. As GameKit does not guarantee execution on the main queue, you need to call `dispatch_async(dispatch_get_main_queue(),...)` if you gonna do changes to the UI.
4. Finally `[GKLocalPlayer localPlayer].alias` is the logged player handle - you show it in our UI label. There's also couple of comments, which you will replace with code in a minute.

If you hit Run and test the app right now, you'll see the following result:



It's already working! You are being logged in the Game Center and your username shows in the label. (On the screenshot above of course is my username.) Note that you are running against the Game Center **sandbox** servers, not the real servers, because the app is still in development.

Next you are going to fetch the picture of the currently logged player from Apple's servers and show it on the screen. Find the comment "**show the user photo**" and replace it with this line:

```
[self loadPlayerPhoto];
```

The only thing left is to define the `loadPlayerPhoto` method, just insert it anywhere inside the implementation body:

```
-(void)loadPlayerPhoto
{
    [[GKLocalPlayer localPlayer] loadPhotoForSize:GKPhotoSizeNormal
    withCompletionHandler: ^(UIImage *photo, NSError *error) {
        dispatch_async(dispatch_get_main_queue(), ^(void) {
            if (error==nil) {
                photoView.image = photo; //show photo notification later
            } else {
                nameLabel.text = @"No player photo"; }
        });
    }];
}
```

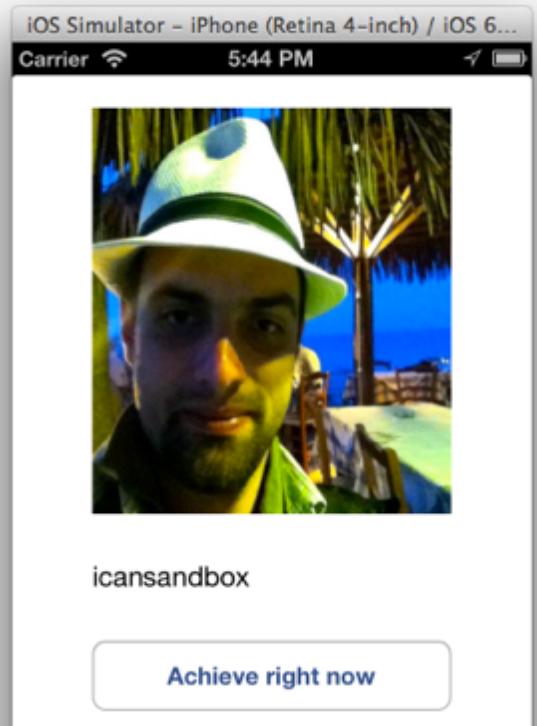
So to get the photo for the local player in iOS 5, all you have to do is call `loadPhotoForSize:withCompletionHandler:` like you see here. Alternatively if you're reading somebody else's photo you'll have to load a `GKPlayer` instance for him/her and then load the photo.

The photos come from Game Center in two sizes; if you don't need the biggest picture - get the smaller one and save some bandwidth. The defined photo size constants are `GKPhotoSizeSmall` and `GKPhotoSizeNormal`.

When there's a response from the server the block you pass is invoked with the following parameters: `UIImage*` `photo` and `NSError *``error`. `photo` is the player's picture and `error`, of course, denotes whether there was an error fetching the photo.

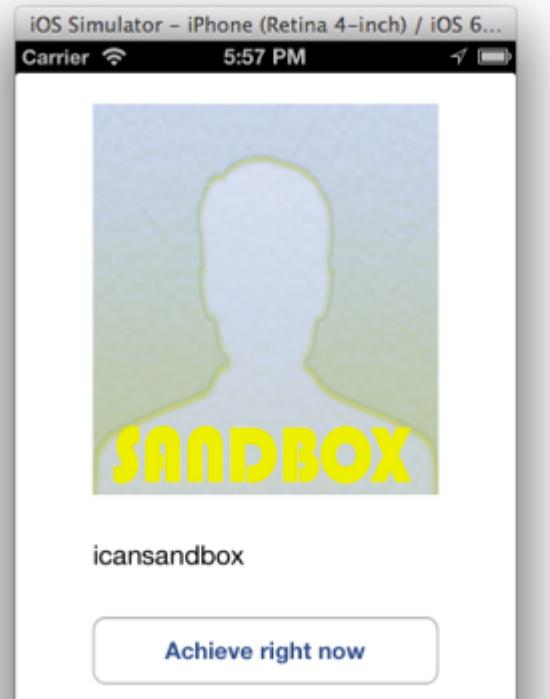
Note: the server will return error if the player didn't yet upload a picture of himself, so keep this in mind.

That's it! Compile and run, and if you have a photo set in the Game Center app, it will show up in the interface, and if not the text label will let us know.



Note: Sandbox server photos are not stored permanently. Few minutes after you set your photo on the Game Center Sandbox server it will be deleted and replaced by one of the default photos.

So don't panic if instead of your photo you see one of the default Sandbox server photos:



Showing A Notification Banner

Showing a message in the standard UI of Game Center is also utterly easy. Let's wrap up a simple method to do that. Make the following changes to **GCDViewController.m**:

```
//add inside the implementation
-(void)showMessage:(NSString*)msg {
    [GKNotificationBanner showBannerWithTitle:@"GameKit message"
                                         message:msg
                                     completionHandler:^{}];
}
```

To show a notification it's as simple as calling `+(void)showBannerWithTitle: message:completionHandler:` on the `GKNotificationBanner` class. You pass a title (the first line on the banner - it's bolded), a message (the 2nd line on the banner), and a completion handler (will be executed after the banner hides).

There are few details to talk about though. `GKNotificationBanner` actually works like a message queue. So - each message from Game Center (no matter if automatic, or created by you) shows up for few seconds, then hides (you can't control for how

long your messages appear), then if there's another message to show - it also shows, hides, and so on. Let's demonstrate that.

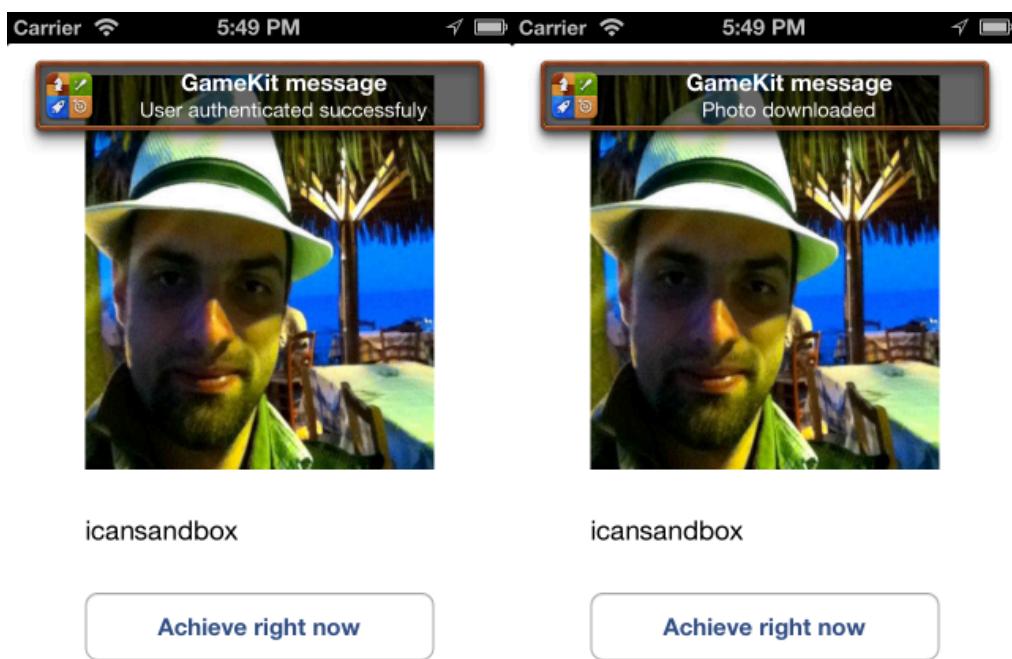
Find the comment that says "**show logged notification**". This place in the code should be executed about the same time the user has finished authenticating, so you're going to show a notification banner and see that it doesn't overlap with the automatic banner that pops up from Game Center. Replace the comment with this code:

```
[self showMessage:@"User authenticated successfully"];
```

To make the queue effect even more visible, go find the "**show photo notification later**" comment and replace it with:

```
[self showMessage:@"Photo downloaded"];
```

The three banners should be created at almost the same time, but you will see them appearing one after another - in the order that they have been added to the banner notification queue. Hit Run in Xcode and see the result:



Automatic Achievement Notifications

Let's wrap up by testing a new feature of `GKAchievement`. New in iOS 5.0 is the `showsCompletionBanner` property which is by default NO, but if you set it to YES the GameKit will take care to automatically show a notification banner when you report an achievement to have been completed.

Just add this method to the view controller's implementation:

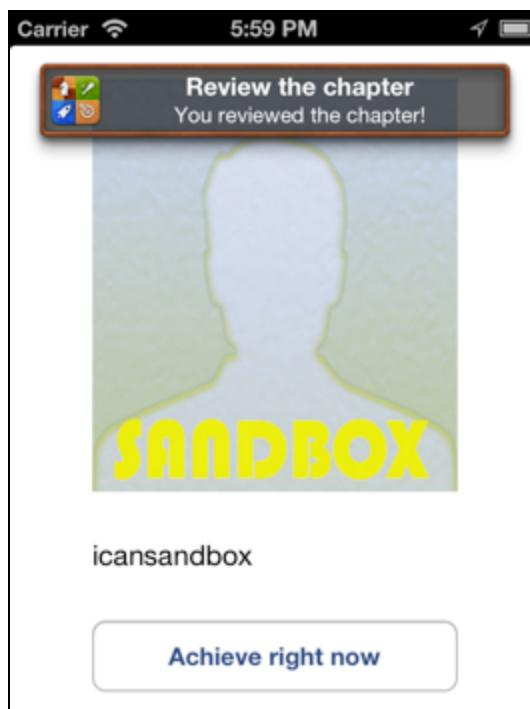
```
- (IBAction)achieveRightNow
{
    GKAchievement* achievement = [[GKAchievement alloc]
        initWithIdentifier: @"writinggamecenterapichapter"];
    achievement.percentComplete = 100;
    achievement.showsCompletionBanner = YES;

    [achievement reportAchievementWithCompletionHandler:
        ^(NSError *error){}];
}
```

You have already connected earlier the button in the UI to this `IBAction`, so let's just quickly have a look at the code. You create a new `GKAchievement`, passing in the identifier you've set in Game Center. You should change this to whatever identifier you used.

Then you set the achievement status to complete (`percentComplete` is 100) and then we also set `showsCompletionBanner` to `YES`. Finally we send the completion notification to Game Center by calling `reportAchievementWithCompletionHandler:`. The achievement is reported as completed and when GameKit receives back confirmation from Game Center it'll show the notification banner to the user.

Hit Run and give it a try:



The text, which appears on the banner, is the title and the description of the achievement - fetched from Game Center. Pretty cool, eh? :]

Where to Go From Here?

If you want to dig into this further into Game Center, be sure to check out the Turn Based Gaming chapter in this book if you haven't already.

Congratulations, you are now familiar with the cool new Game Center features in iOS 5 - we hope to see you use these in some of your apps!

Chapter 29: New Calendar APIs

By Marin Todorov

The EventKit framework in iOS allows you to access event information from a user's calendar. It can be used to add entries to the calendar for a particular date, set alarms, search for events, and much more.

With iOS5, some big changes come to this framework:

- **App-specific calendars.** Apps can now have their own calendars and of course fully manage the events inside.
- **Calendar view controllers.** If you want to let a user edit a calendar event or choose a calendar, you can now use Apple's built in view controllers!
- **Commit and roll back transactions.** When you add events to the event store, you can now choose when you're ready to commit the changes, or rollback to the last saved state. This is huge for a lot of reasons - imagine your app is creating a bunch of events, you now have the ability to create them and save them to your calendar, show a preview to the user and let him decide whether to save the changes permanently or cancel them.
- **Many other modifications.** Apple has cleaned up and made changes to the hierarchy, as well as adding various new APIs, which give broader access to third party developers to the Calendar's underlying engine.

In this chapter you are going to develop a TV Guide application, which will import show schedules of your choice to the user's calendar. In the process, you'll learn all about the new iOS 5 Calendar APIs, including adding alarms, URLs, show guide tips, and much more! This tutorial will be a lot of fun, and is really something you can take, extend and turn into an App Store wonder. Note that the new EventKit APIs don't work on the Simulator, so you'll need an actual device to work with in order to test the code. So mark the calendar - today's the day you learn the new iOS 5 Calendar APIs! :]

Getting Started

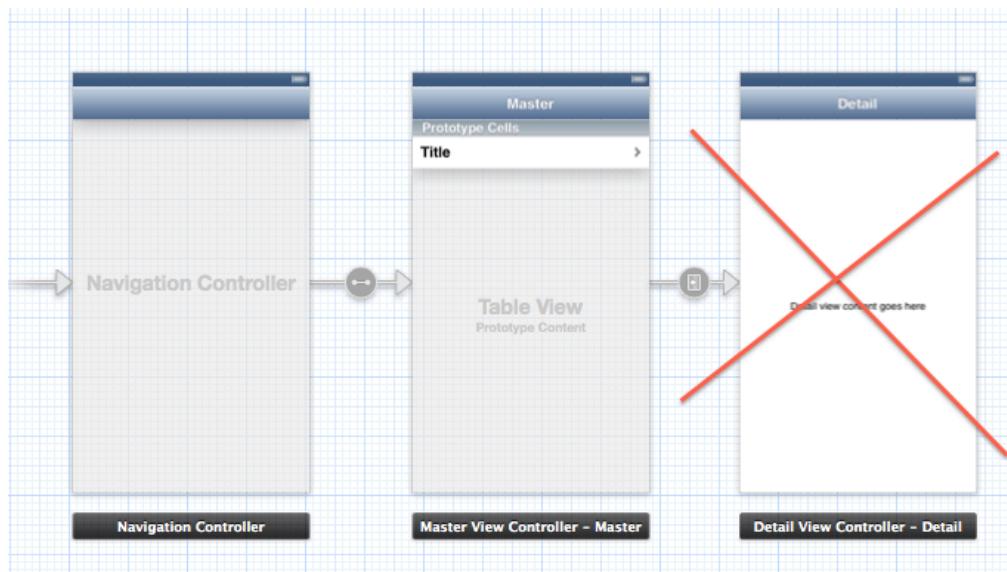
Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Master-Detail Application** template (as this will set up a table view controller for free), and click **Next**. Enter **GoodTimesTVGuide** for the product name, enter **GT** for the class prefix, select iPhone for the Device Family, and make sure that **Use Automatic Reference Counting** and **Use Storyboards** is checked (leave the other check- boxes unchecked). Click **Next** and save the project by clicking **Create**.

Select the project and select the GoodTimesTVGuide target. Open the **Build Phases** tab and open the **Link Binary With Libraries** fold. Click the plus button and double click **EventKit.framework** to add Calendar access to the project. In your project you're going to also use the view controllers Apple provides for editing events and choosing from the existing list of calendars so click again the plus button and also add the **EventKitUI.framework**.

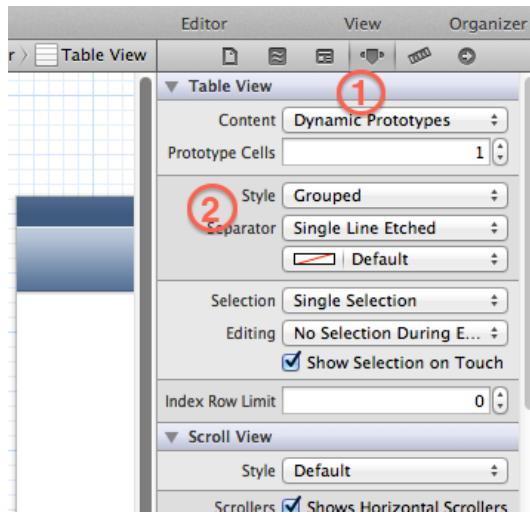
You'll use the table view controller that Xcode created for you, but you don't need the details view controller. So - time for a little cleanup. Delete the following files from the Project Navigator

- GTDetailViewController.h
- GTDetailViewController.m

Now open **MainStoryboard.storyboard** and delete the details view controller (the one on the most right).



Then zoom in on the Master View Controller and click on the table view, which appears in Interface Builder. Make sure the Attributes Inspector is open on the right (1) and from the **Style** combo box choose **Grouped** (2).



Since you are going to work with EventKit and EventKitUI you import `EventKitUI.h` in the interface of the view controller (it also imports `EventKit.h` for you.)

Also you will need one instance variable - the `shows` array. This will hold the list of TV shows your Good Times TV Guide will provide schedules for, and will be the data source for your table view.

Let's move on to the implementation file - you'll add the basics and kick it off onto the EventKit magic. Open **GTMasterViewController.m** and replace the class' boilerplate code with:

```
#import "GTMasterViewController.h"
#import <EventKitUI/EventKitUI.h> //1

@interface GTMasterViewController()
{
    NSArray* shows; //2
}
@end

@implementation GTMasterViewController

-(void)viewDidLoad
{
    [super viewDidLoad];

    self.title = @"Good Times TV Guide"; //3

    //4
    NSString* listPath = [[NSBundle mainBundle].resourcePath
        stringByAppendingPathComponent:@"showList.plist"];
    shows = [NSArray arrayWithContentsOfFile: listPath];
}
```

```
}
```

```
@end
```

Let's have a look at what this initial code does:

1. First you import the EventKitUI headers (this also include the EventKit framework for you).
2. In the interface of the class you declare an `NSArray`, which will hold the list of TV shows your TV Guide app will display.
3. You set the title of the only screen of the app.
4. Finally you load the contents of "**showList.plist**" into the `shows` class variable.
Done!

Go ahead and find the **showList.plist** in this chapter's resources and add it to your Xcode project. Let's have a look at the contents of this file:

Key	Type	Value
Root	Array	(2 items)
Item 0	Dictionary	(6 items)
title	String	How I met your father
url	String	http://www.cbs.com
startDate	String	09/29/2012 19:00 PST
endDate	String	09/29/2012 19:30 PST
dayOfTheWeek	Number	5
tip	String	Usually everyone is a good guy, besides anyone who dates
Item 1	Dictionary	(6 items)
title	String	Prison break-in
url	String	http://www.fox.com/
startDate	String	09/25/2012 14:00 PST
endDate	String	09/25/2012 15:00 PST
dayOfTheWeek	Number	2
tip	String	Jasons is secretly in love with his cell-mate, so he is gonna

For each show you have a title and the URL with more info on the web. The start and end date/time of the first episode for the season and which day of the week the show is being aired. Finally the Good Times TV Guide app also provides a tip to the user with some basic info. Hey, when you watch 15 shows you need a hint so you can remember what was it all about, right?

Although you are getting the data on the TV shows from this .plist file, in your app you could get the information from the Internet if you would like. For an example on this, check out the New Addressbook APIs chapter, where I show an example of fetching plist format data from an online web service.

Now that you have the table view source filled with data, let's add also the code to get the table running. Add these methods to the body:

```
#pragma mark - Table delegate
- (NSInteger)numberOfSectionsInTableView:(UITableView*)tableView
```

```
{  
    return 1;  
}  
  
- (NSInteger)tableView:(UITableView *)tableView  
    numberOfRowsInSection:(NSInteger)section  
{  
    return shows.count;  
}  
  
- (UITableViewCell *)tableView:(UITableView *)tableView  
    cellForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    static NSString *CellIdentifier = @"Cell";  
    UITableViewCell *cell = [tableView  
        dequeueReusableCellWithIdentifier:CellIdentifier];  
  
    // Configure the cell.  
    NSDictionary* show = shows[indexPath.row];  
    cell.textLabel.text = show[@"title"];  
  
    return cell;  
}
```

That's a bit more code - but it's all standard table view code really. In `numberOfSectionsInTableView:` you set one section for our table and in `tableView:numberOfRowsInSection:` you tell the table view that you'll have as many table rows as `shows` there are objects in the `shows` array.

`tableView:cellForRowAtIndexPath:` is pretty standard too – you dequeue a table cell, you get the `show` for that particular cell from the `shows` array, and you set the cell's text to the name of the `show`.

If you hit Run in Xcode right now you'll see that you are ready with the basic interface of the application:



When the user decide that they want to import a given show schedule into their device's Calendar, they'll have the possibility to choose between importing the events for each episode into the Good Times' own calendar or into an existing calendar already present on the device (or in the user's iCloud, Google, or another type of account).

So let's implement one more step of this process: you will show an alert to the user when they tap a table row and ask them to which calendar they'd like to import the show. Add this method to the class implementation:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [[[UIAlertView alloc] initWithTitle:@"Import tv show schedule"
        message:@"Do you want to import to:"
        delegate:self
        cancelButtonTitle:@"Existing calendar"
        otherButtonTitles:@"TV Guide's calendar", nil] show];
}
```

Now if you tap a row in the table you will see the dialogue asking you where to import:



Good progress so far! Now it's time to dig into the EventKit framework.

Managing Your Own Calendar

For this demo project you are going to create a helper class, which you can reuse in your own projects. This class will try to fetch a calendar in a couple of different ways, and if it does not succeed it'll just create a new one for you.

Let's start! Create a new file with the **iOS\Cocoa Touch\Objective-C class** template, name the class **AppCalendar**, and make it a subclass of **NSObject**.

Open **AppCalendar.h** and replace everything with this code:

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

#define kAppCalendarTitle @"Good Times TV Guide"

@interface AppCalendar : NSObject
+(EKEventStore*)eventStore;
+(EKCalendar*)calendar;
+(EKCalendar*)createAppCalendar;
```

```
@end
```

Let's see what you have for the AppCalendar class. First you import the EventKit framework, and then you define `kAppCalendarTitle` - this will be the title of your custom calendar. You define this in the header file so that if you ever need the name of the calendar from another class in your own projects, it's available as a handy constant.

You expose three class methods:

1. **eventStore**: This will return a static `EKEventStore` instance, so all parts of your code will work with the same instance of the Calendar engine.
2. **calendar**: This will return an `EKCalendar` instance for the app's own calendar.
3. **createAppCalendar**: You will only use this from within the class itself, but let's leave it in the header for convenience.

Let's start easy - with the first method. Open **AppCalendar.m** and replace the code there with this:

```
#import "AppCalendar.h"

static EKEventStore* eStore = NULL;

@implementation AppCalendar

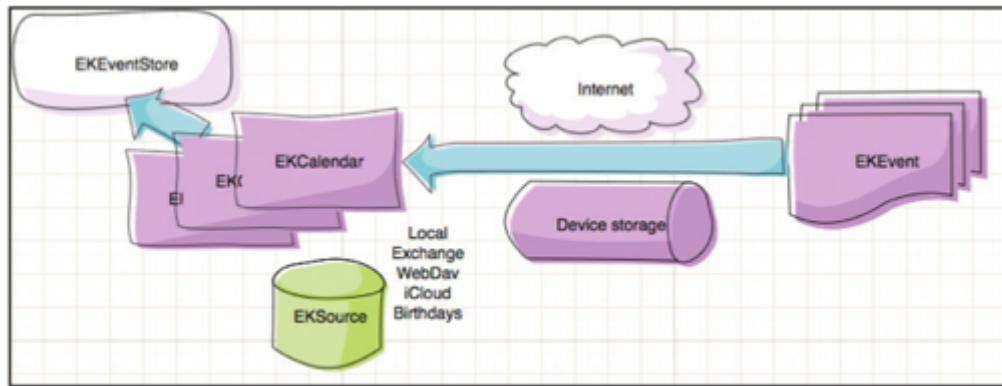
+(EKEventStore*)eventStore
{
    if (!eStore) {
        eStore = [[EKEventStore alloc] init];
    }
    return eStore;
}
@end
```

It's a good start - you have a static object called `estore`, which will hold the `EKEventStore`. Then inside the `eventStore` method, you check if `estore` has been initialized. If it has, you just return the static object; otherwise you initialize and return a new instance. This is a very simple implementation of the Singleton design pattern.

So, anywhere inside your app whenever you need to supply an event store to a method (and that's quite often) or you need to call a method on the event store you can just use:

```
[AppCalendar eventStore]
```

Before going on with coding, let's take a look at the EventKit hierarchy of classes so you'll have a better understanding of the calendar engine:



If you look at the scheme above, `EKEventStore` class is at the top of the hierarchy - `EKEventStore` is a representation of iOS' calendar engine and all the device's calendar data. So this represents the whole calendar storage on the device.

In `EKEventStore` many calendars can exist - in the EventKit context a calendar is just a way to group events, but also a way to sync events from/to different sources. `EKCalendar` contains list of events and is also connected to a source. What is the Calendar's source? It's where the calendar data comes from and where the changes you do are going to be saved. For example there's a "Home" and "Work" calendars on my device - both their sources are my iCloud account. I have also a "TODO" Calendar - this one comes from my iMac.

So depending on the type of calendar source (`EKSource`) the calendar is filled with events from the appropriate location - the web, exchange server or the local storage. The `EKEvent` class inherits from the new for iOS 5.0 abstraction `EKCalendarItem` and generally contains all the data you need for a calendar entry - start and end time, alarms, notes, etc.

In iOS 5 third party apps are way more flexible about using the EventKit framework than before. They can work with different calendars, query the calendars' source, create and delete custom calendars.

Let's go back to the code and add the method to create a new calendar. Inside **AppCalendar.m** add this new method:

```
+ (EKCalendar*)createAppCalendar
{
    EKEventStore *store = [self eventStore];

    //1 fetch the local event store source
    EKSource* localSource = nil;
    for (EKSource* src in store.sources) {
        if (src.sourceType == EKSourceTypeCalDAV) {
```

```
        localSource = src;
    }
    if (src.sourceType == EKSourceTypeLocal &&
        localSource==nil) {
        localSource = src;
    }
}

if (!localSource) return nil;

//2 create a new calendar
EKCalendar* newCalendar = [EKCalendar
calendarForEntityType:EKEntityTypeEvent eventStore: store];

newCalendar.title = kAppCalendarTitle;
newCalendar.source = localSource;
newCalendar.CGColor = [[UIColor colorWithRed:0.8 green:0.251
blue:0.6 alpha:1] /*#cc4099*/ CGColor];

//3 save the calendar in the event store
NSError* error = nil;
[store saveCalendar: newCalendar commit:YES error:&error];
if (!error) {
    return nil;
}

//4 store the calendar id
NSUserDefaults *prefs= [NSUserDefaults standardUserDefaults];
[prefs setValue:newCalendar.calendarIdentifier
forKey:@"appCalendar"];
[prefs synchronize];

return newCalendar;
}
```

Let's go over this code: First you get the `EKEventStore` instance from the `eventStore` method you did earlier. And then you do few more steps in order to create the calendar:

- You iterate over the `EKEventStore`'s available sources and when you reach the source with `sourceType` of `EKSourceTypeLocal` you store it in `localSource`; this is the source type you are going to use for your app's calendar.
- The possible `EKSource` values are as follows: `EKSourceTypeLocal` (local storage), `EKSourceTypeExchange` (Exchange server), `EKSourceTypeCalDAV` (web calendar), `EKSourceTypeMobileMe` (iCloud), `EKSourceTypeSubscribed` (subscribed calendars),

and `EKSourceTypeBirthdays` (the special Birthdays calendar for AddressBook contacts' birthdays).

- You create a new `EKCalendar` instance by simply calling `[EKCalendar calendarWithEventStore: store]`. Then you assign the calendar's title and the source that you just stored in `localSource`. The events inside the new calendar will all have background color defined by the property `CGColor`.
- You call `saveCalendar:commit:error:` on the event store instance to create and save permanently the new calendar. Note the commit parameter. In iOS 5 you can mark changes to the event store as pending or immediate - if you pass `YES` the changes are saved immediately, if you pass `NO` changes will pile up until you call `commit` on the `EKEventStore` instance, or you call `reset` to discard them.
- Finally if the calendar is successfully created and saved you store its unique calendar identifier in the user preferences, so you can look it up by its id next time you need to access it.

This way you create a fully featured calendar and you can see it and modify it from iOS's Calendar application too:



Your custom calendar could appear in different calendar groups "From my Mac" if you sync calendars, "From my iPhone" if you don't, or "iCloud" if you have iCloud enabled.

Note: At the time of this writing there are reproducible issues, which could cause your custom calendars to be invisible in those lists, especially if iCloud is turned on, and/or the user switches on and off iCloud support.

Creating a calendar is pretty easy, but don't want to create a new calendar every time the app runs, right? So, add a new method to **AppCalendar.m** to get the calendar, loading it if it was created already or calling your method to create a new one otherwise:

```
+ (EKCalendar*)calendar
{
    //1
    EKCalendar* result = nil;
    EKEventStore *store = [self eventStore];

    //2 check for a persisted calendar id
    NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
    NSString *calendarId = [prefs objectForKey:@"appCalendar"];

    //3
    if (calendarId && (result =
        [store calendarWithIdentifier: calendarId])) {
        return result;
    }

    //4 check for a calendar with the same name
    for (EKCalendar* cal in
        [store calendarsForEntityType:EKEntityTypeEvent]) {

        if ([cal.title compare: kAppCalendarTitle]==NSOrderedSame) {
            if (cal.mutable == NO) {
                [prefs setValue:cal.calendarIdentifier
                    forKey:@"appCalendar"];
                [prefs synchronize];
                return cal;
            }
        }
    }

    //5 if no calendar is found whatsoever, create one
    result = [self createAppCalendar];

    //6
    return result;
}
```

```
}
```

OK, let's dig into the code by numbers:

1. First you prepare the object to hold the method's result - `result`. Also you use the previously defined `eventStore` method to get hold of an `EKEventStore` instance.
2. You read the key `appCalendar` from the user preferences. If the application has previously created a calendar, you would have saved the calendar's unique identifier here.
3. If there's a unique id found and `calendarWithIdentifier:` gives back a calendar instance for that id, then just return this calendar as a result.
4. If the calendar id is missing from user preferences for some reason, but the app has indeed in the past created a calendar, you can find it by looping over the available calendars and comparing their titles to our desired calendar title. If you find a matching calendar title you also check if the calendar is writable by the current app. If it is, you return that calendar right away and save its unique identifier for future reference.
5. When no calendar is found, you call the previously defined method `createAppCalendar` to create one for your app to use.
6. Finally you return the result. Notice if the new calendar creation failed the `result` object will be `nil`, so the caller can do some error handling.

Pew, that was a lot of code! But hey, now you have a class to take care of managing the application's calendar under the hood, so you won't have to implement more logic in the view controller, and that's great. This is a stand-alone helper class you can reuse anywhere.

Adding Events to Your Calendar

Let's continue by adding a method to create the calendar items for the show the user has selected. To do that you'll need to handle the event when the user taps a button on the alert asking him where they want to import the selected show schedule.

Open **GTMasterViewController.h** modify the interface to mark it as implementing the `UIAlertViewDelegate` protocol, like so:

```
@interface GTMasterViewController() <UIAlertViewDelegate>
```

Then add the delegate method inside **GTMasterViewController.m**:

```
#pragma mark - Alertview delegate
-(void)alertView:(UIAlertView *)alertView
didDismissWithButtonIndex:(NSInteger)buttonIndex
```

```
{  
    if (buttonIndex==0) {  
        //show calendar chooser view controller  
    } else {  
        //use the app's default calendar  
        int row = [self.tableView indexPathForSelectedRow].row;  
        [self addShow:shows[row]  
            toCalendar:[AppCalendar calendar]];  
    }  
}
```

If `buttonIndex` is zero, the user chose to import into an already existing calendar. You'll add the code to show the calendar chooser here later on.

If `buttonIndex` equals one, the user chose to import into the app's own calendar. So here we get the selected row index and store it in the variable "row". You then call the method `addShow:toCalendar:` (which you haven't written yet), and pass as parameters 1) the `NSDictionary` with data about the show and 2) the application's own calendar from our new and shiny `AppCalendar` class.

You should have an error right now in Xcode, since we still didn't import the `AppCalendar` class. Scroll to the top of the file and add the import:

```
#import "AppCalendar.h"
```

And add the initial body of the method inside the class implementation:

```
#pragma mark - Calendar methods  
-(void)addShow:(NSDictionary*)show  
toCalendar:(EKCalendar*)calendar  
{  
    EKEvent* event = [EKEvent eventWithEventStore:  
                      [AppCalendar eventStore]];  
    event.calendar = calendar;  
  
    NSDateFormatter* frm = [[NSDateFormatter alloc] init];  
    [frm setDateFormat:@"MM/dd/yyyy HH:mm zzz"];  
    [frm setLocale:[[NSLocale alloc]  
                  initWithLocaleIdentifier:@"en_US"]];  
    event.startDate = [frm dateFromString: show[@"startDate"]];  
    event.endDate = [frm dateFromString: show[@"endDate"]];  
}
```

You start with the basics. `[EKEvent eventWithEventStore:]` creates a new event in the given event store, and by setting the `calendar` property you tell the event in which calendar it belongs.

Since the show start and end dates are saved in a plist file like strings we need to turn those strings into `NSDate` objects. We create an `NSDateFormatter` and tell it to expect string input in the format "MM/dd/yyyy HH:mm zzz" - if you have a look inside the plist file you'll see that the dates are in this particular format, for example - 09/29/2012 19:00 PST (oooh, 24 hour input - how European! :)) Using the date formatter you turn the two strings from the plist file to `NSDate` objects and you set `startDate` and `endDate` on the event.

In the iOS 5 SDK many of the properties that belonged to `EKEvent` were moved to another class: `EKCalendarItem` (`EKEvent` now inherits from this). `EKCalendarItem` represents an abstract calendar entry - and the code for creating `EKEvents` is actually the same as before.

The event time is set; so let's add some more details. Add this code at the end of `addShow:toCalendar:`:

```
event.title = show[@"title"];
event.URL = [NSURL URLWithString:show[@"url"]];
event.location = @"The living room";
event.notes = show[@"tip"];
```

More good stuff - the title of the show goes as title of the event, you set also the URL property of the event, and the location is just a string - for all shows you set it to "The living room" and finally you copy the "tip" field into the "notes" property of the event.

Here's how the event will eventually look like in the Calendar app:



That's all great but the shows run an episode each week and you are about to create only one calendar entry for the very first episode only!

To fix this, you can set the event's "recurrence" - a rule about how much or how long should the event occur in the calendar. You can make an event occur certain days of the week, some month days, or given weeks in the year. For the full list of options, check out the reference docs for `-[EKRecurrenceRule initRecurrenceWithFrequency:interval:daysOfTheWeek:daysOfTheMonth:monthsOfTheYear:weeksOfTheYear:daysOfTheYear:setPositions:end:]`. That's a mouthful, but it's a great and very flexible API :)

For this app, you will make the TV show event occur weekly for 3 months, so add this code to the bottom of `addShow:toCalendar:`:

```

NSNumber* weekDay = [show objectForKey:@"dayOfTheWeek"]; //1

EKRecurrenceDayOfWeek* showDay =
[EKRecurrenceDayOfWeek dayOfTheWeek: [weekDay intValue]];

//2
EKRecurrenceEnd* runFor3Months =
[EKRecurrenceEnd recurrenceEndWithOccurrenceCount:12];

//3

```

```
EKRecurrenceRule* myReccurrence = [[EKRecurrenceRule alloc]
    initRecurrenceWithFrequency:EKRecurrenceFrequencyWeekly
    interval:1
    daysOfTheWeek:@[showDay]
    daysOfTheMonth:nil
    monthsOfTheYear:nil
    weeksOfTheYear:nil
    daysOfTheYear:nil
    setPositions:nil
    end:runFor3Months];

[event addRecurrenceRule: myReccurrence];
```

You get the day of the week the show runs from the property list dictionary (1 means Sunday, 2 – Monday, and so forth) and then you create the recurrence rule:

1. `EKRecurrenceDayOfWeek` is a helper class to represent (surprise!) a day of the week. You get an instance from the `dayOfWeek:` method and you pass as parameter an integer (1 also means *Sunday* here).
2. `EKRecurrenceEnd` is a helper class to represent an occurrence period - you can either call `recurrenceEndWithEndDate:` and set a date when the event stops occurring, or you can get an instance by calling `recurrenceEndWithOccurrenceCount:` and supplying it with a number of occurrences. You set 12 occurrences for our weekly event, so it runs for 3 months.
3. The `EKRecurrenceRule` initializer has a ton of parameters, and by supplying different input you can create many different rules. But for your example you need a single simple rule - run once a week on a given day for 3 months. So set the frequency to `EKRecurrenceFrequencyWeekly`, the interval to 1 (it means run each week, not every other week or another interval), the days of the week to an array with 1 object (`showDay`), and the end to the previously created recurrence rule.

Here's how it will look in the calendar:



You might notice that in the plist file you can actually see that "Prison break-in" runs on Sundays, but the event in my calendar shows up on Mondays - a bug maybe? No actually - it's a feature (lol)!

Have a look again at the date info in the plist file. All dates contain also a time zone - 09/25/2011 14:00 PST. So the times are within the PST time zone, and my device is located in a time zone 11 hours ahead of PST, so when it's Sunday 2:00 PM in the PST time zone, it is actually already Monday 1:00 AM in here - so the EventKit framework takes care of all this by itself under the hood, so you won't have to! Ain't that just amazing!

Finally you will save the event to the event store and show the event edit dialogue to the user, so they can make changes to the recurring event if they'd like to.

```
//1 save the event to the calendar
NSError* error = nil;

[[AppCalendar eventStore] saveEvent:event span:EKSpanFutureEvents
commit:NO error:&error];

//2 show the edit event dialogue
EKEVENTEditViewController* editEvent =
[[EKEVENTEditViewController alloc] init];

editEvent.eventStore = [AppCalendar eventStore];
```

```
editEvent.event = event;
editEvent.editViewDelegate = self;

[self presentViewController:editEvent animated:YES
completion:nil];
```

First you call `saveEvent:span:commit:error:` (notice how you skip on the error handling to keep things moving, for a real app you MUST implement it). It takes as input the event object, a span value (will explain in a second), and whether to commit the changes immediately or hold them.

So what happens if you keep the changes? Well the changes are present in your `EKEventStore`, but not in other `EKEventStore` instances you can have in the same application or say in the actual iOS Calendar app. So I hope now you realize why you needed a static instance of `EKEventStore` in first place - so you can make sure you always use the same object and you for sure see the pending changes from within your own app.

For the `span` parameter you can pass either `EKSpanThisEvent` or `EKSpanFutureEvents`. `EKSpanFutureEvents` will update with the changes the user makes all future event entries in the calendar (that is - if it is a recurring event) and of course `EKSpanThisEvent` will instruct the EventKit to update only the very instance of the event that has been passed to the edit dialogue.

Then you get an instance of `EKEVENTEditViewController`, which is a new iOS 5 view controller, which gets an event and displays the default event edit form to the user. You need to present the view controller as a modal view controller (i.e. can't push it to the navigation controller stack). The dialogue needs a delegate, so you set the `editViewDelegate` to `self` (your view controller).

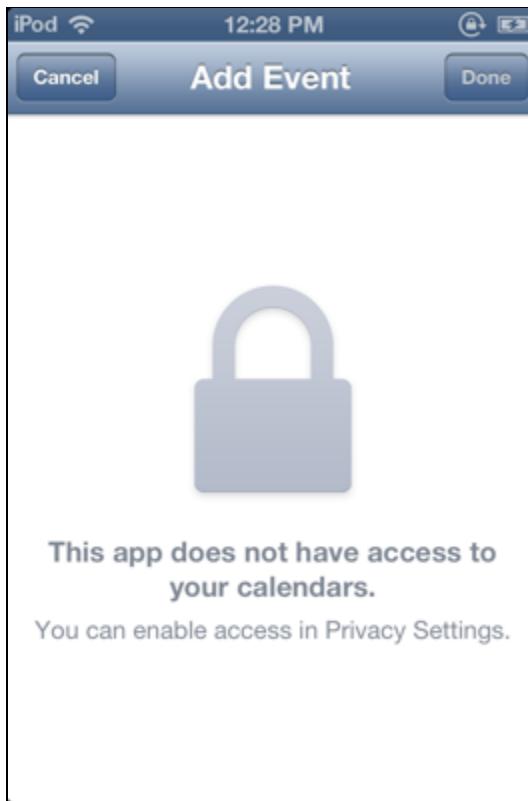
So, open up `GTMasterViewController.h` and mark that the view controller, implements the `EKEVENTEditViewDelegate`, like so:

```
@interface GTMasterViewController() <UIAlertViewDelegate,
EKEVENTEditViewDelegate>
```

Compile and run, and w00t...

Security, Security

At this point you will see the table and you can tap a show in the list. Adding event though you probably won't be able to do. With iOS6 there's new permissions system and when you try to create a new event you will see this screen:



No fear though – getting permissions to use the calendar store is pretty easy.

Find the `viewDidLoad:` method in **GTMasterViewController.m** and add this line to get permissions to access the calendar:

```
[[AppCalendar eventStore]
    requestAccessToEntityType:EKEntityTypeEvent
    completion:^(BOOL granted, NSError *error) {
        NSLog(@"Permission is granted : %@", (granted)?@"YES":@"NO");
    }];
}
```

This method tries to grant the app permissions to the calendar and then executes the block of code you provide. The `granted` parameter (a Boolean) tells you whether your app has rights to access the calendar; if `granted` equals NO you should also check the `error` parameter.

This method not necessarily shows any UI to the user. It might be that permission is granted to your app automatically, so the user does not even know that your app is being granted Calendar rights in the background.

You can now build and run again the project!

Finally Something Visible!

Yeah - I admit I pushed you into a very long stretch, but hey - no pain, no gain, right?



Just how cool is that? You have everything pre-filled and you just need to add the `EKEventEditViewDelegate` methods so you can handle taps on *Cancel* or *Done*. Let's dig in!

The protocol requires you to provide an `EKCalendar` instance that'd be the default calendar for when the user is adding new events. So let's have this method just return our app's calendar (you actually won't need this method at all in the final project, but let's conform to the protocol.)

```
#pragma mark - Edit event delegate
-(EKCalendar*)eventEditViewControllerDefaultCalendarForNewEvents
:(EKEVENTEditViewController *)controller
{
    return [AppCalendar calendar];
}
```

And finally the `EKEventEditViewDelegate` method to save the event or cancel it:

```
- (void)eventEditViewController:  
    (EKEEventEditViewController *)controller  
didCompleteWithAction:(EKEEventEditViewAction)action  
{  
    NSError* error = nil; switch (action) {  
        case EKEEventEditViewActionSaved:  
            [[AppCalendar eventStore] commit:&error];  
            break;  
        case EKEEventEditViewActionCanceled:  
            [[AppCalendar eventStore] reset];  
        default:break;  
    }  
  
    [controller dismissViewControllerAnimated:YES  
        completion:nil];  
}
```

When the user taps either *Cancel* or *Done*, this method will be invoked and the action parameter will be either `EKEEventEditViewActionCanceled` or `EKEEventEditViewActionSaved` respectively.

When the action is `EKEEventEditViewActionSaved`, you just call `commit:` on the event store object, so it'll save all pending changes permanently to the disc. When the user cancels you just call `reset` to rollback the pending event. Those of you with experience in business database application development would already have some good ideas how to use that feature, since you are probably already accustomed to the whole "commit / rollback" paradigm.

The last thing you do is actually dismissing the modal view controller.

So ... actual saving is pretty easy isn't it? It certainly should feel so, when you develop software modularly so you build up to a point when executing a complicated operation actually takes very few lines of code! :)

So let's give it a try! Tap a show and also tap *Done* in the event edit dialogue. Then fire up the iOS Calendar app (remember, you must use an actual device!) and sure enough there are the events shining in the calendar!



Importing Events Into an Existing Calendar

Let's wrap-up the basic functionality by adding support for storing the show schedule into existing calendars.

Find the comment saying "**show calendar chooser view controller**" and add the following code after the comment:

```
EKCalendarChooser* chooseCal = [[EKCalendarChooser alloc]
    initWithSelectionStyle:EKCalendarChooserSelectionStyleSingle
    displayStyle:EKCalendarChooserDisplayWritableCalendarsOnly
    eventStore:[AppCalendar eventStore]];

chooseCal.delegate = self;
chooseCal.showsDoneButton = YES;

[self.navigationController pushViewController:chooseCal
    animated:YES];
```

Pretty easy - you just get an instance of `EKCalendarChooser` and configure it through the initializer.

The selection style parameter can be either `EKCalendarChooserSelectionStyleSingle` to allow only one selected calendar at a time or `EKCalendarChooserSelectionStyleMultiple` to allowing multiple selections. The display style parameter can be either `EKCalendarChooserDisplayAllCalendars` to display all available calendars or `EKCalendarChooserDisplayWritableCalendarsOnly` to show only the calendars you can save events to - the latter is definitely what you want.

The `EKCalendarChooser` needs a delegate - you set it to `self`. You can either react to changes of the selected calendar or you can wait for the user to tap the "Done"

button and check their selection - you set `showsDoneButton` to `YES`, so you are going to implement the second approach.

This view controller you have to push to the navigation controller's stack, so you pass it to `pushViewController:animated:`. And what the user will see is the default calendar selection dialogue from the iOS Calendar application:



Note: This screen may look different on your device, depending whether you have an active iCloud account and other settings.

Scroll up to the interface declaration and right after `EKEventEditViewDelegate` add yet another protocol:

```
@interface GTMasterViewController() <UIAlertViewDelegate,  
EKEventEditViewDelegate, EKCalendarChooserDelegate>
```

Then add the implementation of this protocol:

```
#pragma mark - Calendar chooser delegate  
- (void)calendarChooserDidFinish:  
    (EKCalendarChooser*)calendarChooser  
{  
    //1
```

```
EKCalendar* selectedCalendar =
[calendarChooser.selectedCalendars anyObject];

//2
int row = [self.tableView indexPathForSelectedRow].row;

//3
[self addShow: [shows objectAtIndex:row]
toCalendar: selectedCalendar];

//4
[self.navigationController popViewControllerAnimated:YES];
}
```

And what this code does is:

1. The `selectedCalendars` property of `EKCalendarChooser` is an `NSSet` of the selected calendar objects. In your case you allow only single selection, so the `anyObject` method will give you back the selected calendar.
2. You get the selected row index in the table and store it into the `row` variable.
3. You call `addShow:toCalendar:` just like before, but this time you pass the selected calendar as parameter.
4. Finally just pop the calendar chooser view controller - and you are done!

OK - you have the demo project ready. Fire it up - you choose to import either to the app's calendar or another existing calendar, and everything works as expected!

What About Alarms?

The most natural thing for me as a creator of the Good Times TV Guide would be to want the calendar schedule to alert the users before the show starts, so they don't miss it.

However it seems the combination of 1) uncommitted event, 2) the edit event dialogue and 3) event with alarms - doesn't really work. This is a bug right now - hopefully it'll be fixed in the future, but since it is present right now - you have to take a funky approach to adding the alarms functionality to the application.

So the solution you are going to go for, is to actually commit the event to the event store and then in the case the user hits "Delete event" in the dialogue we'll just remove the event from the calendar. Let's do that real quickly!

Find this line in the code:

```
[[AppCalendar eventStore] saveEvent:event span:EKSpanFutureEvents
commit:NO error:&error];
```

Change "no" to "yes" - that'll make the event store permanently save the event to the device. No worries though - we have the event at hand, so you can delete it any time.

But first let's deal with a hidden issue. The event is saved to the event store, so when the edit event dialogue pops up if the user taps cancel the dialogue will pass the `EKEEventEditViewActionCancelled` action (as before), but if the user just taps *Done* without making modifications to the prefilled data - again an `EKEEventEditViewActionCancelled` event will get passed back to the delegate.

Ack! So as a workaround, you have to remove the cancel button and leave the user with the choice to either 1) tap the "*Done*" button or 2) tap the "*Delete event*" button on the bottom.

Look at the code and find the line:

```
[self presentViewController:editEvent animated:YES  
completion:nil];
```

There's a new method in iOS 5.0 for presenting modal view controllers (and Apple recommends using it) called `presentViewController:animated:completion:` - the difference being that it also takes a code block to execute when finished presenting the view controller. So replace the line above with:

```
[self presentViewController:editEvent animated:YES completion:^{
    UINavigationItem* item = [editEvent.navigationBar.items
        objectAtIndex:0];
    item.leftBarButtonItem = nil;
}];
```

This will present the view controller and also inside the block it'll remove the "*Cancel*" button from the dialogue.

So without the "*Cancel*" button you can adjust also the logic inside `eventEditViewCon troller:didCompleteWithAction:`. Remove the "switch" statement and replace it with this new one:

```
switch (action) {
    case EKEEventEditViewActionDeleted:
        [[AppCalendar eventStore] removeEvent:controller.event
            span:EKSpanFutureEvents error:&error];
    default:break;
}
```

How does that work?

- If the user taps "*Delete event*" - the application deletes all event occurrences from the calendar.

- If the user taps "Done" and he hasn't done any changes - don't do anything.
- If the user taps "Done" and he's done modifications to the event - don't do anything: the edit event dialogue detects that and asks the user which occurrences he wants to edit and saves the changes to the event automatically.

Adding Alarms to the Calendar

Finally you can add alarms to the calendar entries! Go to the code where you create the event and below the line `event.calendar = calendar;` add also this code:

```
EKAlarm* myAlarm = [EKAlarm alarmWithRelativeOffset: - 06*60];
[event addAlarm: myAlarm];
```

Here you create an alarm instance - `alarmWithRelativeOffset:` expects a negative integer and that's the offset from the event's start time when the alarm will be triggered. So by passing "`- 06*60`" as relative offset you can set the alarm to fire up 6 minutes before the show starts.

That's all! You can give the alarms a try and you are all set for today!

Where to Go From Here?

If you want to get more practice with EventKit and the new iOS 5 APIs, you can extend this project with the following new features:

- Implement a web service with actual show schedules and connect the app to it (have a look at the New Address Book APIs chapter for guidance)
- Have a look at the reference for
`initRecurrenceWithFrequency:interval:daysOfTheWeek:daysOfTheMonth:monthsOfTheYear:weeksOfTheYear:daysOfTheYear:setPositions:end:` - you can define wonders with it!
- Query the calendar to show to the user the next shows he should watch.

Chapter 30: Using the New Linguistic Tagger API

By Marin Todorov

When I first found the `NSLinguisticTagger` class in the iOS5 documentation I was really excited about it. I've done in the past work on a science project where I had to work on a lexical aligner and analyzer software, so I was naturally intrigued by what features are Apple integrating in iOS5.

At that time of course still nobody knew about Siri, so it is kind of obvious now that this API is definitely one of the background players behind the Siri software. Unfortunately the `NSLinguisticTagger` doesn't provide speech recognition as we all would've loved to, but just makes educated guesses at what parts of the speech the words in a sentence are and tries to provide some more information about them.

You see the problem with correctly analyzing English text is that the same words in "right turn" and "turn right" have different roles - "turn" is once a verb and another time a noun. In other languages adjectives, verbs and nouns have distinctive recognizable forms, so language analysis is much easier.

So while `NSLinguisticTagger` can make educated guesses, it's not perfect! But it can definitely be handy, as you'll see in this tutorial.

In this tutorial, you are going to develop an RSS reader, but with a twist - next to the title of the article, it will also show its "true topic".

The "true topic" app will use the `NSLinguisticTagger` to grab all nouns from a web page (as nouns are what we will be interested in order to detect the real matter behind the text) and then count how many times they are mentioned. The top 5 nouns will be said to be the true topic of the article!

Getting Started

Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Master-Detail Application** template (as this will set up a table view controller for free), and click **Next**. Enter **TrueTopic** for the product name, enter **TT** for the class prefix, select iPhone for the Device Family, and make sure that **Use Automatic Reference Counting** and **Use Storyboards** is checked

(leave the other checkboxes unchecked). Click *Next* and save the project by clicking *Create*.

If you've worked with XML in iOS apps in the past, you've probably used an XML library (and not the built it `NSXML` class). Since we're looking into new APIs I'm going to show you a new and fun XML library in Objective-C, which chances are you still didn't use, called "RaptureXML".

Let's add it to the project right now so we get this out of the way:

- Open in your browser the [RaptureXML GitHub page](#).
- Click the "ZIP" button to download a .zip file to your computer.
- Unarchive the downloaded file and open the RaptureXML folder.
- Inside there's another folder called RaptureXML (with `RXMLElement.h` and several other files inside), drag it from Finder and into Xcode's Project Navigator.
- Make sure the "Copy items into..." checkbox is checked and click Finish.

You'll enjoy using RaptureXML because it has a nice API that will make sure we write the least code possible to parse XML and focus on our real task at hand.

RaptureXML requires the **libz** and **libxml2** library, so let's add it to the project.

1. Click the project file in the Project navigator and select the TrueTopic target.
2. Click on the "Build phases" tab and unfold the "Link binary with libraries" strip.
3. Click on the "+" button and double-click "libz.dylib" from the list; also do that for libxml2.dylib.

Next switch to the "Build settings" tab and in the search field enter "Header Search Paths". You will see then the Header Search Paths field, so double click on its value and add a new search path: "/usr/include/libxml2" (without the quotes).

Let's do one more thing - I always do this as part of the project setup. Let's add a shortcut for spawning a new background queue (so you can do heavy lifting in the background, while the user UI is responsive and nice) and define the RSS url.

Open up **TTMasterViewController.m** and at the top of the file add:

```
#define kBgQueue
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
#define kRSSUrl [NSURL URLWithString:
@"http://feeds.feedburner.com/RayWenderlich"]
```

Your project should be all set now – if you hit Run you should see an empty table:



Fetch the Remote RSS Feed

Next you are going to add the logic to fetch and parse a remote RSS feed from raywenderlich.com.

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template, set the class name to **ArticleData**, and make sure it's a subclass of **NSObject**. This class is just going to hold few strings for us - the title of the article, the true topic and the link to the article page.

Replace the contents of **ArticleData.h** with the following:

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

@interface ArticleData : NSObject

@property (strong) NSString* title;
@property (strong) NSString* topic;
@property (strong) NSString* link;

@end
```

It's just a simple class to hold three properties.

Now open up **TTMasterViewController.m** and add this two imports below the defines at the top:

```
#import "RXMLElement.h"
#import "ArticleData.h"
```

Next you need to fetch the RSS feed, iterate over the "item" elements inside the feed, and the articles array with ArticleData instances. You are going to do that work in `viewDidLoad`; so replace the boilerplate `viewDidLoad` with this new implementation:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    _objects = [NSMutableArray array];
    dispatch_async(kBgQueue, ^{
        //work in the background
        RXMLElement *xml = [RXMLElement
            elementFromURL: kRSSUrl];
        NSArray* items = [[xml child:@"channel"]
            children:@"item"];
        //iterate over the items

        //reload the table

    });
}

```

Initially you don't do so much: you initialize the `articles` array as you are going to be adding items inside. Then in a background queue (using the `kBgQueue` macro you defined earlier) you take on XML parsing. As you see the `RXMLElement` class has a class method `elementFromURL:` which takes an `NSURL` and gives you back XML object ready to work with. On the next line you just call:

```
[xml child:@"channel"]
```

which returns `RXMLElement` object for the channel child tag and you immediately call the following on it:

```
children:@"item"];
```

`children:` gives you back array of XML elements. I told you RaptureXML is going to save you a bunch of XML parsing code! Now replace the "**//iterate over the items**" comment with the actual code:

```

for (RXMLElement *e in items) {
    //iterate over the articles
    ArticleData* data = [[ArticleData alloc] init];
    data.title = [[e child:@"title"] text];
    data.link = [[e child:@"link"] text];
    [_objects addObject: data ];
}

```

Pretty simple - inside the loop you just create an `ArticleData` instance and set the title and link. You then add the new item to the `articles` list.

Finally, you have to show the results on the screen, so you'll have to reload the table view. Note that you have to do this on the main queue, because you cannot modify UI elements from a background thread. So replace the "**//reload the table**" comment with the following code:

```
dispatch_async(dispatch_get_main_queue(), ^{
    [self.tableView reloadData];
});
```

That's all there is to an RSS reader! There's just one small thing: you need to replace the boilerplate code in `tableView:cellForRowAtIndexPath:` to show the article titles with this:

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

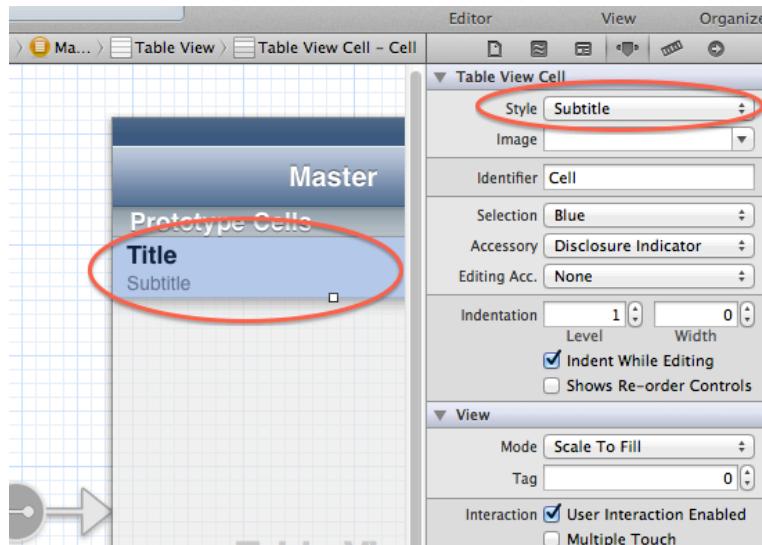
    // Fill in the article data
    ArticleData* data = _objects[indexPath.row];
    cell.textLabel.text = data.title;
    cell.detailTextLabel.text = data.topic;

    return cell;
}
```

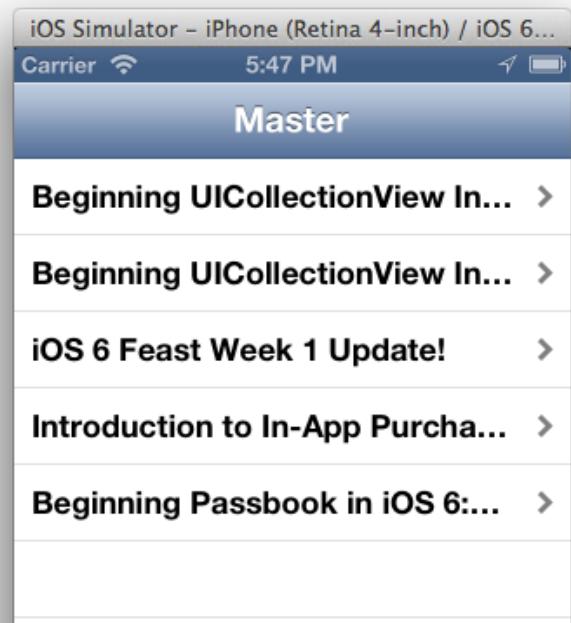
You just fetch the correct `ArticleData` object from the `_objects` array and then you set the `title` of the cell to the article's title and the `detailLabel` of the cell to the `topic` property of the `ArticleData`. (You'll see what the `topic` property is all about in a minute.)

As you already might have guessed from the code you will be using a standard cell style including a subtitle, so you will need to set your cells to feature a label for it.

Open up **MainStoryboard.storyboard**, and zoom-in on the table view in Master View Controller. Select the cell in the table view and choose "Subtitle" as style:



Now you can also run the project and your RSS reader should already work just fine!



You can move on to working with the fetched data!

Putting NSLinguisticTagger to Work

In this part of the tutorial you are going to build a pretty sophisticated class, which will fetch a remote article from a web page, strip out the HTML and get out the nouns out of it. All in a convenient one line call.

Let's start! Create a new file with the **iOS\Cocoa Touch\Objective-C class** template, set the class name to **TagWorker**, and make sure it's a subclass of **NSObject**.

Inside **TagWorker.h** let's define the single method, which will do all the aforementioned work:

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

typedef void (^TaggingCompletionBlock)(NSArray* words);

@interface TagWorker : NSObject

-(void)get:(int)number_ofRealTopicsAtURL:(NSString*)url
completion:(TaggingCompletionBlock)block;

@end
```

I chose to define a new type `TaggingCompletionBlock`, so my method definition looks clearer - the type basically defines a block, which takes one argument in an `NSArray`. The method we have on the `TagWorker` class expects the number of topics to return, a URL to fetch and a block, which to invoke upon completion.

Now - move on to the implementation (and this is going to take some while, be prepared). Open up **TagWorker.m** and modify the class as:

```
#import "TagWorker.h"

@interface TagWorker()
{
    NSLinguisticTagger* tagger;
    NSMutableArray* words;
}
@end

@implementation TagWorker

-(void)get:(int)number_ofRealTopicsAtURL:(NSString*)url
completion:(TaggingCompletionBlock)block
{
    //initialize the linguistic tagger
    tagger = [[NSLinguisticTagger alloc] initWithTagSchemes:
    @[NSLinguisticTagSchemeLexicalClass, NSLinguisticTagSchemeLemma]
    options: kNilOptions];

    //setup word and count lists
    words = [NSMutableArray arrayWithCapacity:1000];
}
```

```
}
```

```
@end
```

You have two instance variables - one is the tagger object itself and the other is a list of the word objects. You will use the words array to store what you find as you iterate over the page's contents.

Now let's look at initializing the tagger class. It takes an array of schemes and an options parameter. You pass `nil` for the options, but what in the heck are schemes?!

Depending on what you pass in for schemes, when you invoke calls to the tagger, you will get back parts of the text classified in different manner:

- **NSLinguisticTagSchemeTokenType** will classify the text parts as words, whitespace, and punctuation.
- **NSLinguisticTagSchemeLexicalClass** will tag the text parts as parts of the speech: nouns, verbs, adjectives, adverbs, determiners, particles, etc. etc.
- **NSLinguisticTagSchemeNameType** will get you known personal, organizations or places names.
- **NSLinguisticTagSchemeLanguage** gets you the tagger's best guess what language the word belongs to.

There are even more schemes to look into, but we're going to use only `NSLinguisticTagSchemeLexicalClass` as we will be interested in the nouns of our text.

The tagger however parses plain text, and the article is going to be fetched from a web page so it will be actually HTML source code. We'll need to do some cleanup first, so add this code at the end of the method:

```
//get the text from the web page
NSString* text = [NSString stringWithContentsOfURL:
    [NSURL URLWithString: url] encoding:NSUTF8StringEncoding
    error:NULL];

//the list of regexes to cleanup the html content
NSArray* cleanup = @[
    @"\\A.*?<body.*?>", //1
    @"</body>.*?\\Z", //2
    @"<[^>]+>", //3
    @"\W+$"]; //4
```

First you fetch the HTML content of the supplied URL into the `text` object. Next you define an array with some pretty awkward strings. These are the regular

expressions you will run on the text to clean it and get only the plain text out of the web page. I just put them quickly together, so they might not be perfect, but for the purpose at hand they should be enough. Just to quickly go through what each regex does:

1. Will match anything from the beginning of the text (\A) to the body tag including.
2. Will match anything from the closing body tag to the end of the text (\Z).
3. Will match all tags (i.e. all occurrences of < and > and everything in between).
4. Cleans all whitespace at the end of the text (\W+).

What you are going to do is replace the matches of those regexes with empty strings - i.e. going to delete everything which is not text content. Add this to the method:

```
//1 run the regexes, get out pure text
for (NSString* regexString in cleanup) {
    NSRegularExpression *regex = [NSRegularExpression
        regularExpressionWithPattern:regexString
        options:NSRegularExpressionDotMatchesLineSeparators
        error:NULL];

    text = [regex stringByReplacingMatchesInString:text
        options:NSRegularExpressionDotMatchesLineSeparators
        range:NSMakeRange(0, [text length])
        withTemplate:@""];
}

//2 add an artificial end of the text
text = [text stringByAppendingString:@"\nSTOP."];

//3 put the text into the tagger
[tagger setString: text];
```

Again let's go through the code step by step:

1. You create the regexes one by one, and then replace the matches with @"" (no text, empty string.)
2. You add a sentence "STOP." at the end of the text - you are going to use that control sentence to detect when you have reached the end of the text.
3. Finally - we feed the plain text to the tagger.

So to put it visually (as at that point is still early to test) let's have a look at a classical before and after example:

<pre> <div class="post-image-inner right"> </div> <p>For the ones who haven't heard about Unity3D which runs on absolute coolness; it's flexible, easy to pick up, binaries for iOS, PC, Mac, Wii and more. If you are an indie game with its super-low price tag of 40\$ for the iOS enabled version!</p> <h2>trial team for few months now and I proudly present my latest in Unity3D; series. I wrote 2 tutorials for absolute Unity</h2> </pre>	<p>heard about Unity3D ; this is a 3D software powerful editor / 3D engine, which creates game build the software for you with its super-low price tag seriesI've been a member of Ray's iOS to make a 2.5D game with Unity3D; series. I wrote layout and basic interaction setting up a scene in audio effects adding particle systems creating 2D and go through the tutorials you don't need to learn trial of Unity to play around with it! I wish you like 2.5D game with Unity3D; Part 2 Marin http://tutorials and excerpts about iPhone and iPad programming-games-with-unity3d/Tutorial: Building iPhone on del.icio.us Share this on Facebook Digg this! Share developer and publisher. He's got more than 10 years Marin's homepage &nbsp; Contact &nbsp;</p>
Before	After

It's not perfect plain text, but is good enough.

Let's make a little detour for a second. To count the nouns in the text you will need a class, which can hold a string (the word) and the number of occurrences you've found so far. Let's create this class right now, before you start tagging around.

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template, set the class name to **WordCount**, and make sure it's a subclass of **NSObject**.

Then replace **WordCount.h** with the following:

```

#import <Foundation/Foundation.h>

@interface WordCount : NSObject

@property (strong) NSString* word;
@property int count;

+(WordCount*)wordWithString:(NSString*)str;

@end

```

As you see it's a pretty humble class - two properties to hold the word and the count of occurrences, plus a class method to create a new instance easily. Next switch to **WordCount.m** and add the implementation:

```

#import "WordCount.h"

@implementation WordCount

+(WordCount*)wordWithString:(NSString*)str
{
    WordCount* word = [[WordCount alloc] init];
    word.word = str;
    word.count = 1;
}

```

```

    return word;
}

@end

```

Pretty standard code, right? But let's get to something more interesting. You will be storing instances of `WordCount` in an array, and every time the tagger finds a word it'll have to check whether the object already exists in the array or not - so you will need an `isEqual:` method to our `WordCount` class.

Also in the end we will want only the most popular words, so you will have to order the array of found words, thus you will need also a sorting helper method for the `WordCount` class. Add them now to the implementation:

```

//method to compare instances
- (NSComparisonResult)compare:(WordCount *)otherObject {
    return otherObject.count-self.count;
}

//method to check for equal instance values
- (BOOL)isEqual:(id)otherObject
{
    return [self.word compare:((WordCount*)otherObject).word]
        ==NSOrderedSame;
}

```

So, there's a bit of trickery here. When `WordCount` objects need to be sorted, you'll be using the `compare:` method, which compares their count of occurrences. For the `isEqual:` method on the other hand, which will be invoked automatically from `[NSArray containsObject:]` you compare the words stored in the class. So in fact two objects holding the same word, but with different count of occurrences will be considered the same object. Pretty cool!

You are now finished with `WordCount` class, so go back to **TagWorker.h** and add at the top:

```
#import "WordCount.h"
```

We can now go forward with invoking the linguistic parser. Bring up **TagWorker.m** and add the following code to the end of the method:

```

//get the tags out of the text
[tagger enumerateTagsInRange: NSMakeRange(0, [text length])
    scheme: NSLinguisticTagSchemeLexicalClass
    options: NSLinguisticTaggerOmitPunctuation |
        NSLinguisticTaggerOmitWhitespace |

```

```

NSLinguisticTaggerOmitOther |
NSLinguisticTaggerJoinNames
usingBlock:^(NSString *tag, NSRange tokenRange, NSRange
sentenceRange, BOOL *stop)
{
    //process tags

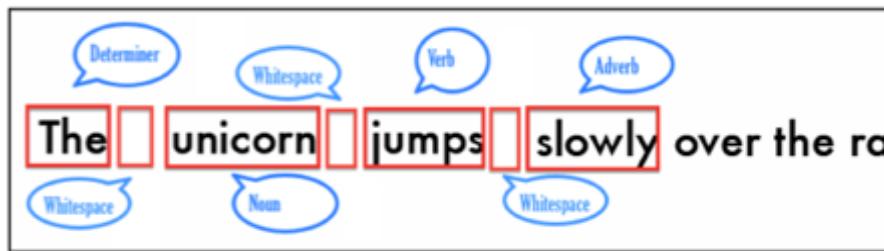
    //check if it's the last sentence in the text

}];

```

This is the main method of `NSLinguisticTagger` - `enumerateTagsInRange:scheme:options:usingBlock:`. All you need to is to specify what range of the text you want to look into, the scheme and options (we discussed before) and the block to invoke for every tag found.

Now I hope by this time you figured out the tags we are talking about here have nothing to do with HTML or XML or other such markup tags. Tags in this case are - items, foundlings, and parts - any way you want to put it. So, again just forget markup language tags for the moment in here - what the tagger does is to enumerate over the parts of the text and tag them like so:



Let's have deeper look at what parameters you set for the tagging process. The scheme is `NSLinguisticTagSchemeLexicalClass`, which will give you back parts of speech: verbs, nouns, adverbs, adjectives, etc. As "options" you pass several values, which mean:

- **NSLinguisticTaggerOmitPunctuation**: don't invoke the block for punctuation tags.
- **NSLinguisticTaggerOmitWhitespace**: don't invoke the block for whitespace of all sorts.
- **NSLinguisticTaggerOmitOther**: don't invoke the block for all kinds of symbols, and other such parts, which are not recognizable.
- **NSLinguisticTaggerJoinNames**: multi word names will be joined and processed as a single tag.

All these options make a lot of sense for what we are trying to do right? Then let's go onto the block part and see what you are going to do with the tags found. Replace the "`//process the tags`" comment with the code:

```
//check for nouns only
if (tag == NSLinguisticTagNoun) {
    WordCount* word = [WordCount wordWithString:
        [text substringWithRange: tokenRange]
    ];

    int index = [words indexOfObject: word];
    if (index != NSNotFound) {
        //existing word - just increase count
        ((WordCount*)words[index]).count++;
    } else {
        //new word, add to the list
        [words addObject: word];
    }
}
```

The tag parameter to the block is what kind of tag according to the current scheme was found. The `NSLinguisticTagSchemeLexicalClass` scheme finds tags such as `NSLinguisticTagNoun`, `NSLinguisticTagVerb`, `NSLinguisticTagAdjective`, `NSLinguisticTagAdverb`, and so forth. You are interested in `NSLinguisticTagNoun` only.

Next you create a `WordCount` instance with the noun found. You get the index of the `WordCount` for that word inside the `words` list.

So if the index that `indexOfObject:` returns is not `NSNotFound` you cast the element inside the array to a `WordCount` instance and increase the `count` property. Otherwise this is a new word, and you can just add it to the word list.

Phew! Lots of coding. There's just one last final step of the tagging and you will be done completely with the `TagWorker` class.

Replace "**//check if it's the last sentence in the text**" with this code that'll do the task:

```
//check if it's the last sentence in the text
if (text.length==sentenceRange.location+sentenceRange.length) {
    *stop = YES;
    [words sortUsingSelector:@selector(compare:)];
    NSRange resultRange = NSMakeRange(0,(number < words.count)?
        number:words.count );
    block( [words subarrayWithRange: resultRange] );
}
```

Since you are sure that our artificial end sentence is the last one, we are just going to check whether the `sentenceRange` passed to the block is at the end of the text. If so, we set `*STOP` to `YES`, so we instruct the tagger we don't need it to continue working.

Next you just call `sortUsingSelector:` to sort the words array. It'll sort itself in decreasing order (you need the most mentioned words).

Next you create a range that will get you number words out of the resulting array (number is a parameter to the current method). If there's less words found than the value of number, then you make a range for just as many as there are found.

Finally you invoke the completion block and you pass over a sub-array of the words array - containing just the number of words asked for.

Wow! That concludes the `TagWorker` class. Very soon you will be able to see some results.

Bringing it All Together

Now that you have your tagger doing wonders let's also integrate it with the rest of the project. Open up **TTMasterViewController.m** and add these two imports:

```
#import "TagWorker.h"
#import "WordCount.h"
```

Now, here's all the code you need - just inside `viewDidLoad` right before **reloading** the table:

```
for (ArticleData* data in _objects) {
    TagWorker* worker = [[TagWorker alloc] init];
    [worker get:5 ofRealTopicsAtURL: data.link
        completion:^(NSArray *words) {
            data.topic = [words componentsJoinedByString:@" "];
            //show the real topics
        }];
}
```

You loop over the fetched RSS articles and for each you create a `TagWorker` and tell it to get the 5 most mention words from the article's web page. Upon completion you just join together the returned words and store them in the topic property in the table data source.

One detail though, now `data.topic` contains something along the lines of this:

```
<WordCount: 0x68cdf80> <WordCount: 0x68cc1a0> <WordCount: 0x68cc4e0>
```

Time for another little trick - `componentsJoinedByString:` actually calls the `description` method on each object, so you are going to override `description` for the `WordCount` class and make it return the stored word. Open up again **WordCount.m** and add this method:

```
- (NSString*)description
{
    return self.word;
}
```

Cool. The only thing left is to show the real topics on the screen. Switch back to **TTMasterViewController.m** and replace this comment "**//show the real topics**" with the code to do the work:

```
dispatch_async(dispatch_get_main_queue(), ^{
    [self.tableView reloadData];
});
```

Now it's time to hit Run and enjoy as "True Topic" reveals the true keywords behind each post:



Hmmm ... alright, I admit that the articles on raywenderlich.com have pretty honest titles ... but I can think of bunch of sites where that's not true - pretty often the title says "Have a free kitten", but inside they only try to sell you their new megazargoyan-superfolio-gamma-ray-absorbant brain protecting shield hats:



A photo by Tom Kirk

Where To Go From Here?

Using the different schemes provided by the `NSLinguisticTagger` class you get not only the parts of the text, but also meta information about them. And you can then draw conclusions about the text, about the author's style, the topic and most importantly the content! So even if you're not able to precisely understand the meaning of the text you can still make educated guesses about it.

If you want to play around with this some more, you can set up the details view controller of the application. Pass the link to the article and open it in a `UIWebView` any way you like.

There is also much more you can do with the linguistic tagger. Keep in mind that the `TagWorker` class from this tutorial is very universal, so you can tweak it to your wishing and reuse as needed. Here are some ideas of where you can take things from here:

- Have a look at all supported schemes and see if you can benefit from using them in your own apps
- You can run a separate tagger on each found tag to get the stem of the word (using the `NSLinguisticTagsSchemeLemma` scheme) - it'll turn "tutorials" to "tutorial", so you won't have both as separate words
- How about creating a text based quest like the ones back on Apple][?

31

Chapter 31: Conclusion

Wow – if you've made it through the entire book, huge congratulations –you really know iOS 5!

You now have experience with all of the major new APIs that were introduced in iOS 5 and should be ready to start using these cool new technologies in your own apps. I'm sure you're already brimming with cool new ideas and possibilities, so we can't wait to see what you've come up with!

If you have any questions or comments as you go along, please stop by our book forums at <http://www.raywenderlich.com/forums>.

Thank you again for purchasing this book. Your continued support is what makes everything we do on raywenderlich.com possible, so we truly appreciate it!

Best of luck with your iOS adventures,

- Steve, Adam, Jacob, Matthijs, Felipe, Cesare, Marin and Ray from the Tutorial Team