

iOS 5 by tutorials

By the raywenderlich.com iOS Tutorial Team

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

iOS 5 By Tutorials

By the raywenderlich.com iOS Tutorial Team

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

Copyright © 2011 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.

Contents

Chapter 1: Introduction	6
Chapter 2: Beginning ARC	13
Chapter 3: Intermediate ARC	73
Chapter 4: Beginning Storyboards	115
Chapter 5: Intermediate Storyboards	179
Chapter 6: Beginning iCloud	221
Chapter 7: Intermediate iCloud	253
Chapter 8: Beginning OpenGL ES 2.0 with GLKit	363
Chapter 9: Intermediate OpenGL ES 2.0 with GLKit	394
Chapter 10: Beginning UIKit Customization	419
Chapter 11: Intermediate UIKit Customization	434
Chapter 12: Beginning Twitter	458
Chapter 13: Intermediate Twitter	472
Chapter 14: Beginning Newsstand	531
Chapter 15: Intermediate Newsstand	541
Chapter 16: Beginning UIPageViewController	561
Chapter 17: Intermediate UIPageViewController	578
Chapter 18: Beginning Turn Based Gaming	613
Chapter 19: Intermediate Turn Based Gaming	662
Chapter 20: Beginning Core Image	688
Chapter 21: Intermediate Core Image	705
Chapter 22: UIViewController Containment	729
Chapter 23: Working with JSON in iOS 5	751
Chapter 24: UIKit Particle Systems	764
Chapter 25: Using the iOS Dictionary	776

Chapter 26: New AddressBook APIs	789
Chapter 27: New Location APIs	807
Chapter 28: New Game Center APIs	825
Chapter 29: New Calendar APIs	839
Chapter 30: Using the New Linguistic Tagger API	866
Chapter 31: Conclusion	885

Dedications

To Kristine, Emily, and Sam.

- Steve Baranski

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

Introduction

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

But as we were researching iOS 5, we realized there isn'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 iOS 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 for 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 has in store for you, you're in the right place. Sit back, relax, and prepare for some fun and informative tutorials!

Who This Book Is For

This book is for intermediate or advanced iOS developers, who already know the basics of iOS development but want to upgrade their skills to iOS 5.



If you're a complete beginner, you can follow along 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 this 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 has a ton of killer new APIs that you'll want to start using in your apps right away. Here's what we'll be covering 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 do we cover how to use ARC, but also how it works, 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 past, you would generally have one XIB for each 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. Well now with iOS 5, you can use the new Storyboarding feature to design the visual look of your app in one place, and easily visualize and manage the transitions between different view controllers. What's more, Storyboards can save you a lot of time, because they introduce cool new features such as easily creating table view cells directly within the editor. In this book, we'll dive deep into Storyboards and show you how you can use all the major 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 a third party API like Dropbox. 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 with 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 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 a lot easier. In this book, we'll dive into the new GLKView, GLKViewController, GLKBaseEffect, GLKTextureLoader, and GLKMath APIs - in a manner easy to follow whether you're 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 past you had to resort to strange workarounds to make this work. With iOS 5, this has become a lot easier, so in this book we're going to dig in to some practical examples of customizing just about every control you might want in UIKit!

Beginning and Intermediate Twitter Integration

iOS 5 makes it a lot easier to use Twitter from within your app, with built-in Twitter integration. In these chapters, we will show you how to use these new APIs from your apps to send Tweets easier than ever!



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, we'll discuss the benefits of distributing your content via Newsstand, describe how to modify your app to appear in Newsstand, and cover 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 uses the GPU to run the filters, so they are extremely fast! In these chapters, we'll show you how to use the Core Image framework to easily apply cool filters to your images. We'll also show you how you can 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. It makes it extremely easy to create multiplayer turn-based games where you take a turn, and then asynchronously wait for a friend to take his turn, and get notifications when it's your turn again. In these chapters, we'll dive into an example of how you can integrate this into a simple game, and then dig further to show how you can create a 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! In these chapters, we'll dive into how this new view controller works, and 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. In these bonus chapters, we dig into almost every new iOS 5 API not already covered - things like View Controller Containment, new JSON, Address Book, and Location APIs, and much more! Even though these APIs contain smaller advancements for iOS we've worked hard to present them in real life projects, which you could benefit directly 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

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

Since you bought the book, you also have access to a special book forum at raywenderlich.com/forums. We will be posting updates to the book there, so be sure to register and check from time to time. Also, that is a great place to ask any questions you have about the book or iOS 5 in general, or submit any errata you may find.

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 this book!
- **Vicki Wenderlich:** For designing the book cover, and much of the lovely artwork and illustrations in the book.
- **Mike Daley:** For helping out in our investigations of GLKit.
- **Sean Berry, Adam Eberbach, 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:



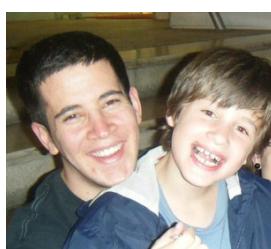
Steve Baranski is the founder of komorka technology, a creator of original & commissioned mobile apps based in Oklahoma City. When not making apps or spending time with his family, he can usually be found on a bicycle of some sort.



Jake Gundersen is a gamer, maker, and developer. He is a Co-founder of Third Rail Games, which you can find at <http://third-railgames.com>.



Matthijs Hollemans is an independent iOS developer and designer who lives to create awesome software. His focus is on iPad apps because he thinks tablets are way cooler than phones. Whenever he is not glued to his screen, Matthijs plays the piano, is into healthy and natural living, and goes out barefoot running. Visit his website at <http://www.hollance.com>



Felipe Laso Marsetti is an iOS programmer and aspiring game developer just doing what he loves. Felipe loves reading and working on further developing his programming and iOS related skills. You can follow Felipe on Twitter as [@Airjordan12345](#) or visit his company website at <http://www.bananasplitgames.com>.



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 snowboard and beach tennis.



Marin Todorov is a software developer with experience on various platforms and languages. He has published couple of books and writes regularly about iOS development on his blog <http://www.touch-code-magazine.com>. At present he is an independent iOS developer and publisher. He loves to read, listen and produce music and learn new (human) languages.



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

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

How It Works

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

- If you need 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

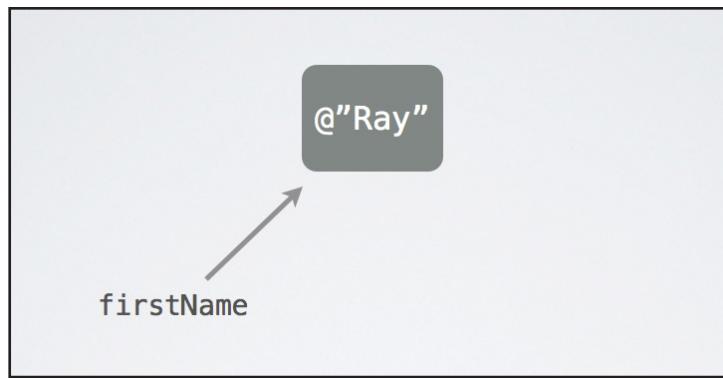
The new rules you have to learn for ARC are quite simple. With manual memory management you needed to retain an object to keep it alive. That is no longer necessary, all you have to do is make a pointer to the object. As long as there is a variable pointing to an object, that object stays in memory. 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,

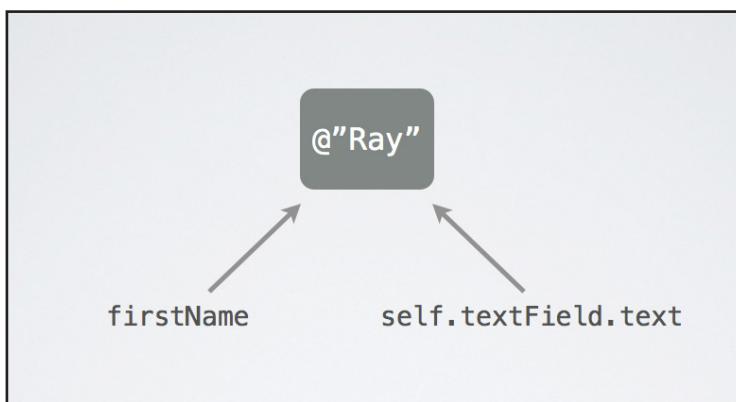
```
NSString *firstName = self.textField.text;
```

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.

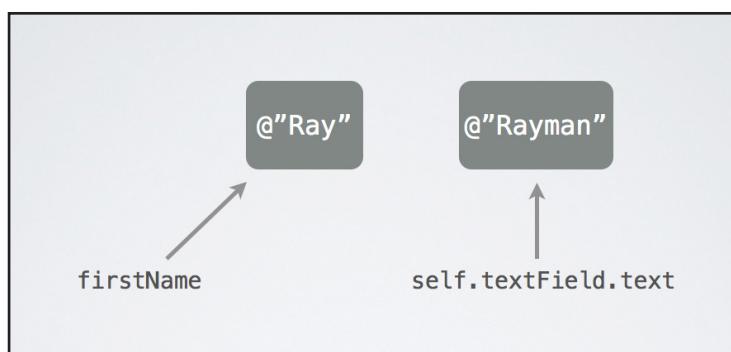




An object can have more than one owner. Until the user changes the contents of the UITextField, its text property is also an owner of the string object. There are two pointers keeping that same string object alive:



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.

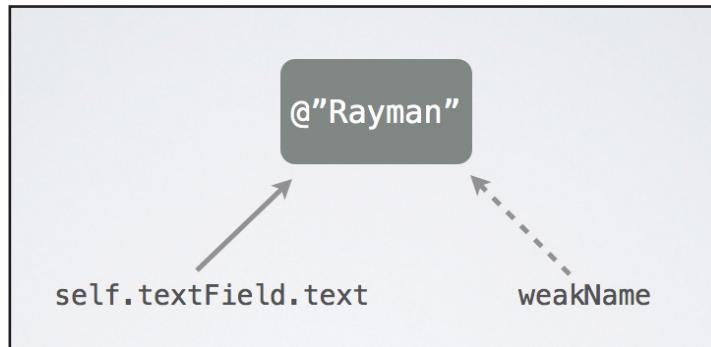




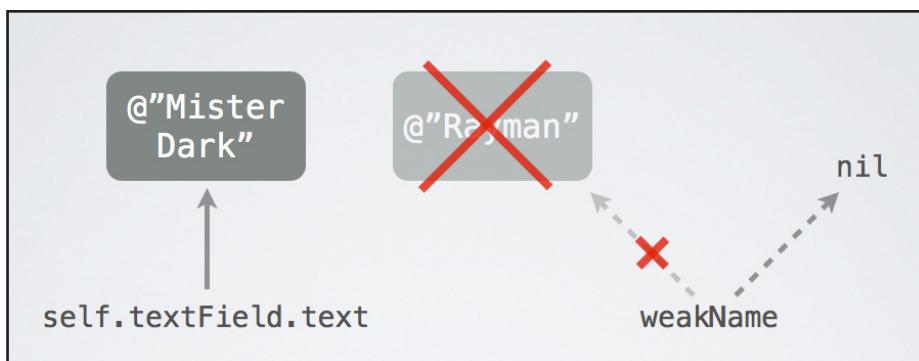
We call pointers such as `firstName` and `textField.text` "strong" because they keep objects alive. By default all instance variables and local variables are strong pointers.

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;
```



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:

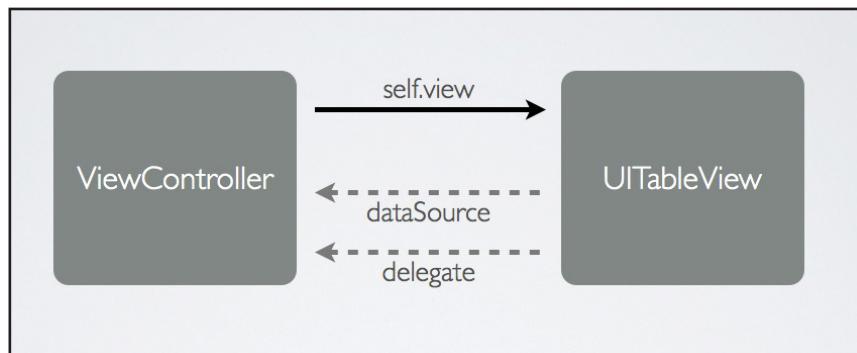


When this happens, the value of weakName automatically becomes nil. It is said to be a "zeroing" weak pointer.

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!

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.

An 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. We'll talk more about this later.



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 string object (because str is weak) and the object will be deallocated immediately after it is created. Xcode will give a warning when you 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. On ARC the above code works as intended. Because we put the object into the obj variable, which is a strong pointer, the array is no longer the only owner of the object. Even if we remove the object from the array, the object is still alive because obj keeps pointing at it.

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. We'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.

But because we're 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 I'll show you several ways how.



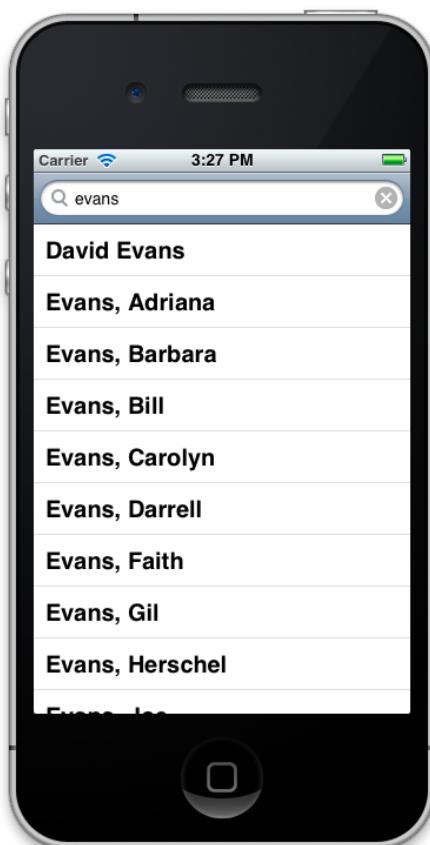
ARC even combines well with C++. With a few restrictions you can also use ARC on iOS 4, which should only help to speed up the adoption.

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 me, switching is a no-brainer.

The App

To illustrate how to use Automatic Reference Counting in practice, I have prepared a simple app that we are going to convert 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:



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 project contains the following source files:

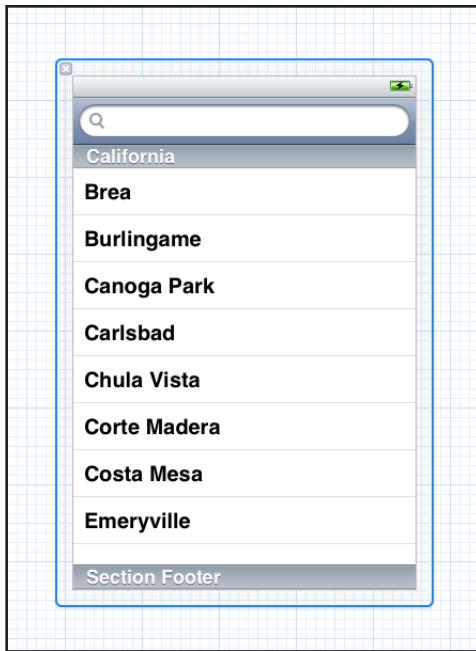
- **AppDelegate.h/.m:** The application delegate. Nothing special here, every app has one. It loads the view controller and puts it into the window.
- **MainViewController.h/.m/.xib:** The view controller for the app. It has a table view and a search bar, and does most of the work.
- **SoundEffect.h/.m:** 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 uses 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. I did not include the full library because we only need this one class. You can find the complete package at: <https://github.com/gowalla/AFNetworking>
- **SVProgressHUD.h/.m/.bundle:** A progress indicator that we will display 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>

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, we fill 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 `UITableViewDataSource` 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, we create a new HUD and show it on top of the table view and search bar, blocking any user input until the network request is done:



Then we create the URL for the HTTP request. We use the MusicBrainz API to search for artists.

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

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

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

We add 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 we construct a User-Agent header that looks like:

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

(I took this formula from another part of the AFNetworking library and put it into the userAgent method in the view controller.)

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 our app -- it's unlikely that a user will be doing that many searches -- so we take no particular precautions for this.

Once we have constructed the NSMutableURLRequest object, we give it to AFHTTPRequestOperation to perform:

```
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 we can add it to an NSOperationQueue (in the queue variable) and it will be handled asynchronously. Because of the HUD, the app ignores any user input while the request is taking place.

We give AFHTTPRequestOperation a block that it invokes when the request completes. Inside the block we first check 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 we 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 we use 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(localizedCa
seInsensitiveCompare:)];
```

You can look up the logic for the XML parsing in the NSXMLParserDelegate methods, but essentially we just look for elements named "sort-name". These contain the names of the artists. We add those names 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

We are going to convert the Artists app to ARC. Basically this means we'll just get rid of all the calls to retain, release, and autorelease, but we'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.

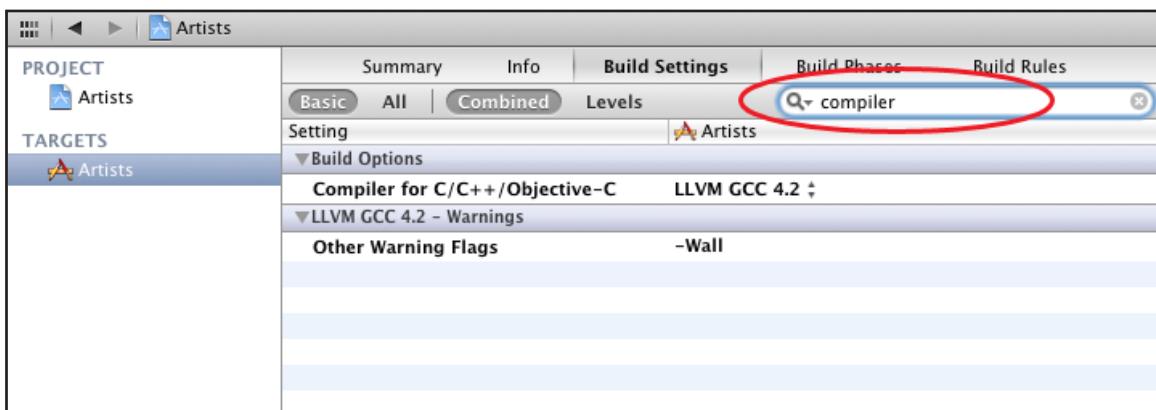


3. You can disable ARC for source files that you do not want to convert. This is useful for third-party libraries that you don't feel like messing with.

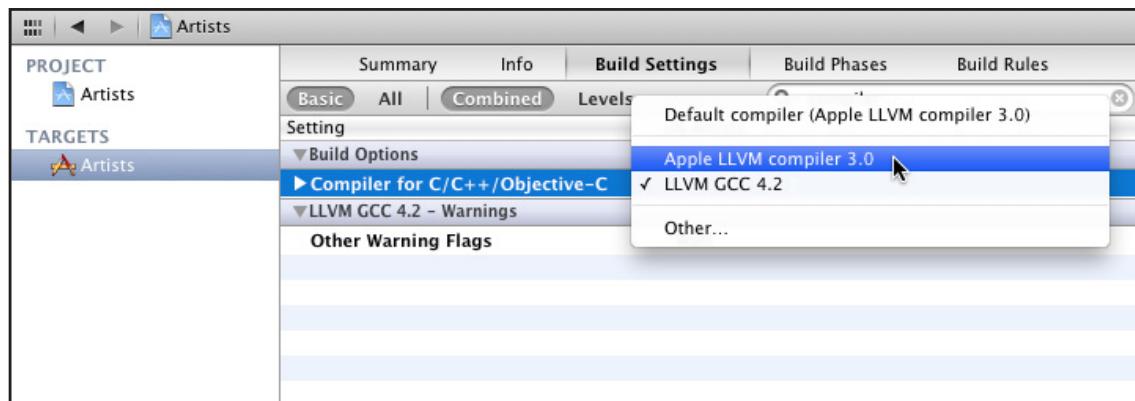
We will use all of these options on the Artists app, just to show you how it all works. In this section, we are going to convert the source files with Xcode's automated conversion tool, except for MainViewController and AFHTTPRequestOperation.

Before we do that, you should make a copy of the project as the tool will overwrite the original files. Xcode does offer to make a snapshot of the source files but just as a precaution I would make a backup anyway.

ARC is a feature of the new LLVM 3.0 compiler. 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:



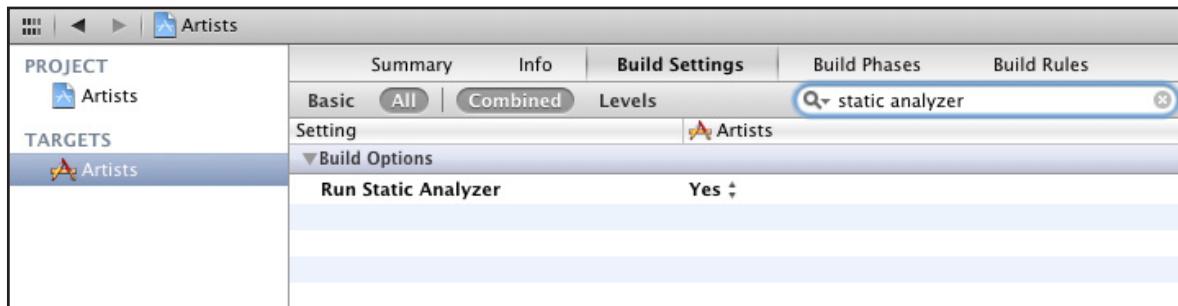
Click on the Compiler for C/C++/Objective-C option to change it to Apple LLVM compiler 3.0:



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 I find it useful to always have all warnings on and to treat them as fatal errors. In other words, if the compiler gives a warning I will first fix it 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 I recommend 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:

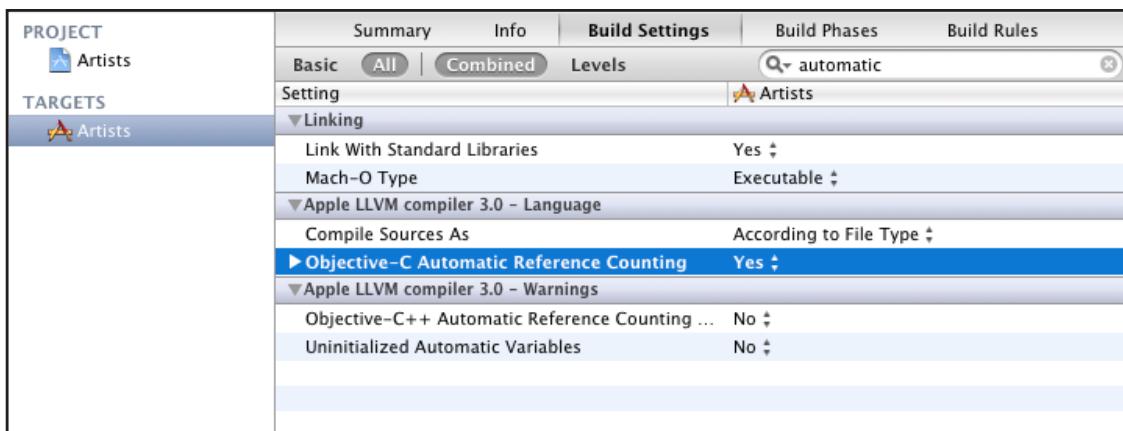


Xcode will now run the analyzer every time we build the app. That makes the builds a bit slower but for an app of this size that's barely noticeable.

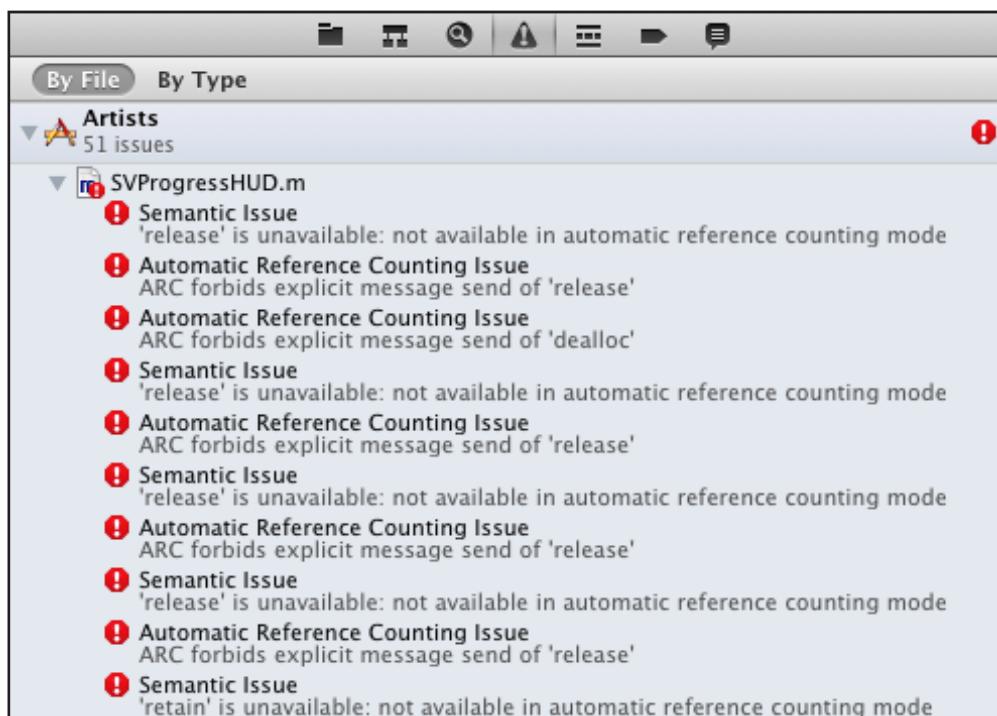
Let's build the app to see if it gives any problems with the new compiler. First do a clean using the Product -> Clean menu option (or Shift-Cmd-K). Then press Cmd-B to build the app. Xcode should give no errors or warnings, so that's cool. If you're converting your own app to ARC and you get any warning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

Just for the fun of it, let's switch the compiler to ARC mode and make it build the app again. We'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.



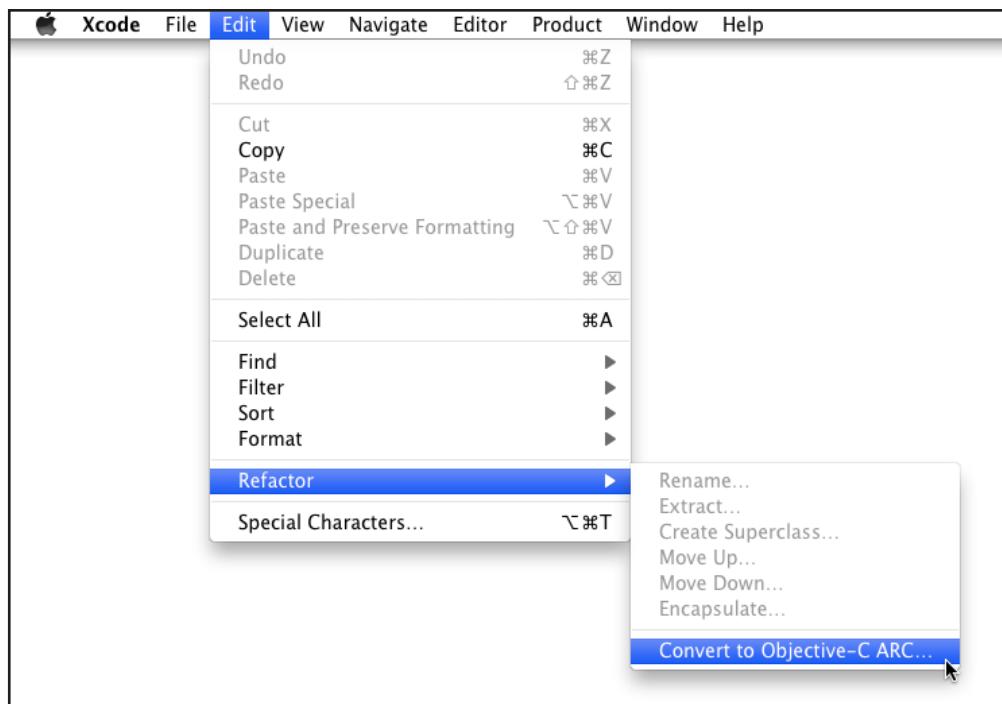
Build the app again. Whoops, you should get a ton of errors:



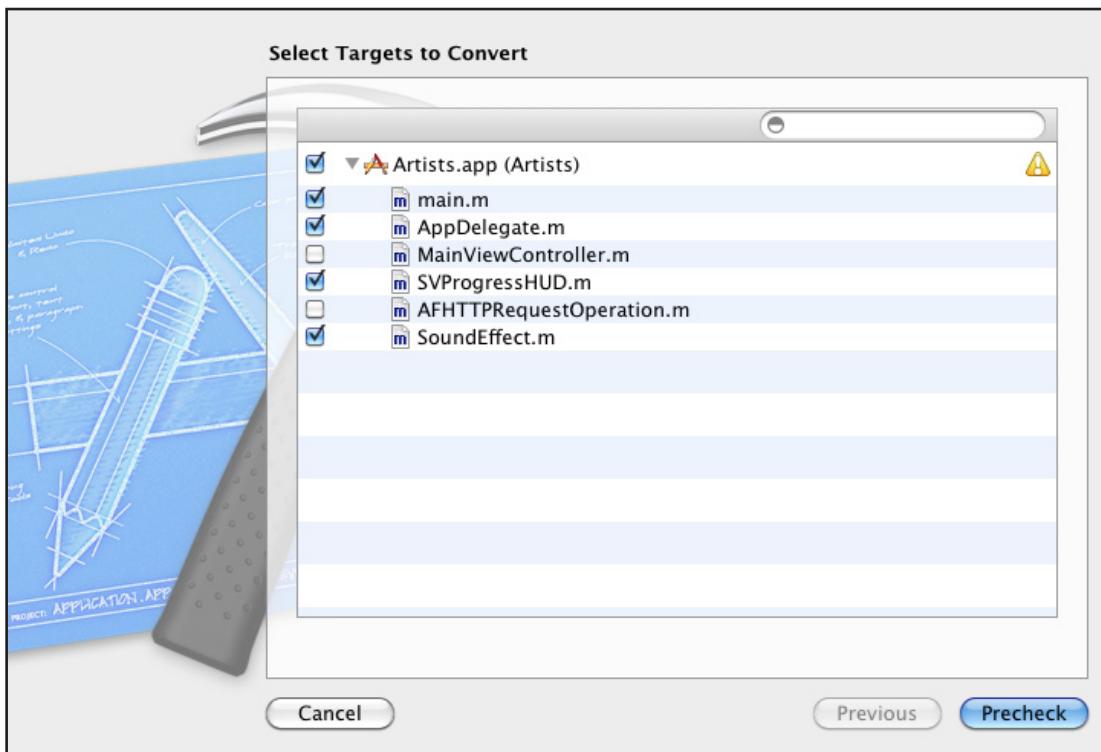
Clearly we have some migrating to do! Most of these errors are pretty obvious, they say you can no longer use retain, release and autorelease. We could fix all of these errors by hand but it's much easier to employ the automatic conversion tool. The tool will compile the app in ARC mode and rewrites the source code for every error it encounters, until it compiles cleanly.

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:



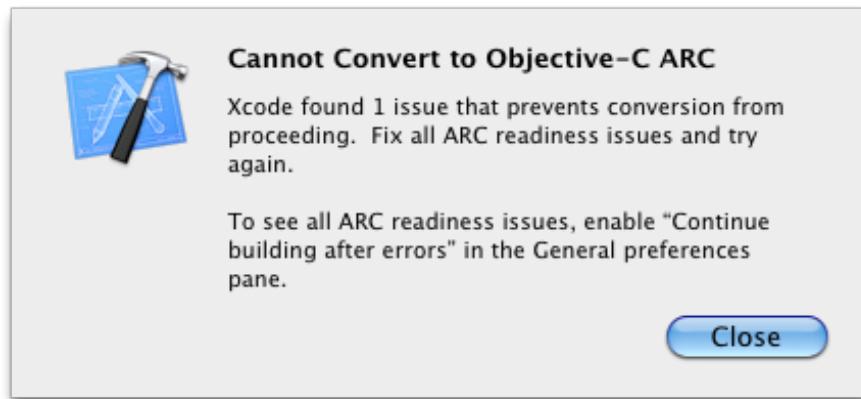
For the purposes of this tutorial, we don't want to do the whole app, so select only the following files:



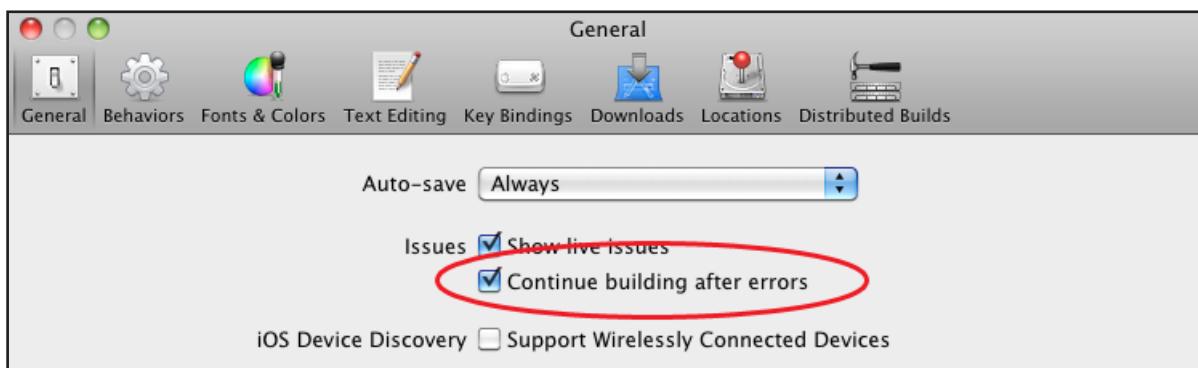
- main.m
- AppDelegate.m
- SVProgressHUD.m
- SoundEffect.m

The dialog shows a little warning icon indicating that the project already uses ARC. That's because we've enabled the Objective-C Automatic Reference Counting option in the Build Settings earlier and now the conversion tool thinks this is already an ARC project. You can ignore the warning, it won't interfere with the conversion.

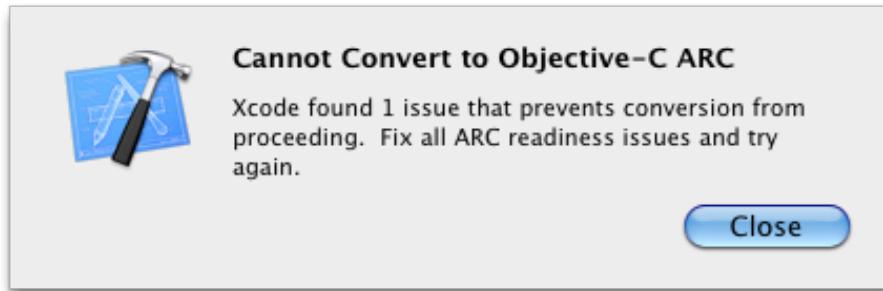
Press the Precheck button to begin. The tool first checks whether your code is in a good enough state to be converted to ARC. We did manage to build our app successfully with the new LLVM 3.0 compiler, but apparently that wasn't good enough. Xcode gives the following error message:



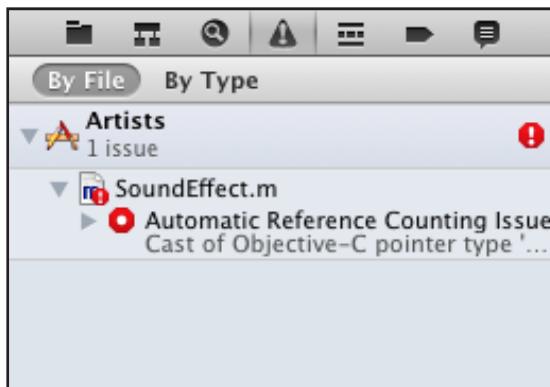
It complains about "ARC readiness issues" and that we should enable the "Continue building after errors" option. We'll do the latter first. Open the **Xcode Preferences** window (from the menubar, under Xcode) and go to the **General** tab. Enable the option **Continue building after errors**:



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



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

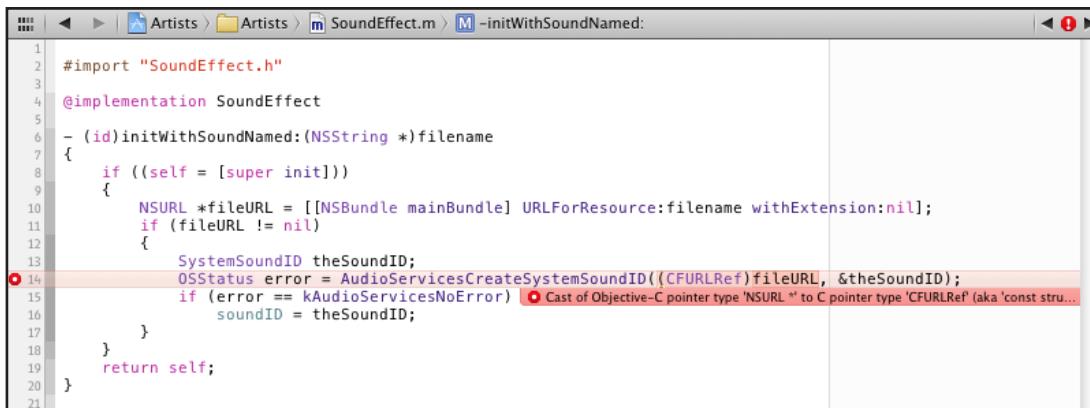


(You may have more errors in this list than are displayed here. Sometimes the conversion tool also complains about things that are not really "ARC readiness" issues.)

The full description of our issue is:

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

What it looks like in the source editor:



The screenshot shows a portion of the Xcode code editor for a file named SoundEffect.m. The code is written in Objective-C. A specific line of code, line 14, is highlighted in red with a circular error icon. The line contains the call to `AudioServicesCreateSystemSoundID`. The error message in the status bar at the bottom right of the editor window reads: "Cast of Objective-C pointer type 'NSURL' to C pointer type 'CFURLRef' (aka 'const struct _CFURLRef *')". This indicates that the compiler is unable to safely convert between the two types.

```

1 #import "SoundEffect.h"
2
3 @implementation SoundEffect
4
5 - (id)initWithSoundNamed:(NSString *)filename
6 {
7     if ((self = [super init]))
8     {
9         NSURL *fileURL = [[NSBundle mainBundle] URLForResource:filename withExtension:nil];
10        if (fileURL != nil)
11        {
12            SystemSoundID theSoundID;
13            OSStatus error = AudioServicesCreateSystemSoundID((CFURLRef)fileURL, &theSoundID);
14            if (error == kAudioServicesNoError) // Cast of Objective-C pointer type 'NSURL' to C pointer type 'CFURLRef' (aka 'const struct _CFURLRef *')
15                soundID = theSoundID;
16        }
17    }
18    return self;
19 }
20
21

```

I will go into more detail about this later, but the source code here 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're 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 will 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: `__bridge`, `__bridge_transfer` and `__bridge_retained`. We will go into more depth on how to use these later in the tutorial.

For now, we need to change the source code to the following:

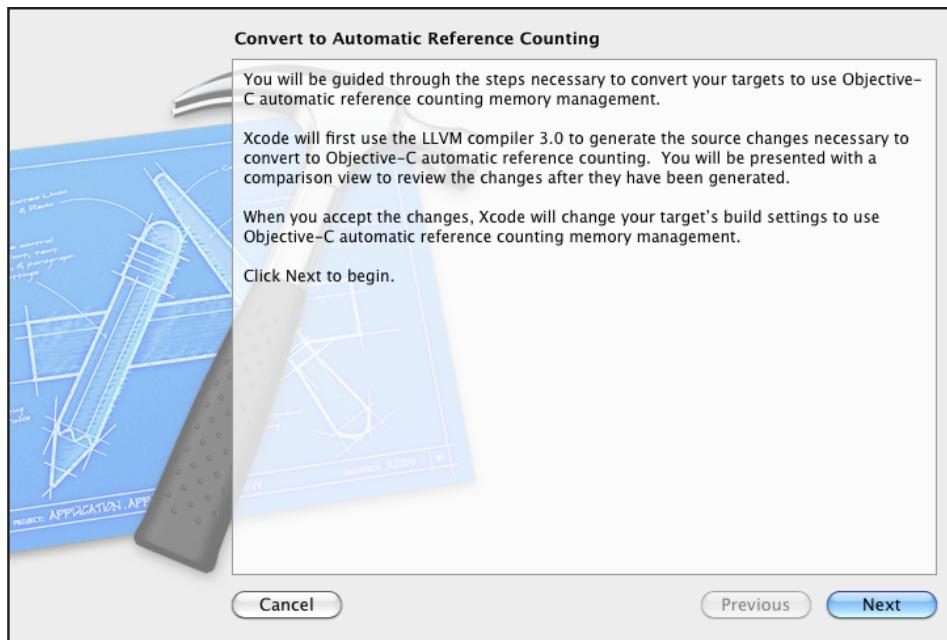
```
OSStatus error = AudioServicesCreateSystemSoundID((__bridge CFURLRef)
    fileURL, &theSoundID);
```

We added the `__bridge` keyword inside the cast to CFURLRef.

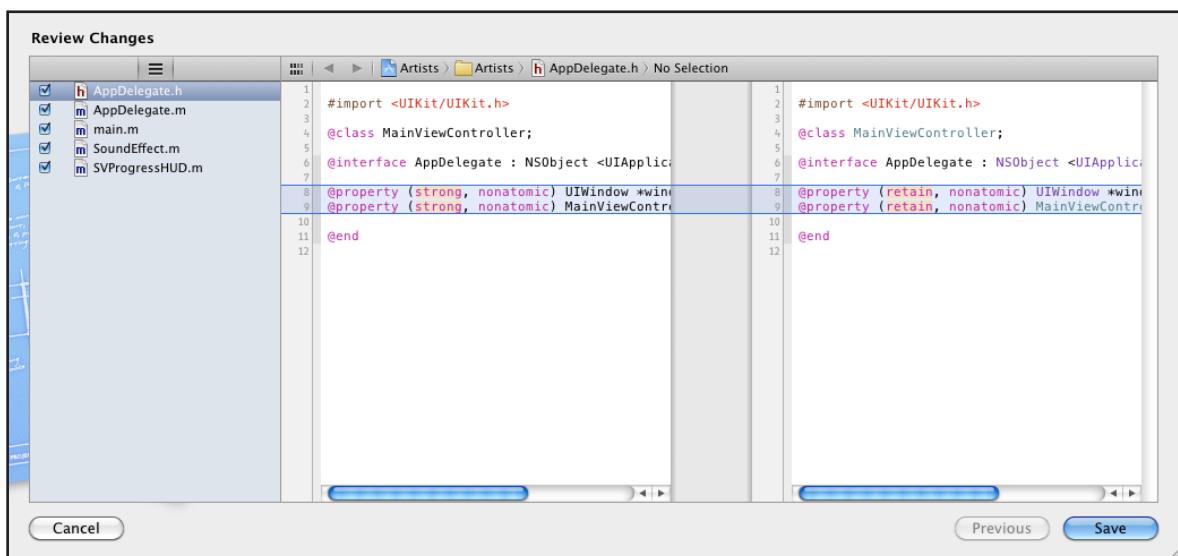
The pre-check may have given you errors other than just this one. You can safely ignore those, the above change in SoundEffect.m is the only one we need to make. The conversion tool seems a little uncertain in what it considers an "ARC readiness issue" from time to time.

Let's run the conversion tool once more - **Edit\Refactor\Convert to Objective-C ARC**. This time the pre-check runs without problems and we'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. 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 change from this,

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

to this:

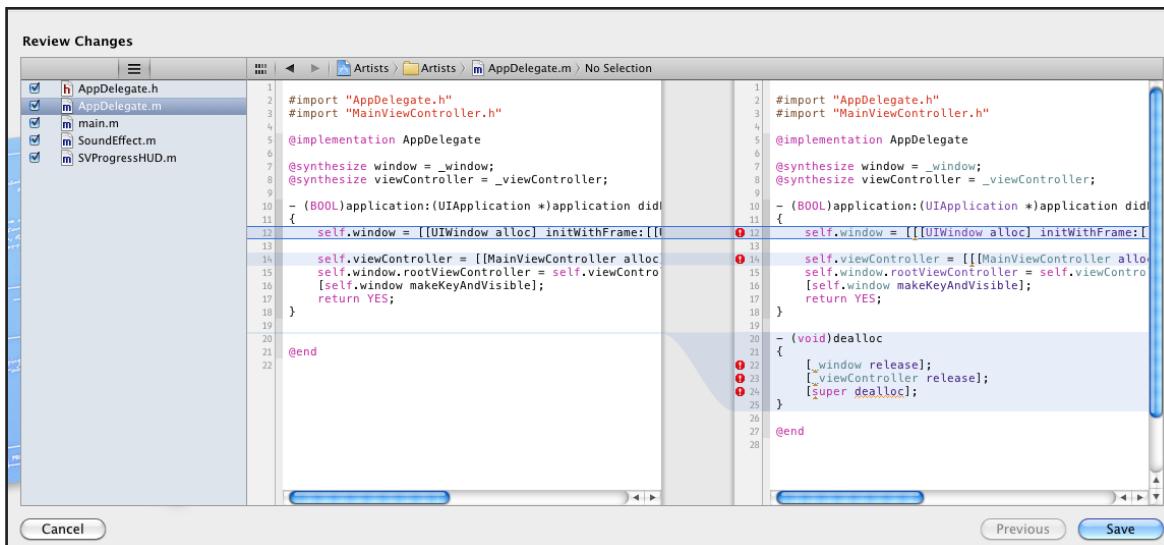
```
@property (strong, nonatomic)
```

The strong keyword means what you think it does. It tells ARC that the synthesized ivar 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:





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 a retained object 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 I love 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 ivars by hand.

Something that the conversion tool doesn't do is making AppDelegate a subclass of UIResponder instead of NS0bject. When you create a new app using one of Xcode's templates, the AppDelegate class is now a subclass of UIResponder. It doesn't seem to do any harm to leave it as NS0bject, but you can make it a UIResponder if you want to:

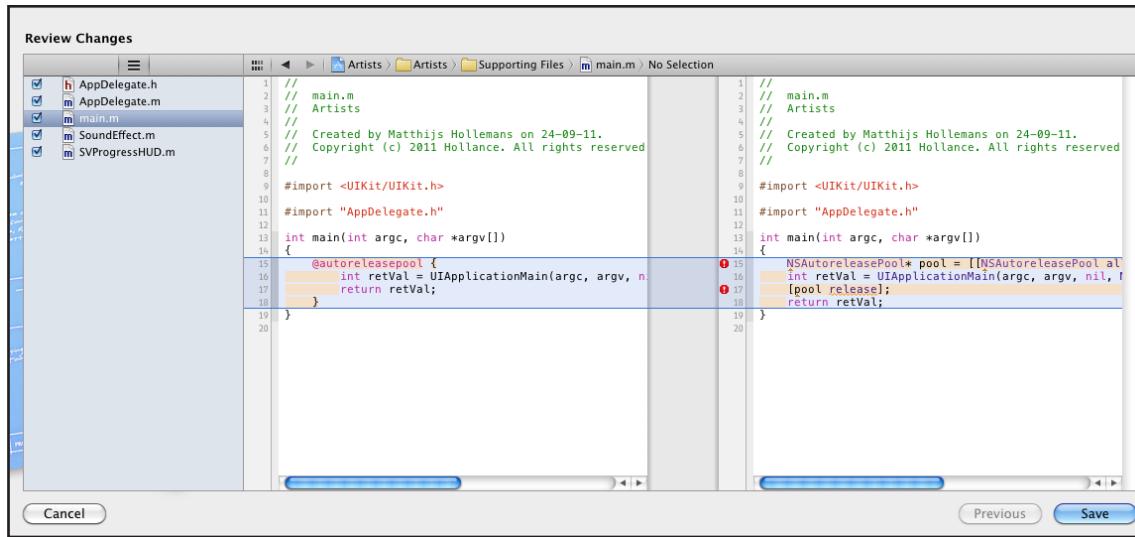
```
@interface AppDelegate : UIResponder <UIApplicationDelegate>
```

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





The conversion tool turned our 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` block. 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

File	Line	Text
AppDelegate.h	3	@implementation SoundEffect
AppDelegate.m	4	- (id)initWithSoundNamed:(NSString *)filename
main.m	5	{
SoundEffect.m	6	if ((self = [super init]))
SVProgressHUD.m	7	{
	8	if (!fileURL ![[NSBundle mainBundle] URLForResourceNamed:filename])
	9	{
	10	NSURL *fileURL = [[NSBundle mainBundle] URLForResourceNamed:filename];
	11	if (fileURL != nil)
	12	{
	13	SystemSoundID theSoundID;
	14	OSStatus error = AudioServicesCreateSystemSoundID(fileURL, &theSoundID);
	15	if (error == kAudioServicesNoError)
	16	soundID = theSoundID;
	17	}
	18	}
	19	return self;
	20	}
	21	- (void)dealloc
	22	{
	23	AudioServicesDisposeSystemSoundID(soundID);
	24	}
	25	- (void)play
	26	{
	27	AudioServicesPlaySystemSound(soundID);
	28	}
	29	@end
	30	
	31	
	32	
	33	
	34	@implementation SoundEffect
	35	- (id)initWithSoundNamed:(NSString *)filename
	36	{
	37	if ((self = [super init]))
	38	{
	39	NSURL *fileURL = [[NSBundle mainBundle] URLForResourceNamed:filename];
	40	if (fileURL != nil)
	41	{
	42	SystemSoundID theSoundID;
	43	OSStatus error = AudioServicesCreateSystemSoundID(fileURL, &theSoundID);
	44	if (error == kAudioServicesNoError)
	45	soundID = theSoundID;
	46	}
	47	}
	48	return self;
	49	}
	50	- (void)dealloc
	51	{
	52	AudioServicesDisposeSystemSoundID(soundID);
	53	[super dealloc];
	54	}
	55	- (void)play
	56	{
	57	AudioServicesPlaySystemSound(soundID);
	58	}
	59	@end
	60	
	61	
	62	
	63	

Cancel Previous Save

Notice that a deallocate method is still necessary here. In most of your classes you can simply forget about deallocate and let the compiler take care of it. Sometimes, however, you will need to release resources manually. That's the case with this class as well. When the SoundEffect object is deallocated, we still need to call `AudioServicesDisposeSystemSoundID()` to clean up the sound object and deallocate 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

File	Line	Left Content	Right Content
AppDelegate.m	4	// Created by Sam Vermette on 27.03.11.	// Created by Sam Vermette on 27.03.11.
AppDelegate.m	5	// Copyright 2011 Sam Vermette. All rights reserved	// Copyright 2011 Sam Vermette. All rights reserved
main.m	6	//	//
SoundEffect.m	7		
SVProgressHUD.m	8	#import "SVProgressHUD.h"	#import "SVProgressHUD.h"
	9	#import <QuartzCore/QuartzCore.h>	#import <QuartzCore/QuartzCore.h>
	10		
	11	@interface SVProgressHUD ()	@interface SVProgressHUD ()
	12		
	13	@property (nonatomic, readonly) SVProgressHUDMaskType	@property (nonatomic, readonly) SVProgressHUDMaskType
	14	@property (nonatomic, strong) NSTimer *fadeOutTimer;	@property (nonatomic, retain) NSTimer *fadeOutTimer;
	15	@property (nonatomic, strong) UILabel *stringLabel;	@property (nonatomic, retain) UILabel *stringLabel;
	16	@property (nonatomic, strong) UIImageView *imageView	@property (nonatomic, retain) UIImageView *imageView
	17	@property (nonatomic, strong) UIActivityIndicatorView	@property (nonatomic, retain) UIActivityIndicatorView
	18	- (void)showInView:(UIView*)view status:(NSString*)s	- (void)showInView:(UIView*)view status:(NSString*)s
	19	- (void)setStatus:(NSString*)string;	- (void)setStatus:(NSString*)string;
	20		
	21	- (void)dismiss;	- (void)dismiss;
	22	- (void)dismissWithStatus:(NSString*)string error:(B	- (void)dismissWithStatus:(NSString*)string error:(B
	23	- (void)dismissWithStatus:(NSString*)string error:(B	- (void)dismissWithStatus:(NSString*)string error:(B
	24	- (void)dismissWithStatus:(NSString*)string error:(B	- (void)dismissWithStatus:(NSString*)string error:(B
	25		
	26	- (void)memoryWarning:(NSNotification*)notification;	- (void)memoryWarning:(NSNotification*)notification;
	27		
	28	@end	@end
	29		
	30		
	31	@implementation SVProgressHUD	@implementation SVProgressHUD
	32		
	33	@synthesize maskType, fadeOutTimer, stringLabel, ima	@synthesize maskType, fadeOutTimer, stringLabel, ima
	34		
	35		

Cancel **Save** **Previous**

At the top of **SVProgressHUD.m** you'll find a so-called "class extension", `@interface SVProgressHUD ()`, 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 use a class extension on someone else's class.)

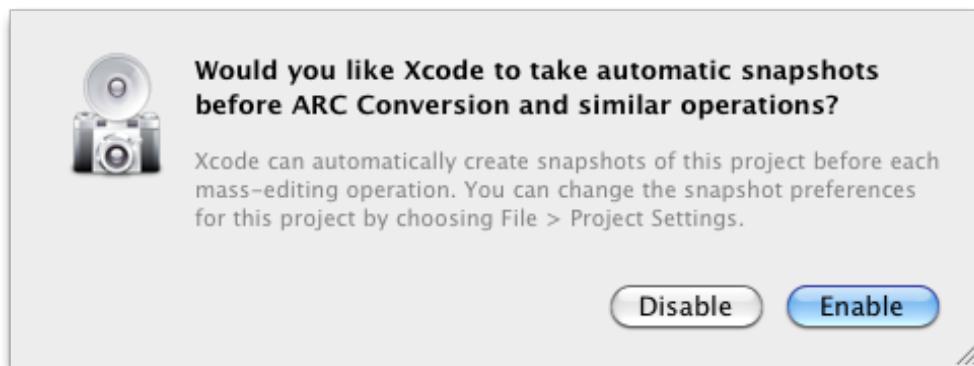
The cool thing about class extensions is that they allow you to add private properties and method names to your classes. If you don't want to expose certain properties or methods in your public `@interface`, then you can put them in a class extension. That's exactly what the author of `SVProgressHUD`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

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

As we'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:

```

124 }
125
126 #pragma mark - Instance Methods
127
128 - (void)dealloc {
129     if(fadeOutTimer != nil)
130         [fadeOutTimer invalidate], [fadeOutTimer release], fadeOutTimer = nil; // Expression result unused
131     [[NSNotificationCenter defaultCenter] removeObserver:self];
132 }
133
134 - (id)initWithFrame:(CGRect)frame {
135     if ((self = [super initWithFrame:frame])) {
136         self.userInteractionEnabled = NO;
137         self.backgroundColor = [UIColor clearColor];
138         self.alpha = 0;
139     }
140 }
141
142
143
144

```

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.

If you're confused by the commas, 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 ivar 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 we don't need to do fadeOutTimer = nil; in dealloc, because

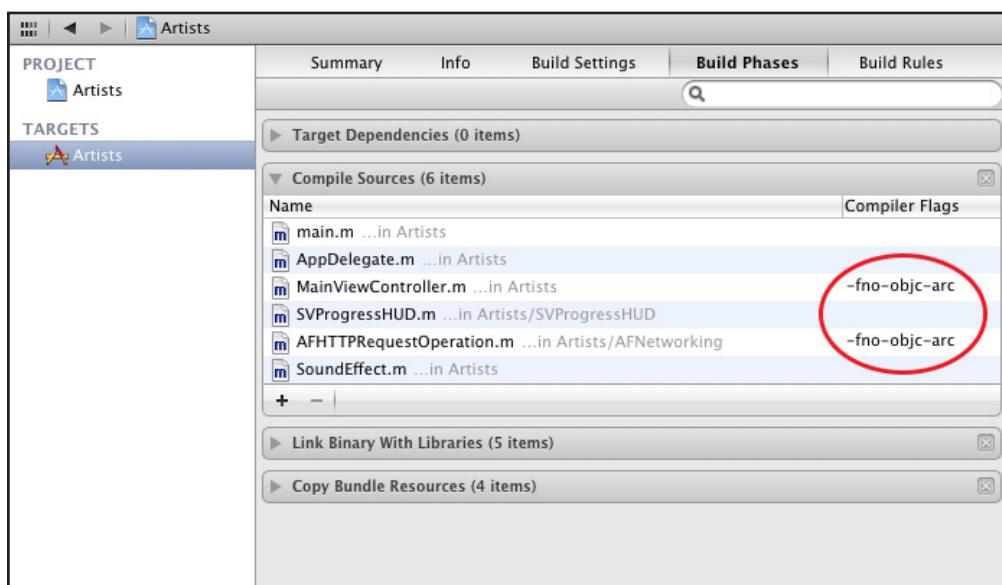


the object will automatically release any instance variables when it gets deleted. In the other methods where the timer is invalidated, however, you definitely should set fadeOutTimer to nil. If you don't, the SVProgressHUD keeps hanging on to the invalidated NSTimer object longer than it's supposed to.

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

But wait a minute... we skipped MainViewController and AFHTTPRequestOperation when we did the conversion. How come they suddenly compile without problems? When we 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:



We enabled ARC on a project-wide basis earlier when we 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 do that is to simply convert 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 could 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

Our conversion went fairly smooth. We only had to make a single change to SoundEffect.m (inserting a `__bridge`) statement and then the tool did the rest.

However, the LLVM 3.0 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 we've seen before. When the compiler can't figure out by itself how to do the 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.

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

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

Now the 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 no longer valid:

```
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 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

We've converted almost the entire project to ARC already, except for `MainViewController` and `AFHTTPRequestOperation`. In this section I'll show you 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 we move such implementation details into the @implementation section of the class. I'm happy to say that this is now possible with LLVM 3.0 (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 ivars where they really belong.

You can do the same for the SoundEffect class. Simply move the instance variable section into the .m file. Because we now don't 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.

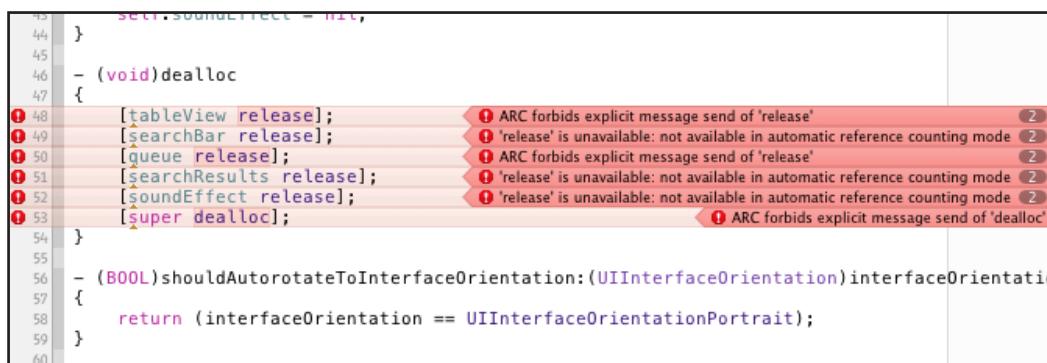


Note: You can also put instance variables in class extensions. This is useful for when the implementation of your class is spread over multiple files. You can then put the extension in a shared, private header file, so that all these different implementation files have access to the instance variables.

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**. You may have some problems getting Xcode to recognize this. Try doing a new build. You should get a ton of errors but if Xcode still says "Build Succeeded", then close the project and reopen it.

Dealloc

Let's go through these errors and fix them one by one. We begin with dealloc:



The screenshot shows a portion of the MainViewController.m code in Xcode. The dealloc method is highlighted. Numerous errors are visible, each pointing to a line that contains a release message send. The errors are as follows:

- Line 48: [tableView release]; (ARC forbids explicit message send of 'release')
- Line 49: [searchBar release]; (ARC forbids explicit message send of 'release')
- Line 50: [queue release]; (ARC forbids explicit message send of 'release')
- Line 51: [searchResults release]; (ARC forbids explicit message send of 'release')
- Line 52: [soundEffect release]; (ARC forbids explicit message send of 'release')
- Line 53: [super dealloc]; (ARC forbids explicit message send of 'dealloc')

Every single line in dealloc gives an error. We're not supposed to call [release] anymore, nor [super dealloc]. Because we're not doing anything else in dealloc, we 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.

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 (something we will get into soon) 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:

```

45
46     - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
47     {
48         return (interfaceOrientation == UIInterfaceOrientationPortrait);
49     }
50
51     - (SoundEffect *)soundEffect
52     {
53         if (soundEffect == nil) // lazy loading
54         {
55             SoundEffect *theSoundEffect = [[SoundEffect alloc] initWithSoundNamed:@"Sound.caf"];
56             self.soundEffect = theSoundEffect;
57             [theSoundEffect release]; // 'release' is unavailable: not available in automatic reference counting mode
58         }
59         return soundEffect;
60     }
61
62     #pragma mark - UITableViewDataSource
63
64     - (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
65     {

```

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. I used a common pattern here 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 I 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];

```

We 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, we 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 Release Me, Let Me Go

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

```

71
72 - (UITableViewCell *)tableView:(UITableView *)theTableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
73 {
74     static NSString *CellIdentifier = @"Cell";
75
76     UITableViewCell *cell = [theTableView dequeueReusableCellWithIdentifier:CellIdentifier];
77     if (cell == nil)
78         cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier] autorelease];
79     if ([self.searchResults count] == 0)
80         cell.textLabel.text = @"(Nothing found)";
81     else
82         cell.textLabel.text = [self.searchResults objectAtIndex:indexPath.row];
83
84     return cell;
85 }
86
87

```

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 I dedicate a special section to.

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

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 (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 my 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 I think it's better 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;
}
```

Of course, this means we can no longer do `self.searchResults` and `self.soundEffect`. Change `viewDidLoad` to the following:

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

We still need to set `soundEffect` to `nil` because we do want to deallocate the `SoundEffect` object here. When the iPhone gets a low-memory warning, we should free as much memory as possible and the `SoundEffect` object is expendable at that point. 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.

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 we 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 we set soundEffect to nil (in viewDidLoad), or until the MainViewController is deallocated.

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 in searchBarSearchButtonClicked, we still do:

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

This will work even though we 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:

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

Don't change it to this, though:

```
[soundEffect play];
```

Because we use lazy loading to load the SoundEffect object, the soundEffect instance variable will always be nil until you call the soundEffect method. Therefore, to be certain we actually have a SoundEffect object, you should always access it through self. If you feel that this pattern does morally require you to declare soundEffect as a @property, then go right ahead. Different strokes for different folks. :-)

As a best practice, if you define something as a property, then you should always use it as a property. The only places where you should access the property's backing instance variable directly are in init 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 ivar:

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

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



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. However, it is better to call your properties strong because that's the proper term from now on. But for these two particular properties I have other plans. Instead of strong, we 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. These view objects are already part of the view controller's view hierarchy and don't need to be retained elsewhere. The big advantage of declaring your outlets weak is that it saves you time writing the `viewDidUnload` method.

Currently our `viewDidUnload` looks like this:

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

You can now simplify it to the following:

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

That's right, because the `tableView` and `searchBar` properties are weak, they are automatically set to nil when the objects they point to are destroyed. That's why we call them "zeroing" weak pointers.

When the iPhone receives a low-memory warning, 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. There is no more need to do this ourselves in `viewDidUnload`. In fact, by the time `viewDidUnload` gets called these properties already are nil. I shall demonstrate this soon, but for that we need to add a second screen to the application.



This doesn't mean you can completely forget about `viewDidUnload`. Remember, as long as you keep a pointer to an object, it stays alive. If you no longer want to hang on to objects, you need to set their pointers to nil. That's exactly what we do for `soundEffect`. We don't want to delete the `searchResults` array at this point but if we did then we would also set that to nil here. Any data that you don't need that isn't a weak outlet property you still need to nil out in `viewDidUnload`! The same goes for `didReceiveMemoryWarning`.

So from now on, make your outlet properties weak. The only outlets that should be strong are the ones from File's Owner that are connected to top-level objects in the nib.

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`.

Before ARC, you were able to write this:

```
@property (nonatomic, readonly) NSString *result;
```

which would implicitly create an assign property. That used to be fine for readonly values. After all, what does retained mean when you talk about read-only data? However, with ARC the above will give the following error:

```
"ARC forbids synthesizing a property of an Objective-C object with unspecified ownership or storage attribute"
```

You must explicitly state whether you want this property to be strong, weak or `unsafe_unretained`. Most of the time, `strong` is the proper answer:

```
@property (nonatomic, strong, readonly) NSString *result;
```

Earlier I mentioned that if you declare a property you should always access it through `self.propertyName` rather than through the backing instance variable (ex-



cept in `init` and in any custom getter or setter methods). This is especially true 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 your .h:

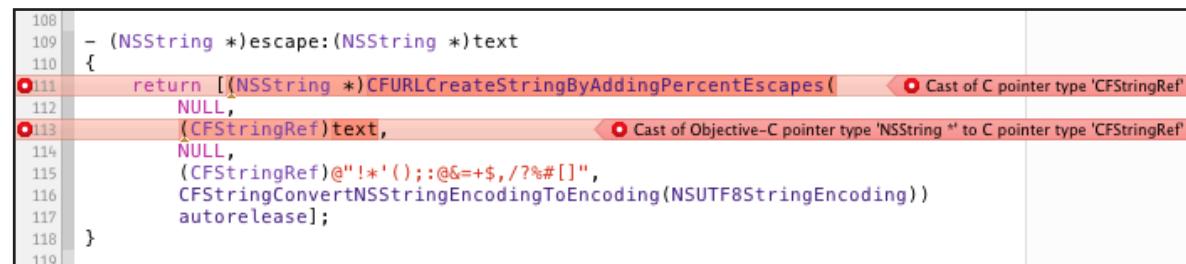
```
@interface WeatherPredictor
@property (nonatomic, strong, readonly) NSNumber *temperature;
@end
```

In your .m:

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

Toll-Free Bridging

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



```
108
109 - (NSString *)escape:(NSString *)text
110 {
111     return [(NSString *)CFURLCreateStringByAddingPercentEscapes(
112         NULL,
113         (CFStringRef)text,
114         NULL,
115         (CFStringRef)@"!*'();:@&+$,/?%#[",
116         CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding))
117         autorelease];
118 }
119 }
```

The screenshot shows a portion of Objective-C code within Xcode. Lines 111 and 113 are highlighted in red, indicating errors. A tooltip for line 111 says "Cast of C pointer type 'CFStringRef'". A tooltip for line 113 says "Cast of Objective-C pointer type 'NSString *' to C pointer type 'CFStringRef'". Line 117 contains the word "autorelease", which is also highlighted in red, indicating another error.

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:

- Cast of C pointer type 'CFStringRef' to Objective-C pointer type 'NSString *' requires a bridged cast
- Cast of Objective-C pointer type 'NSString *' to C pointer type 'CFStringRef' requires a bridged cast
- Semantic Issue: 'autorelease' is unavailable: not available in automatic reference counting mode
- Automatic Reference Counting Issue: ARC forbids explicit message send of



'autorelease'

These last two errors are really the same and simply mean that we cannot do [autorelease]. Let's get rid of that first. What remains is this:

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

The other two errors have to do with casts that apparently should be "bridged". There are three casts in this method:

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

The compiler only complains about the first two. The third one 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.

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 is a regular C string now and therefore doesn't begin with the @ sign. Instead of making an NSString object and casting it to a CFStringRef, we directly make a CFStringRef object. Which one you like better is largely a matter of taste, as they both deliver the exact same results.

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!



For all intents and purposes, `NSString` and `CFStringRef` can be treated as the 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 that was as simple as doing a cast:

```
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.

Let's look at the simple case first. The `CFURLCreateStringByAddingPercentEscapes()` function takes a bunch of parameters, two of which are `CFStringRef` objects. These are the Core Foundation equivalent of `NSString`. Previously we could just cast these `NSString` objects into a `CFStringRef` but with ARC the compiler needs more information. We've already dealt with the constant string, which requires no bridging cast because it is a special object that will never be released. However, the text parameter is another story.

The text variable is an `NSString` object that was given to this method as a parameter. 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.



We'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 we can cast the one into the other. Because we still want ARC to release the object when we're done with it, we 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,  
        CFSTR("!*'();:@&=+$,/?%#[[]]"),  
        CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding));  
}
```

That takes care of all but one of the errors.

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 or relieve ARC of its ownership. In that case there are two other bridging casts that you can use:

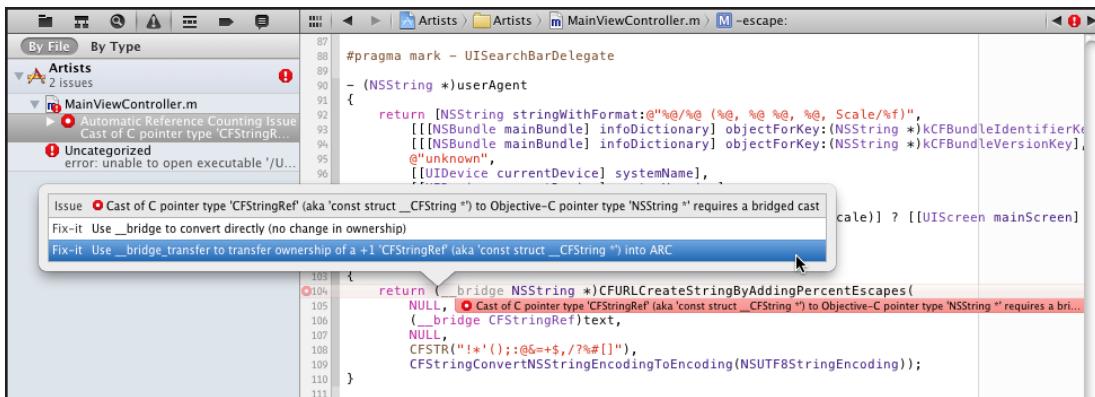
- **`__bridge_transfer`**: Give ARC ownership
- **`__bridge_retained`**: Relieve ARC of its ownership

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

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

If you click on the error message, the following Fix-it will pop up:





It gives two possible solutions: `__bridge` and `__bridge_transfer`. The correct choice here is `__bridge_transfer`. The `CFURLCreateStringByAddingPercentEscapes()` function creates a new `CFStringRef` object. Of course, we'd rather use `NSString` so we need to do a cast. What we're really attempting to do 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. Someone is responsible for releasing that retained object at some point. If we weren't returning this object as an `NSString`, then our code may have looked something like this:

```
- (void)someMethod
{
    CFStringRef result = CFURLCreateStringByAddingPercentEscapes
        (. . .);

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

    CFRelease(result);
}
```

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 we want to do in `escape` is convert that new `CFStringRef` object to an `NSString` object, and then ARC should automatically release that string whenever we're no longer using it. But ARC needs to be told about this. Therefore, we use the `__bridge_transfer` modifier to say: "Hey ARC, this `CFStringRef` object is now an `NSString` object and we want you to dispose of it, so that we don't have to call `CFRelease()` on it ourselves."

The method now becomes:

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

We take the result from `CFURLCreateStringByAddingPercentEscapes()`, which is a `CFStringRef` object, and convert it into an `NSString` that is now under the care of ARC.

If we 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!

To make it a little easier to remember which type of bridge to use, there is a helper function named `CFBridgingRelease()`. It does the exact same thing as a `__bridge_transfer` cast but the meaning is clearer. The final version of the `escape:` method becomes:

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

Instead of doing `(__bridge_transfer NSString *)` we now wrap `CFURLCreateStringByAddingPercentEscapes()` inside a call to `CFBridgingRelease()`. This `CFBridgingRelease()` function is defined as an inline function so it is no slower than doing the cast directly. It is named `CFBridgingRelease()` because you use it anywhere you would otherwise do a `CFRelease()` to balance the creation of the new object.

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

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



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, __bridge_released? 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 __bridge_released 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 = (__bridge_retained CFStringRef)s1;  
  
// do something with s2  
// . . .  
  
CFRelease(s2);
```

As soon as the (`__bridge_retained CFStringRef`) cast happens, ARC considers itself no longer duty-bound to release the string object. 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.

There is also a helper function for this kind of cast: `CFBridgingRetain()`. You can tell by the name that it makes Core Foundation do a retain on the object. The above example is better written as:

```
CFStringRef s2 = CFBridgingRetain(s1);  
  
// . . .  
  
CFRelease(s2);
```

Now the meaning of the code is clearer. The call to `CFRelease()` is properly balanced by `CFBridgingRetain()`. 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'll have a lot of Core Foundation code in your apps. 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, the compiler will 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 `__bridge` 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;
    ...
}
```

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

To summarize:

- When changing ownership from Core Foundation to Objective-C you use `CFBridge-ingRelease()`.
- When changing ownership from Objective-C to Core Foundation you use `CFBridge-ingRetain()`.
- 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. We won't convert `AFHTTPRequestOperation` to ARC in this tutorial.



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 I 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, we'll first have to add a new screen to the app.

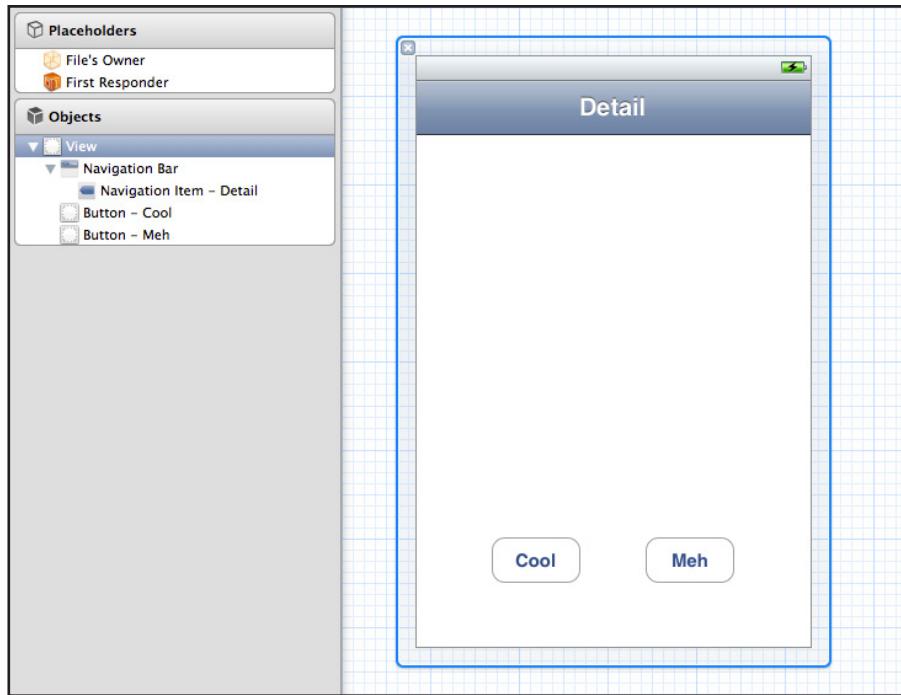
Add a new **UIViewController subclass** to the project, with XIB for user interface, and name it **DetailViewController**.

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

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

We will connect these actions to two buttons in the nib. Open **DetailViewController.xib** and change the design to:





It just has a navigation bar and two buttons. 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 we'll leave these methods empty:

```
- (IBAction)coolAction
{
}

- (IBAction)mehAction
{
}
```

We will make some changes to the main view controller so that it invokes this Detail screen when you tap on a search result. Change the didSelectRowAtIndexPath method in **MainViewController.m** 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.

Then 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 we put a single row into the table that says "(Nothing found)". We don't want to open the Detail screen when the user taps that row.

Because MainViewController doesn't know anything about the DetailViewController class yet, we have to add an #import. Add this to **MainViewController.h**:

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

(We're adding it to the .h file and not the .m file for reasons that will soon become apparent.)

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, we'll 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 a delegate of B. I'm sure you've seen this pattern before as it's used all over 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;

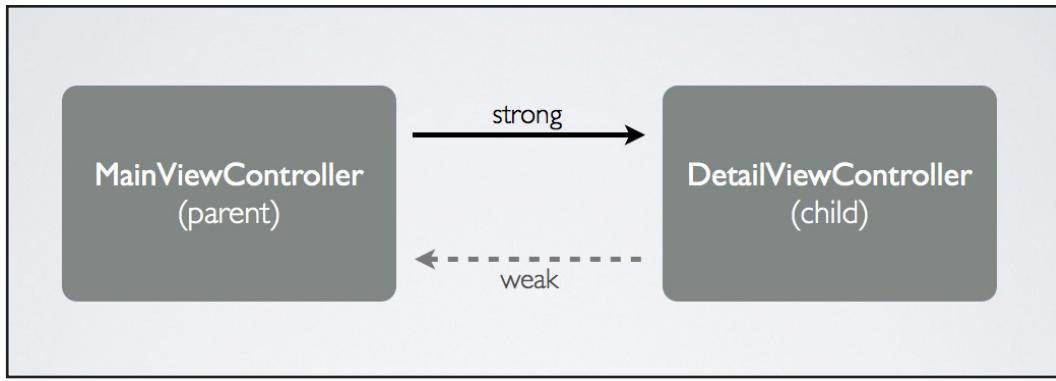
@end
```

We'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 we have a situation where two objects point at each other:





If both of these pointers were strong, then we 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 the exception.

In **DetailViewController.m**, synthesize the delegate:

```
@synthesize delegate;
```

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 we added the `#import` statement to the `.h` file earlier, instead of to the `.m` file.)

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



```

- (void)tableView:(UITableView *)tableView
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 we 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 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. We'll see examples of ownership cycles that do cause problems when we talk about blocks in the second part of this tutorial.



The Detail screen isn't very exciting yet but we 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 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.

Synthesize these properties in **DetailViewController.m**:

```
@synthesize artistName;
@synthesize navigationBar;
```

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!

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

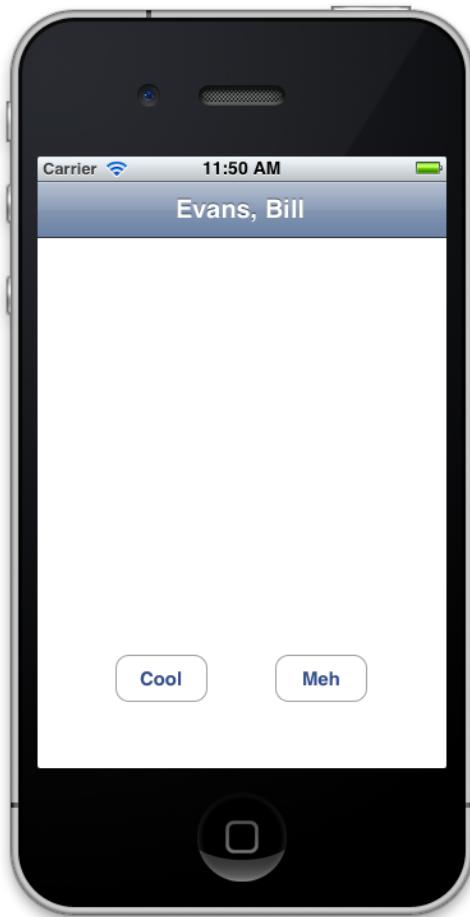
```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [theTableView deselectRowAtIndexPath:indexPath animated:YES];

    NSString *artistName = [searchResults
        objectAtIndex: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:





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 we log the values of `artistName` and `navigationBar` in the `dealloc` method:

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

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

```
Artists[833:207] dealloc DetailViewController
Artists[833:207] artistName 'Evans, Bill'
Artists[833:207] navigationBar <UINavigationBar: 0x686d970; frame = (0 0;
320 44); autoresize = W+BM; layer = <CALayer: 0x686d9e0>>
```

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 we can test the `viewDidUnload` method from **MainViewController.m**. To do this, add some `NSLog()` statements to that method:

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

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

    soundEffect = nil;
}
```

Run the app and open the Detail screen. Then from the Simulator's Hardware menu, choose Simulate Memory Warning. In the Debug pane you should see this:

```
Artists[880:207] Received memory warning.
Artists[880:207] tableView (null)
Artists[880:207] searchBar (null)
```

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



Just to see the difference, let's make them strong references in **MainViewController.h**:

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

Run the app again and repeat the simulated memory warning. Now the Debug pane shows that the objects are still alive:

```
Artists[912:207] Received memory warning.
Artists[912:207] tableView <UITableView: 0xa816400; . . .>
Artists[912:207] searchBar <UISearchBar: 0x8821360; . . .>
```

It's not necessarily wrong to make outlets strong but then you accept responsibility for setting these properties to nil by hand in `viewDidUnload`.

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:

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

Run the app and simulate the low-memory warning.

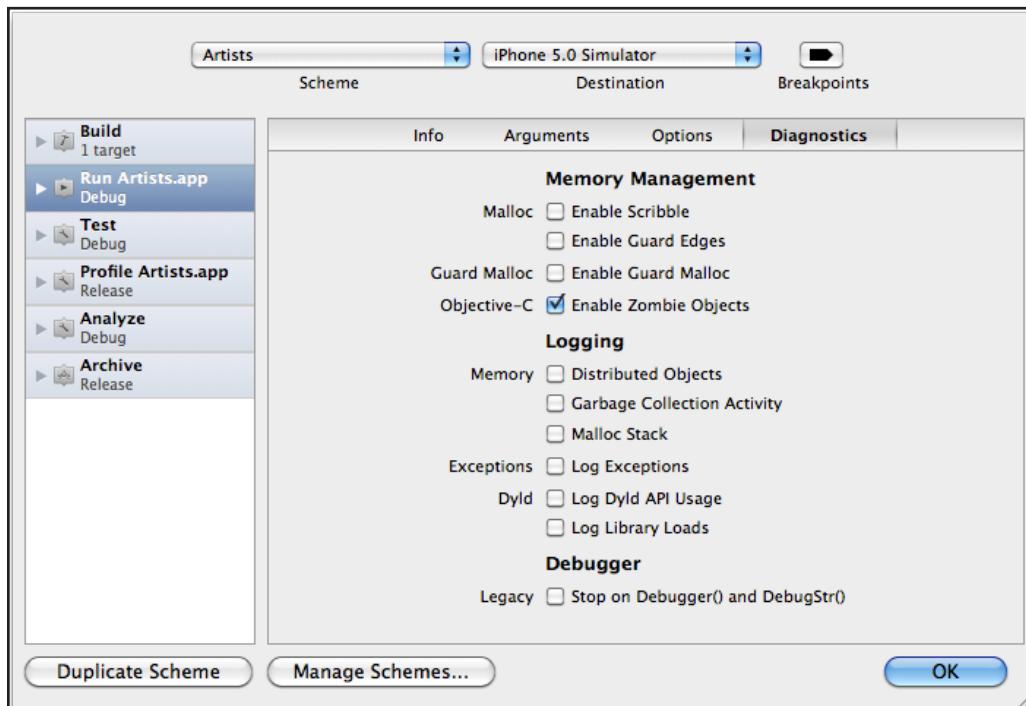
```
Artists[982:207] Received memory warning.
Artists[982:207] *** -[UITableView retain]: message sent to deallocated
instance 0x7033200
```



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 viewDidLoad was called (its only owner was the main view). If this was a true weak pointer, then its value would be set to nil. Remember, that was the cool feature of "zeroing" weak pointers. We 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 we saw happening here. That should illustrate why these things are called "unsafe".

Note: we caught this bug because I enabled zombies in the Diagnostics tab for this scheme. To see this settings panel, choose Product -> Edit Scheme... from the menubar.



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 iOS 4.

Using ARC on iOS 4

Because ARC is largely a new feature of the LLVM 3.0 compiler and not of iOS 5, you can also use it on iOS 4.0 and up. The only part of ARC that does require iOS 5 are the weak pointers. 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 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`!

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.** I'll also explain 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.



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, we're going to continue our 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 - we'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.

We'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.

Add a new **UIView subclass** to the project named **AnimatedView**. 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 a block that takes four parameters: `context`, `rect`, `totalTime` and `deltaTime`. The `context` and `rect` are for drawing, 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

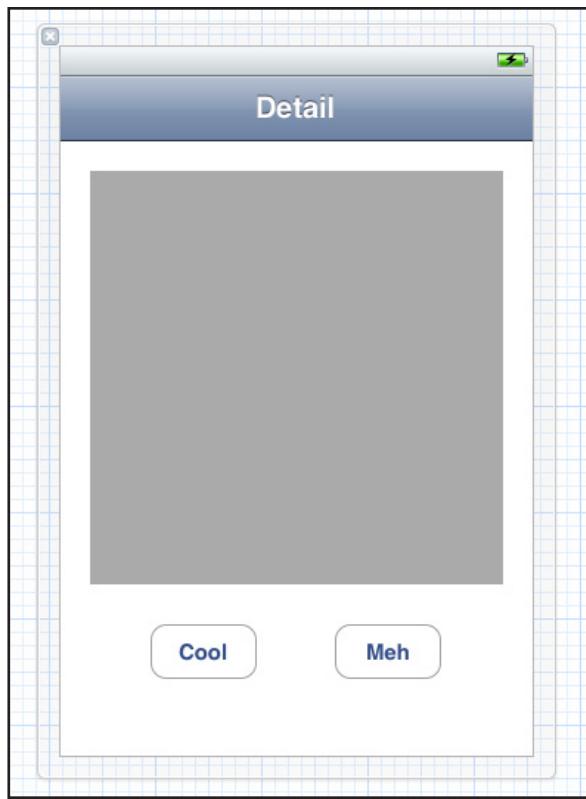
@synthesize block;

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

@end
```

The implementation doesn't do much yet, but first I want to hook up this class to the nib. Open **DetailViewController.nib** 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 we can actually see what we're doing, and 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 our view subclass for this view.

Add an outlet property for the view in **DetailViewController.h**. Again this is a weak property because it's an outlet for a subview:

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

Also add a forward declaration so the compiler knows that AnimatedView is an object:

```
@class AnimatedView;
```

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

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

And synthesize the property:

```
@synthesize animatedView;
```

The final step is to connect the view from the nib to this new property.

After you've done that, build and run the app to make sure everything still works. When you close the Detail screen, the Debug pane should also say:



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

I put an NSLog() in dealloc so we can verify the AnimatedView is truly deallocated when we'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 we will soon see, sometimes those strong pointers aren't immediately obvious.

Let's make our view actually do something. We'll add a timer to AnimatedView. Every time it fires we ask the view to redraw itself, and inside the drawRect method we invoke the block. These are the changes to **AnimatedView.m**:

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

@implementation AnimatedView
{
    NSTimer *timer;
    CFTimeInterval startTime;
    CFTimeInterval lastTime;
}

@synthesize block;

- (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 -- we create the timer in initWithCoder and we stop the timer in dealloc -- but already we're dealing with an ownership cycle. Run the app and close the Detail screen. You'll notice that the dealloc method from AnimatedView isn't called anymore!

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 we explicitly release one of these objects, they will keep each other alive indefinitely. I opted to break this particular retain cycle by adding a stopAnimation method. Change dealloc and stopAnimation to the following:

```

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

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

```

Also add the method signature for stopAnimation to the AnimatedView header:

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

Before the AnimatedView gets released, the user of this class should call stopAnima-



tion. In our 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 ivar 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 us any good. The NSTimer is still owned by another object, the run loop, and because the timer still has a strong reference back to AnimatedView it will keep AnimatedView existing forever. The timer does not release its target until we invalidate it.

Let's make the view do some drawing. We're not going to make it animate just yet, we'll simply draw the same thing over and over. For now we just put the name of the selected artist in the view. 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 we'll create a UIFont object and calculate how big the text will be when drawn. Inside the block we'll use that textSize to center the text in the rectangle and then we draw it. Just to show you that this block is called every couple of milliseconds, I've also added an NSLog().

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 we access self.artistName inside the block. Now DetailViewController will never be deallocated even after it gets closed because the block keeps holding on to it. The timer also keeps running because we never got around to calling stopAnimation (although we 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->ivar behind the scenes and therefore still refers to self.

For example, we 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];
};
```

Any variables prefixed with the `__block` keyword were not retained by the block. 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)
    {
        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. We let the block capture `weakSelf` instead of `self`, so there is no ownership cycle. However, we shouldn't actually use `weakSelf` inside the block. Because this is a weak pointer, it will become `nil` when `DetailViewController` is deallocated. While it is allowed to send messages to `nil`, it's still a good idea to check inside the block whether the object is still alive. Even better, we temporarily turn it into a strong reference so that the object cannot be destroyed out from under us while we're using it:

```
DetailViewController *strongSelf = weakSelf;
if (strongSelf != nil)
{
    CGPoint textPoint = CGPointMake((rect.size.width -
        textSize.width)/2, (rect.size.height - textSize.height)/2);
    [strongSelf.artistName drawAtPoint:textPoint withFont:font];
}
```

For the Artists app this is probably a bit of overkill. We could have simply used `weakSelf` and everything would have worked fine. 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 we were using the `artistName` instance variable directly, we 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. You will need to take additional steps to make sure it does 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 we specified the ownership qualifier `copy` instead of `strong`. If you try it with `strong`, the app will crash immediately 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 the absence of a `retain` call 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 now 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 we'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 you're also not allowed to override these methods.

In my opinion, this 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 `NSNotificationCenter`.

To demonstrate the preferred way of making singletons, we'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:

```
+ (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 we 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. 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 then the output would be random numbers:

```

Woot[2186:207] Number: 67
Woot[2186:207] String: 0x4bab5

```

With ARC, however, the output is:

```

Artists[2227:207] Number: 10120117
Artists[2227:207] 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 = [NSArray arrayWithObjects:(id)color1.CGColor,
        (id)color2.CGColor, (id)color3.CGColor, nil];

    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 we'll use that to create a simple animation later.

We use the `CGGradientCreateWithColors()` function from Core Graphics to construct the gradient. This takes a pointer to a `CGColorSpaceRef` object, an array of `CGColor-`



Ref 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, we can write:

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

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

Because NSArray can only hold Objective-C objects, not Core Foundation objects, we have to cast that CGColorRef 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 we 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 we need to cast colors to a CFArrayRef 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 we still want to keep ARC responsible for releasing the NSArray object, so we use a simple __bridge cast to indicate that no transfer of ownership is taking place.

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

No __bridge statement was necessary when we casted the CGColorRef objects to id. 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 as we will be calling this method from our 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.

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 our 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 = [NSArray arrayWithObjects:(__bridge id)cgColor1,
    (__bridge id)cgColor2, (__bridge id)cgColor3, nil];
```

If you do this in your app it will crash. The reason is simple, we create a `UIColor` object that is not autoreleased but retained (because we 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!

It does work when you do this:

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

Because we now allocate the `UIColor` object using a method that returns an autoreleased object, it stays alive until the autorelease pool is flushed. If you don't want to worry about these weird situations, then mix and match as little with Core Foundation as possible. :-)



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 we take the name of the artist and use that to derive three colors. Each color contains three components (red, green, blue) so that makes nine color components in total. First we loop through the name of the artist, converted to lowercase, and transform each character into a value between 0.0f and 1.0f using a simple 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;
}

```

Once we have these color components, we turn them 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];

```

We can do all of this outside of the block because it only needs to happen once. Inside the block we do the trick with weakSelf and strongSelf again and then calculate 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. I used a sine function to ease the animation in and out.

Now that we have the colors and the midpoint position, we can create the new gradient object:



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

We access the GradientFactory object through the sharedInstance class method and then call newGradient. The very first time we do this the GradientFactory instance is created and from then on we simply re-use that same instance.

Then we use a Core Graphics function to draw that gradient between the start and end points:

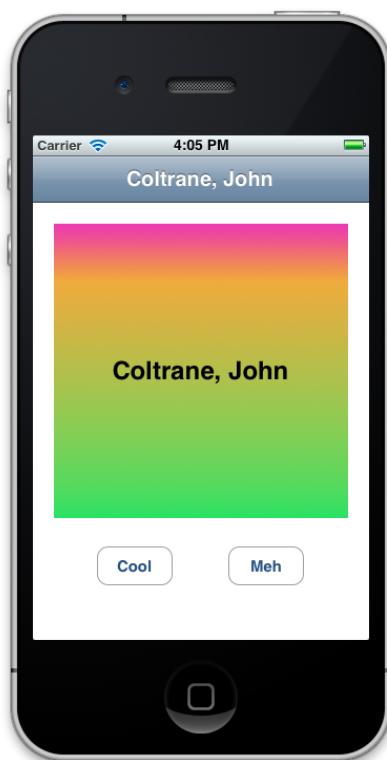
```
CGContextDrawLinearGradient(context, gradient, startPoint, endPoint,
    kCGGradientDrawsBeforeStartLocation | 
    kCGGradientDrawsAfterEndLocation);
```

And finally, we release the gradient object:

```
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

We already saw 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 here
```

However, when you do 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), we 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 here
```

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:

```
__autoreleasing NSString *s;  
@autoreleasepool  
{  
    s = [NSString stringWithFormat: . . .];  
}  
NSLog(@"%@", s); // 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 you press the Cool button we're going to capture the contents of the `AnimatedView` and save this image to a PNG file.

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

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



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];
}
```

Also add an import for QuartzCore at the top:

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

The coolAction method is quite straightforward. First we capture 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 we turn that image into a PNG file, which gives us an NSData object. We also create 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"];
```



```

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

```

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 our 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.

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

```

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

    NSError* error;
    if (![data writeToFile:(NSString *) options:(NSDataWritingOptions) error:(NSError *__autoreleasing *)])
    {
        BOOL writeToFile:(NSString *) atomically:(BOOL)
        [self.delegate BOOL writeToFile:(NSString *) options:(NSDataWritingOptions) error:(NSError *__autoreleasing *)];
        [self.delegate BOOL writeToURL:(NSURL *) atomically:(BOOL)
        - (IBAction BOOL writeToURL:(NSURL *) options:(NSDataWritingOptions) error:(NSError *__autoreleasing *)];
        [self.delegate detailViewController:self didPickButtonWithIndex:1];
    }
}

```

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 we've never 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 our 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 our 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. 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;
    NSNumber *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 use `__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 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 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 should 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 could use your own autorelease pool to prevent the app from running out of memory:

```
for (int i = 0; i < 10000; i++)
{
    @autoreleasepool
    {
        NSString *s = [NSString stringWithFormat:. . .];
    }
}
```

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 6 times faster than NSAutoreleasePool was, so plugging an @autoreleasepool block into a tight loop should not slow down your app at all.

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 object, 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 that there is no such thing as autorelease a Core Foundation object. 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`.

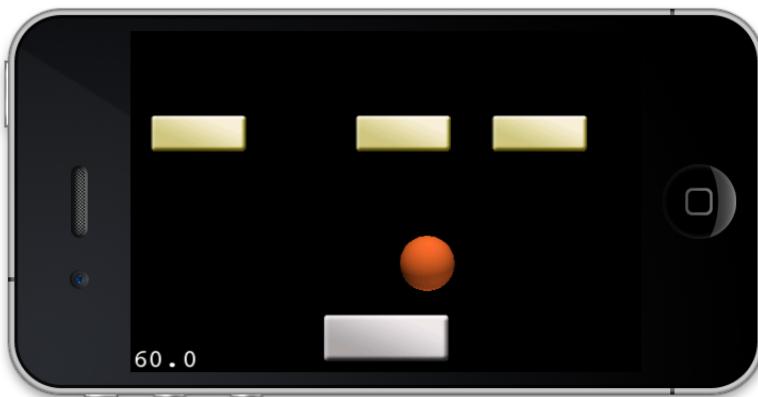
Cocos2D and Box2D

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 tutorial 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>





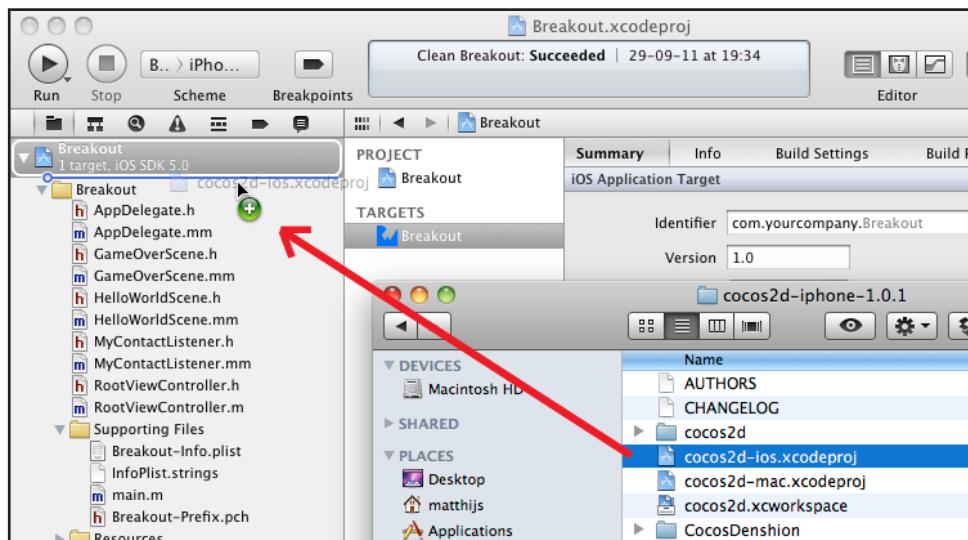
I adapted the code for the latest version of Cocos2D at the time of writing (v1.0.1 stable). You can find the starter code with this chapter's resources. This is a simple project that uses Cocos2D and Box2D to make a ball & paddle game. Because Box2D is written in C++, most of the game's source code files are Objective-C++ (they have a .mm file extension).

The starter code does not include Cocos2D, so first we need to install that. Download the complete package from:

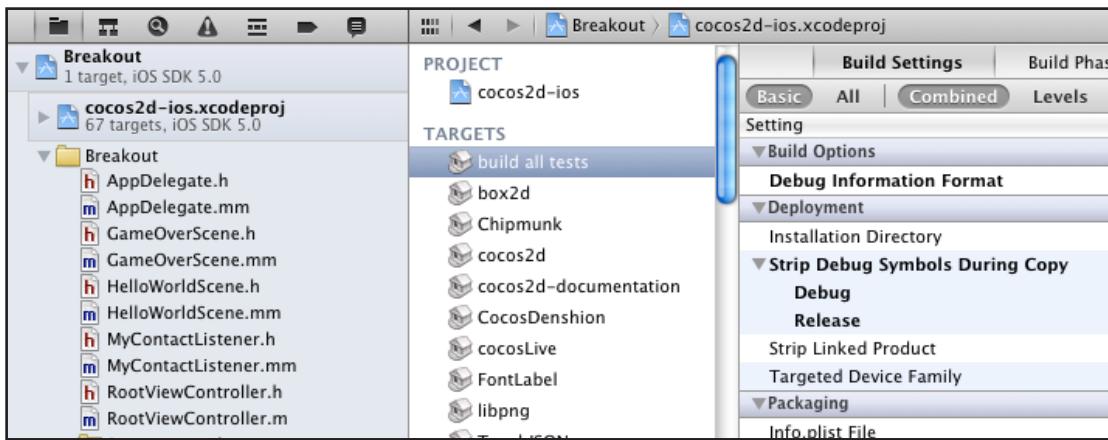
- <http://cocos2d-iphone.googlecode.com/files/cocos2d-iphone-1.0.1.tar.gz>

You may also download a newer version from the v1.x branch but there could be slight differences with my descriptions.

After the download is complete, unzip the package. Inside the cocos2d-iphone-1.0.1 folder you will find a file named cocos2d-ios.xcodeproj. Drag this file into the Breakout project in 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. We will now add some of these libraries to our Breakout project so that it will link with cocos2d.

Breakout already links with the following system frameworks:

- AVFoundation
- AudioToolbox
- CoreGraphics
- Foundation
- OpenAL
- OpenGL ES
- QuartzCore
- UIKit
- libz

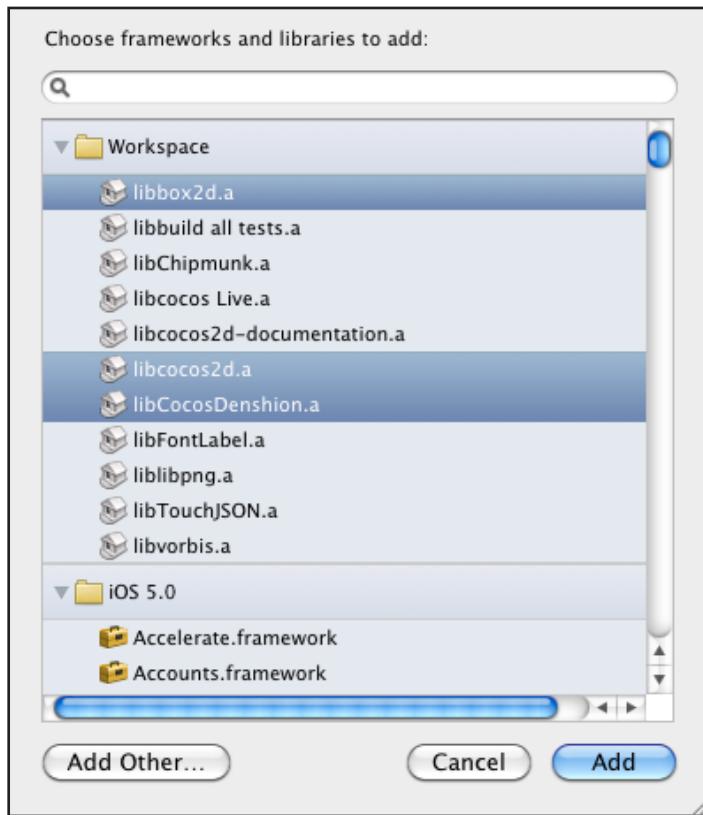
I mention this because you will also need to add these frameworks to your own game projects in order to use Cocos2D and the CocosDenshion audio library.

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:

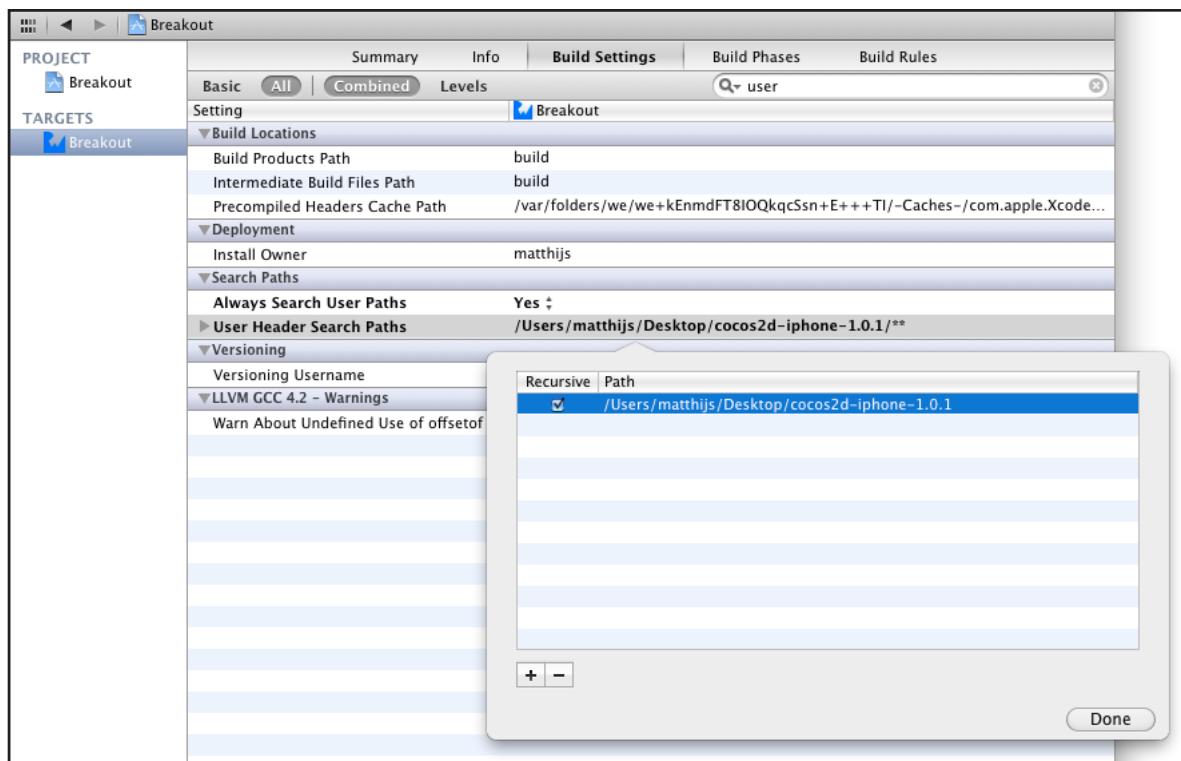
- libbox2d.a



- libcocos2d.a
- libCocosDenshion.a



We are almost done. In order to make Xcode find the .h files from Cocos2D, we 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 check the Recursive option!

Note: If the path to the cocos2d folder contains spaces, then put double quotes around it, otherwise Xcode gets confused.

You should now be able to build and run the game!

The project is currently built with LLVM-GCC so before we do the conversion to ARC let's first switch the compiler to LLVM 3.0. You do this in the **Build Settings** tab on the Project screen, under **Build Options**. I also 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 run the app again. You should get no errors or warning messages. That's always a good sign. :-)

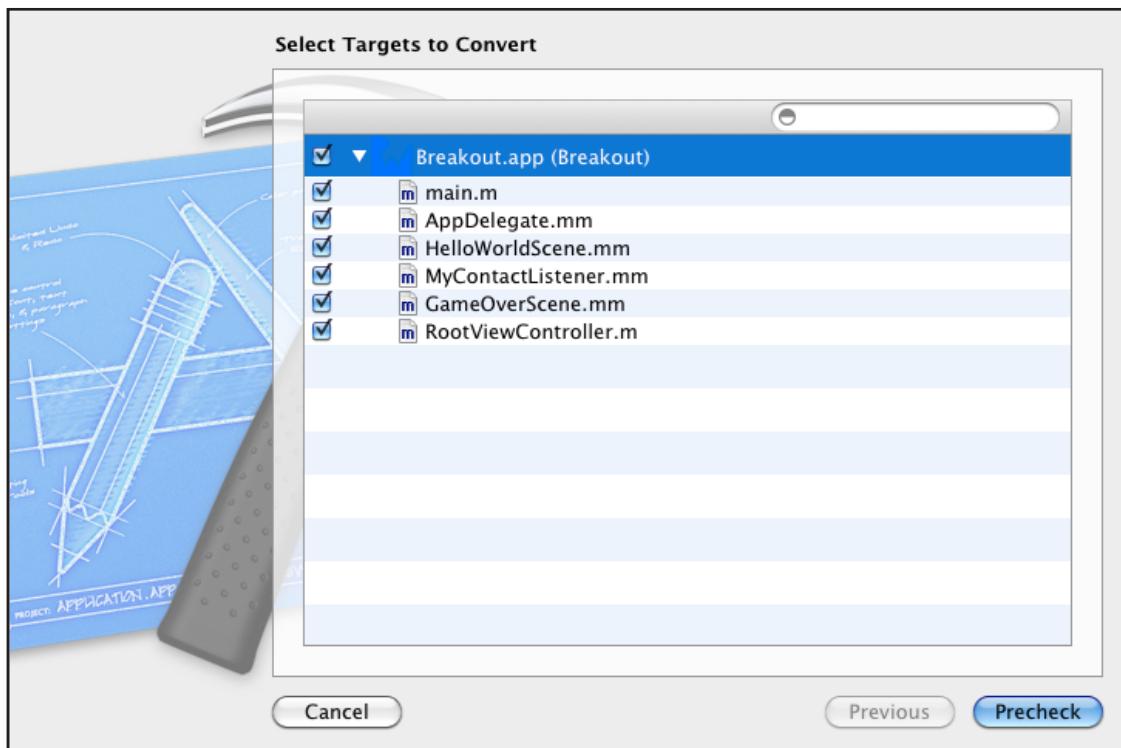
Cocos2D is a pretty big library. It would be insane to try and convert Cocos2D itself to ARC. That would probably take you a few weeks. It's smarter to convert the rest of the game but to leave Cocos2D as-is. It's very handy that Cocos2D already comes as a static library. That means we can tell Xcode to compile the Breakout project as ARC but the Cocos2D library as non-ARC.

There is only one problem: some of the .h files from Cocos2D do things that are not compatible with ARC. This is no problem when the Cocos2D static library gets com-



piled because that doesn't use ARC anyway. However, it **is** a problem when we #import those .h files into our Breakout source files. The compiler throws a tantrum when we do that. So at least with v1.0.1 of Cocos2D we will have to make some changes to the source code of Cocos itself to fix these errors. It is possible that you're working with a later version of Cocos2D where these issues have been fixed, so the next section may not apply to you.

First, let's run the conversion tool. Select the Breakout project and then choose **Edit\Refactor\Convert to Objective-C ARC**. Select all the files under Breakout.app:



If the pre-check gives any errors on ccCArray.h or CCDirectorIOS.h, or any other files from Cocos2D itself, then Cocos first needs fixing. The following instructions are based on an excellent blog post by Jerrod Putman:

- <http://www.tinytimgames.com/2011/07/22/cocos2d-and-arc/>

CCDirectorIOS.h gives an error on the following piece of code:

```
@interface CCDirectorFast : CCDirectorIOS
{
    BOOL isRunning;

    NSAutoreleasePool *autoreleasePool;
}
```



This attempts to use `NSAutoreleasePool` but as you know that is no longer available. Remove that line. Now open **CCDirectorIOS.m** and search for the line `@implementation CCDirectorFast`. Change it to:

```
@implementation CCDirectorFast
{
    NSAutoreleasePool *autoreleasePool;
}
```

We have simply moved the `autoreleasePool` variable from the header into the `@implementation` section, which is now possible with LLVM 3.0.

The other file that gave problems was `ccCArray.h`. One of the errors is "Pointer to non-const type 'id' with no explicit ownership". If we look in the source, that happens here:

```
typedef struct ccArray {
    NSUInteger num, max;
    id *arr;
} ccArray;
```

The offending line is "`id *arr;`". This looks harmless enough, but 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. Unfortunately for us, that's exactly what Cocos2D is doing here.

There are a few workarounds. You can use a `void *` instead. If you haven't done much C programming before, then you can compare `void *` 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 *`. So 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. In the case of Cocos2D, however, we want to change the code as little as possible so changing the type to `void *` is not such a good idea. A better solution is to declare the offending variable `__unsafe_unretained`:

```
typedef struct ccArray {
    NSUInteger num, max;
    __unsafe_unretained id *arr;
} ccArray;
```

ARC will leave this pointer alone now. It inserts no code at all for dealing with `__unsafe_unretained` pointers. That's why they are unsafe.

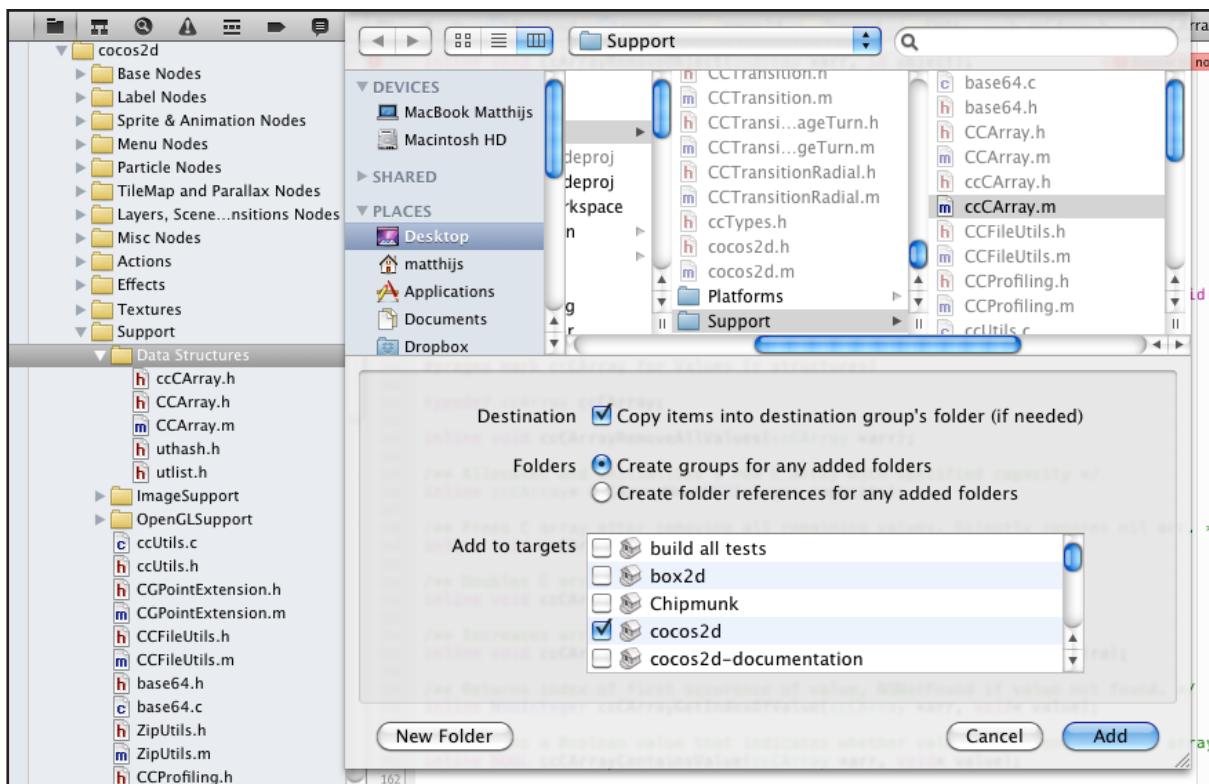
That change was simple enough to make, but unfortunately there are about 30 other issues in `ccCArray.h`. Some of these problems are caused by inline functions that do retains and releases. We need to move these into a new source file, `ccCArray.m`, so that they can be compiled into the `libcocos2d.a` static library.

Jerrod Putman, who did excellent investigative work to make Cocos2D work under ARC, was kind enough to make available the fixed `ccCArray.h` and `.m` files. I have included them in this chapter's source code folder under "Cocos2D Fixes".

To get these fixes into your project,

1. Copy **ccCArray.h** and **ccCArray.m** into your Cocos2D folder, under **cocos2d/Support/**. This will overwrite the existing `ccCArray.h`.
2. The `ccCArray.m` file is new. You will need to add it to the **cocos2d-ios project**, under **cocos2d/Support/Data Structures**. Make sure you select the **cocos2d target** so it gets compiled into the main Cocos2D library.





Build the app (in non-ARC mode) to make sure Cocos2D still compiles with these new files and then run the ARC conversion tool again.

We're still not done with those pre-check errors but at least now they are all inside the Breakout project, in `HelloWorldScene.mm`. Cocos2D itself is now ready for use in our ARC game!

Note: You may need to do something similar on `CCActionManager.h` and maybe other Cocos2D source files as well when you convert your own projects. If the compiler is complaining about an object being used inside a struct, then add `__unsafe_unretained` in front of the declaration. For more details, see Jerrod's blog post.

Fixing up Breakout

The first error in `HelloWorldScene.mm` is:

```
Assigning to 'void *' from incompatible type 'CCSprite *__strong'
```



```
59
60     // Create ball body
61     b2BodyDef ballBodyDef;
62     ballBodyDef.type = b2_dynamicBody;
63     ballBodyDef.position.Set(100/PTM_RATIO, 100/PTM_RATIO);
64     ballBodyDef.userData = ball; // Assigning to 'void *' from incompatible type 'CCSprite * _strong'
65     b2Body * ballBody = _world->CreateBody(&ballBodyDef);
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;
b2Body * ballBody = _world->CreateBody(&ballBodyDef);
```

This is related to the errors we just fixed in the Cocos2D code, where you were not allowed to store an Objective-C object into a C struct. Here the situation is slightly different because we'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 *`.

We can't simply do this:

```
paddleBodyDef.userData = (void *)paddle;
```

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 we encountered before with toll-free bridging. The compiler needs to know who will be responsible for releasing the object. In this case, we just want to give the `b2BodyDef` a pointer to the sprite so that we can update the sprite's position when the body moves. We don't want to change owners, so the proper fix is:

```
paddleBodyDef.userData = (__bridge void *)paddle;
```

The following two errors have exactly the same fix.

The remaining errors are all in the `tick:` method:



```

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

```

Here we do the reverse, we 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.

If you look at the proposed changes in the preview window you will see they are very similar to what we did on the Artists app. Properties go from retain to strong, retain and release calls are dropped, and 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!

Other C++ notes

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 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:@"ball.png"];
sprites[0].sprite = ball;
```

No problem whatsoever. It works because a C++ struct is much more powerful than a 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(@"%@", *i);
}
```

This creates a `std::vector` with a strong reference to a `CCSprite` object. 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);
}
```

You can get away with zeroing weak references here because unlike NSArray, std::vector can hold NULL objects just fine.

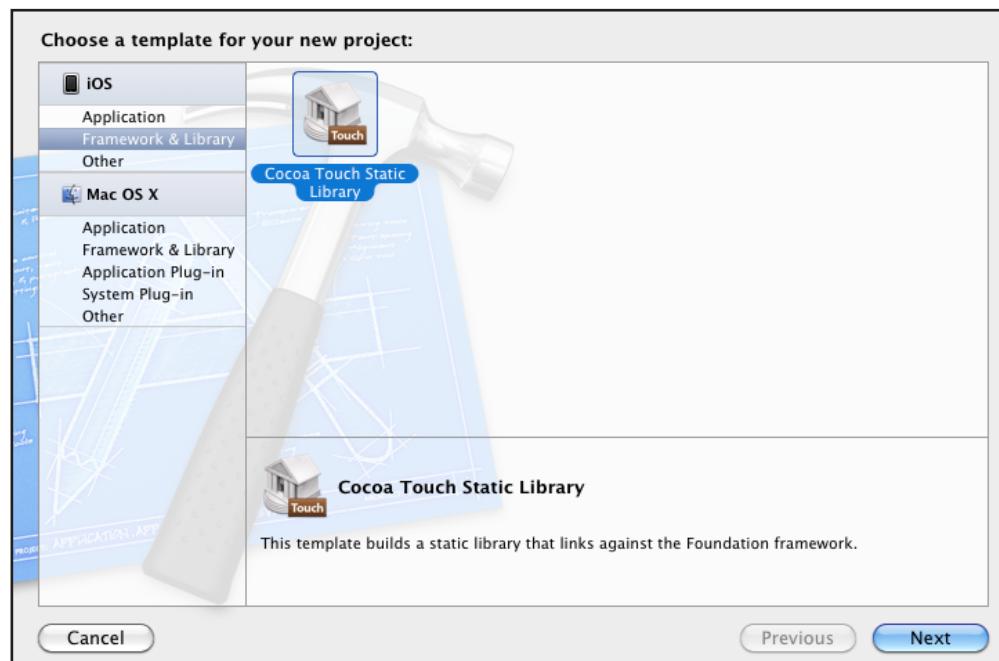
Making your own static library

Not all third-party libraries come 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 separate static libraries that are not compiled with ARC, just like we've done above with Cocos2D.

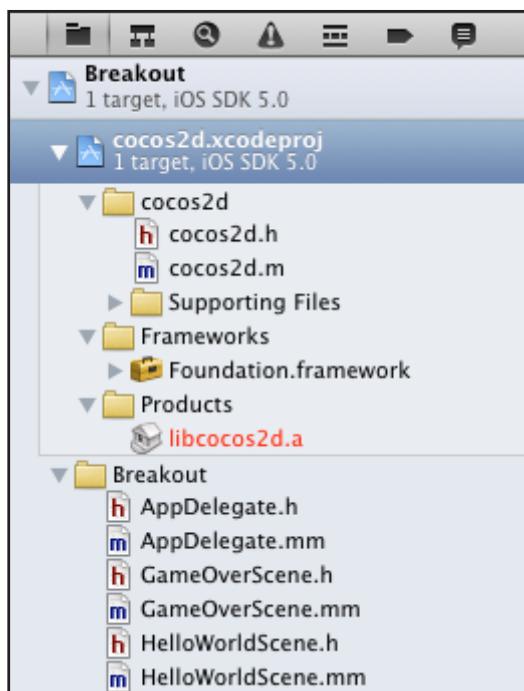
Creating your own static library is pretty easy. In this section we will create one that includes all of the Cocos2D sources that we need for the Breakout project.

First remove the cocos2d-ios project from the Breakout project. Don't delete the files from disk, choose Remove Reference Only.

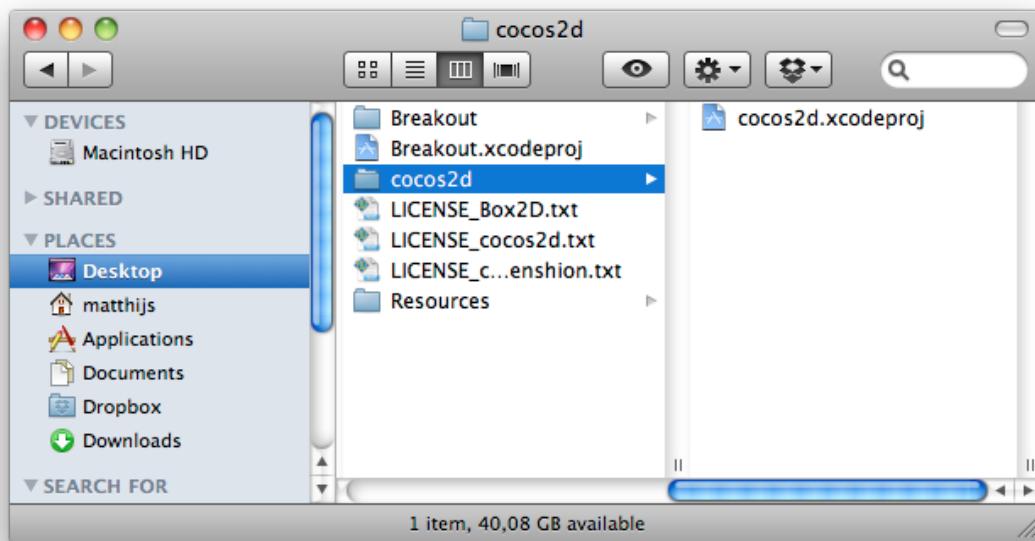
Select the Breakout project and choose **New Project**. Pick the **iOS\Framework & Library\Cocoa Touch Static Library** project template:



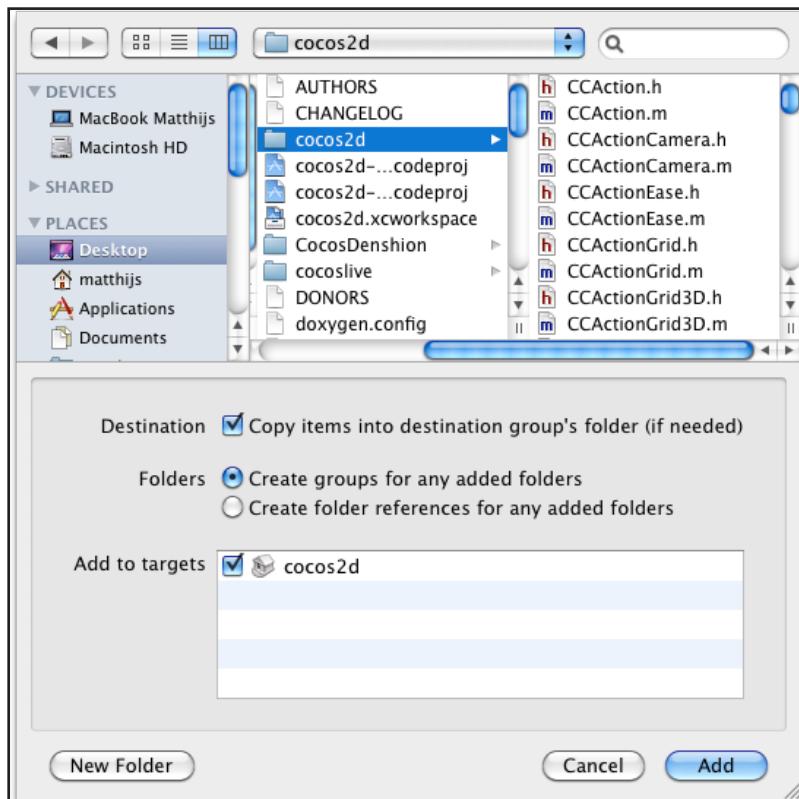
For Product Name choose "cocos2d". Disable Automatic Reference Counting. This will add a new static library project inside the Breakout project, with a few default files:



You can delete the cocos2d source folder. 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, we don't want to add any of the Testbed files).

We need to change some build settings to make this all work. Select the **cocos2d** project and go the Build Settings tab.

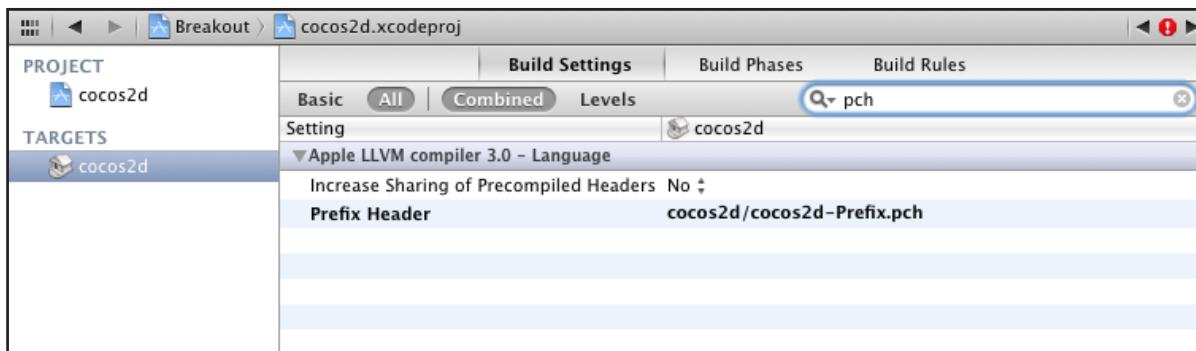
- **Search Paths\Always Search User Paths:** Set to **YES**
- **Search Paths\User Header Search Paths:** Set to **./****

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 make archives for distribution to the App Store because the static library will be installed inside the package too and that should not happen.

One more thing before we can build our 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 we deleted the one from the template earlier, so we 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. Congrats, you just built your own static library!

Switch the active project back to Breakout. In the **Build Phases\Linked Frameworks and Libraries section**, add libcocos2d.a. This is the static library that we've just created. You should be able to build and run the Breakout app now, using our custom version of Cocos2D!

Note: If you're doing this in a real project, then 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 brain-power 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, they 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
```

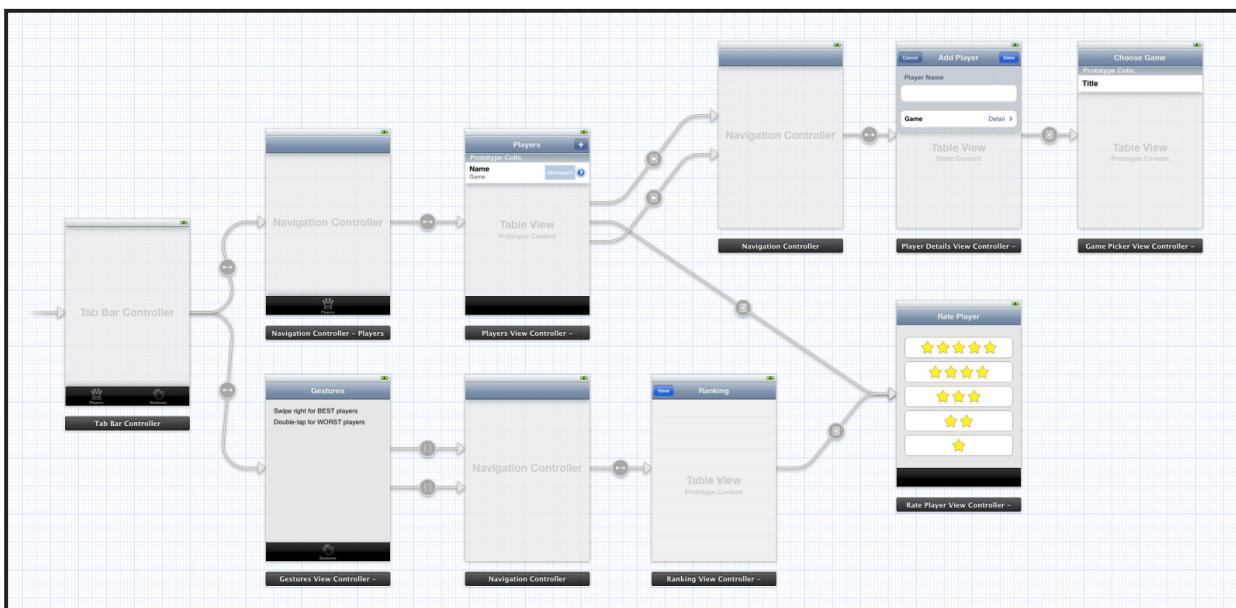
The official documentation for ARC is "Transitioning to ARC Release Notes" that you can find in your Xcode documentation. Other good sources for information are WWDC 2011 videos 323, Introducing Automatic Reference Counting, and 322, Objective-C Advancements in Depth.



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 that we will be building in this tutorial:



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 every-



thing 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 Editor isn't as powerful as Interface Builder yet, there are a few things IB can do that the Storyboard Editor 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.

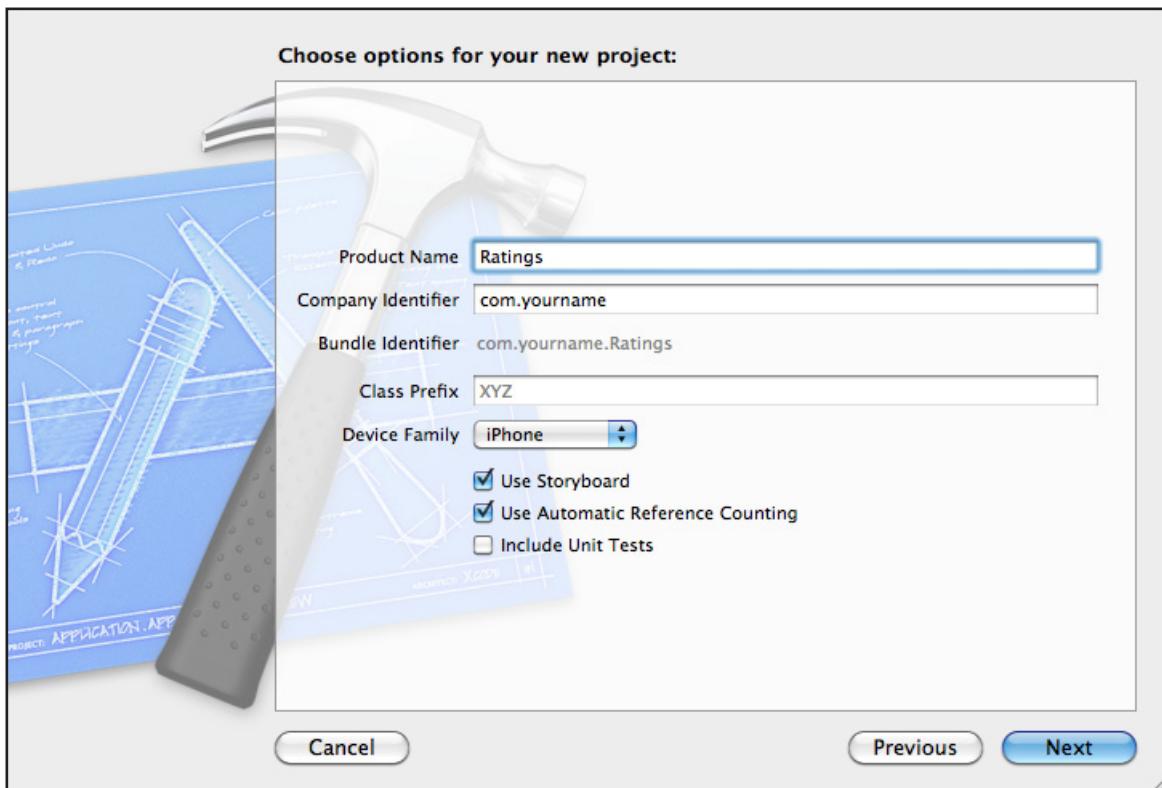
You can still use nibs with iOS 5 and Xcode 4.2. Using Interface Builder isn't suddenly frowned upon now that we have storyboards. If you want to keep using nibs then go right ahead, but know that you can combine storyboards with nibs. It's not an either-or situation.

In this tutorial we'll take a look at what you can do with storyboards. We'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 and create a new project. We'll use the Single View Application template as our starting point and then build up the app from there.

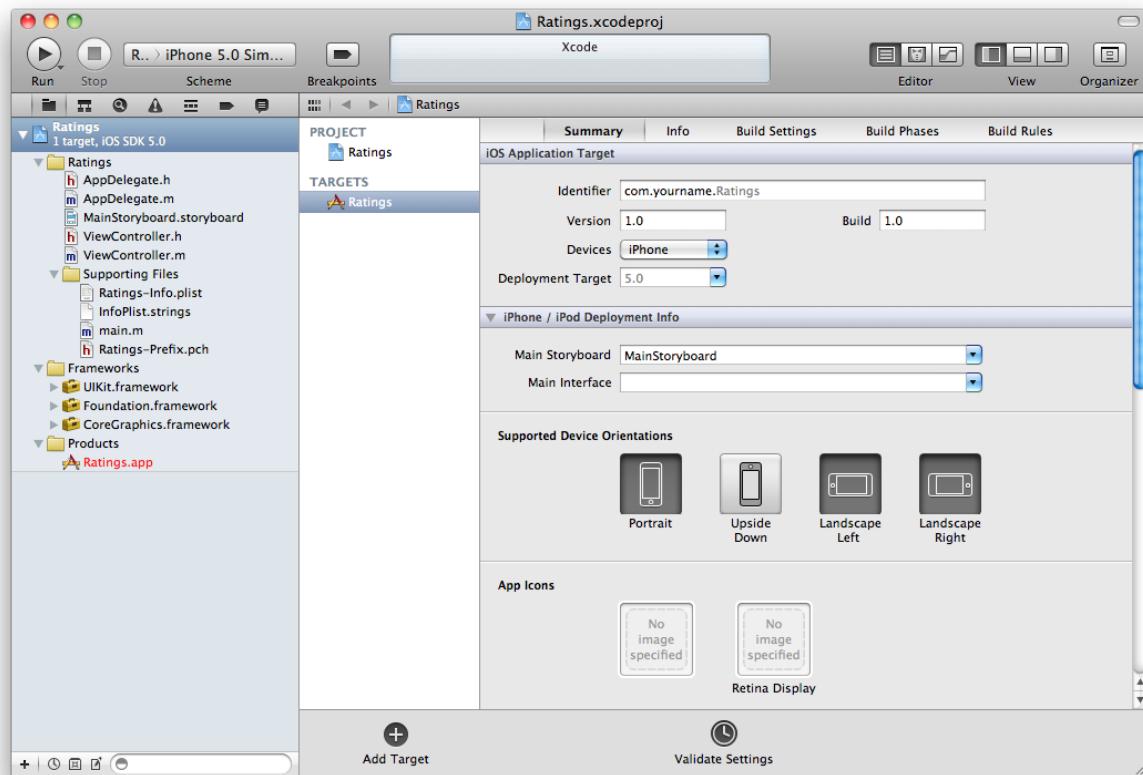


Fill in the template options as follows:

- **Product Name:** Ratings
- **Company Identifier:** the identifier that you use for your apps, in reverse domain notation
- **Class Prefix:** leave this empty
- **Device Family:** iPhone
- **Use Storyboard:** check this
- **Use Automatic Reference Counting:** check this
- **Include Unit Tests:** this should be unchecked

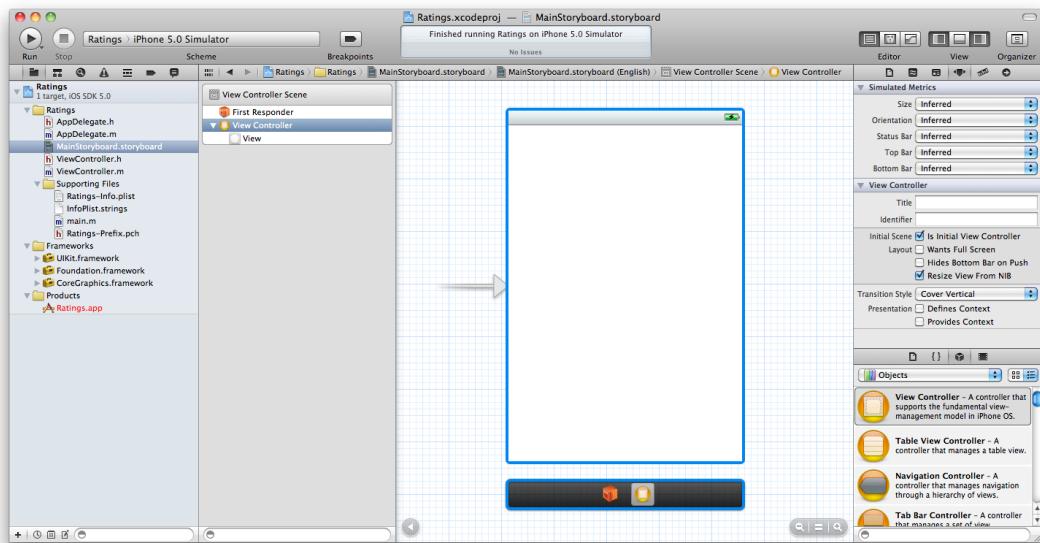
After Xcode has created the project, the main Xcode window looks like this:





Our 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`.

Let's take a look at that storyboard. Click the **MainStoryboard.storyboard** file in the Project Navigator to open the Storyboard Editor:

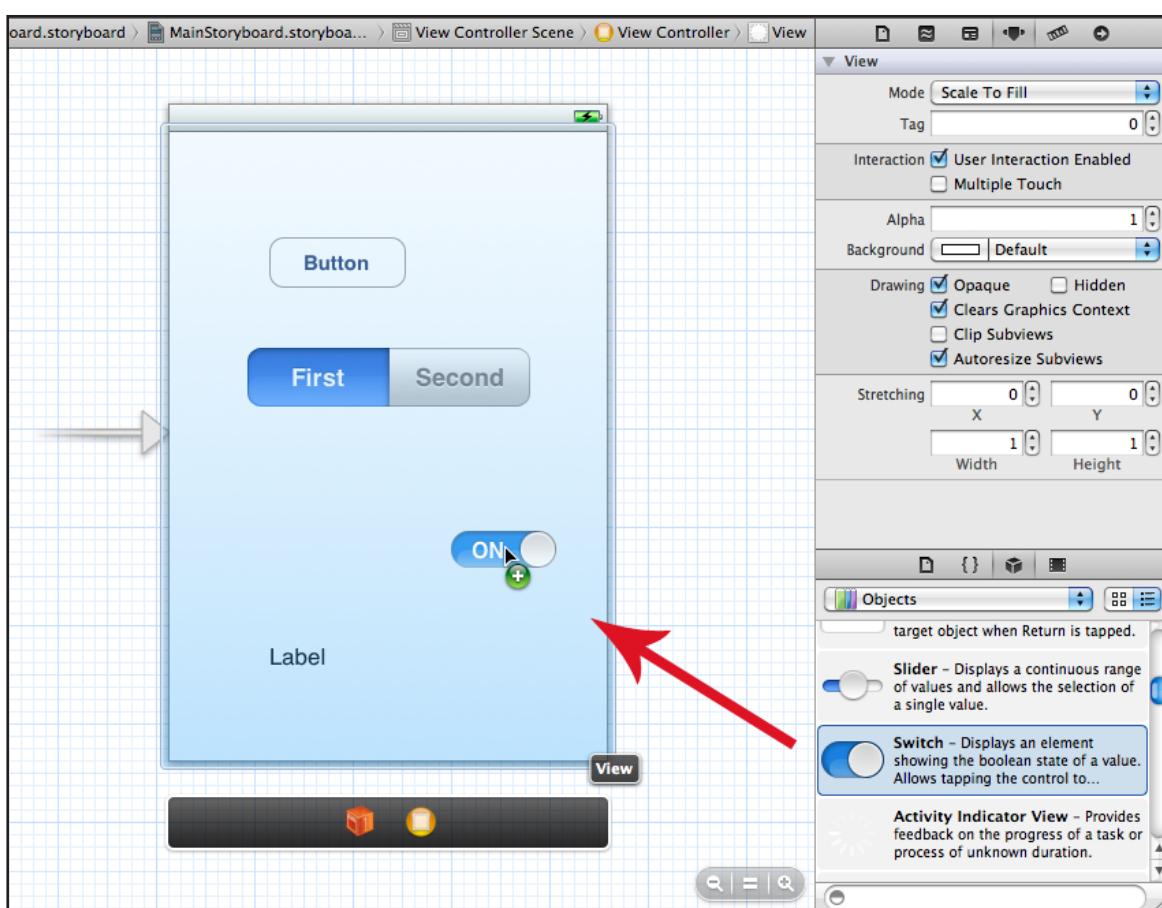


The Storyboard Editor looks and works very much like Interface Builder. You can drag new controls from the Object Library (see bottom-right corner) into your view controller to design its layout. The difference is that the storyboard doesn't contain just one view controller from your app, but all of them.

The official storyboard terminology is "scene", but a scene is really nothing more than a view controller. 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 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.

To get some feel for how the editor works, drag some controls into the blank view controller:

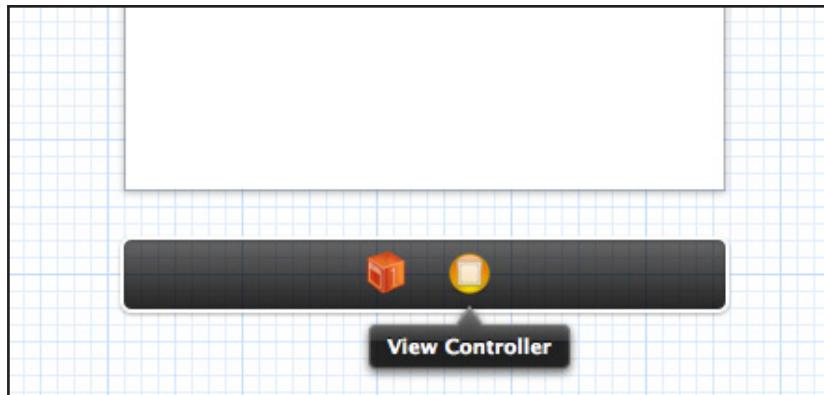


The sidebar on the left is the Document Outline:



In Interface Builder this area lists just the components from your nib but in the Storyboard Editor it shows the contents of all your view controllers. Currently there is only one view controller in our storyboard but in the course of this tutorial we'll be adding several others.

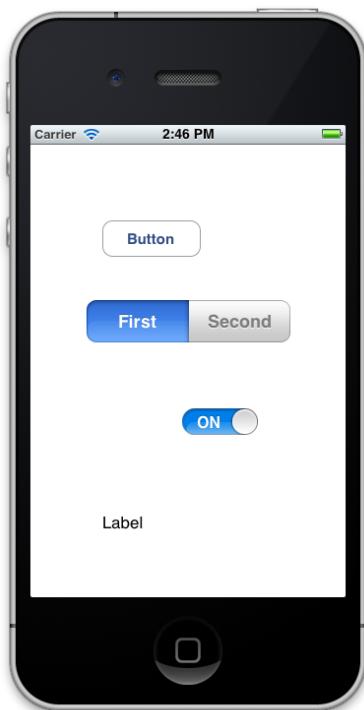
There is a miniature version of this Document Outline below the scene, named the Dock:



The Dock shows the top-level objects in the scene. Each scene has at least a First Responder object and a View Controller object, but it can potentially have other top-level objects as well. More about that later. The Dock is convenient for making connections. If you need to connect something to the view controller, you can simply drag to its icon in the Dock.

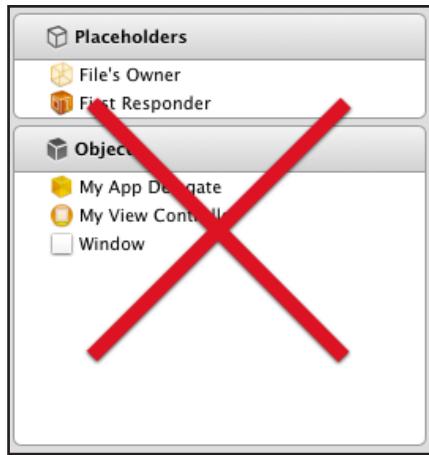
Note: You probably won't be using the First Responder very much. This is a proxy object that refers to whatever object has first responder status at any given time. It was also present in Interface Builder and you probably never had a need to use it then either. As an example, you could hook up the Touch 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 we designed in the editor:



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 our 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;

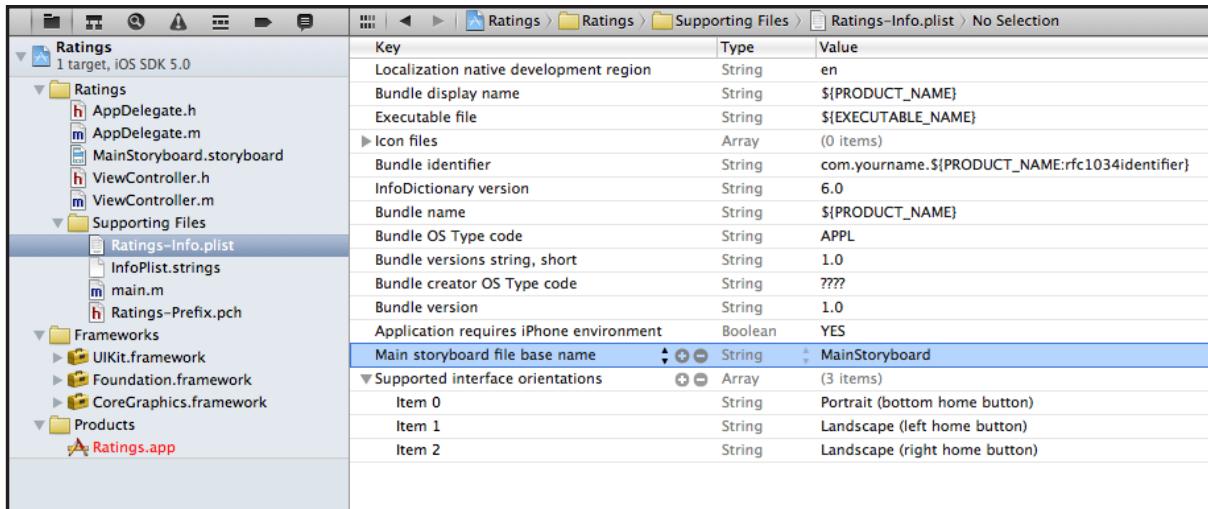
@end
```

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 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** (it's in the **Supporting Files** group) and you'll see this:

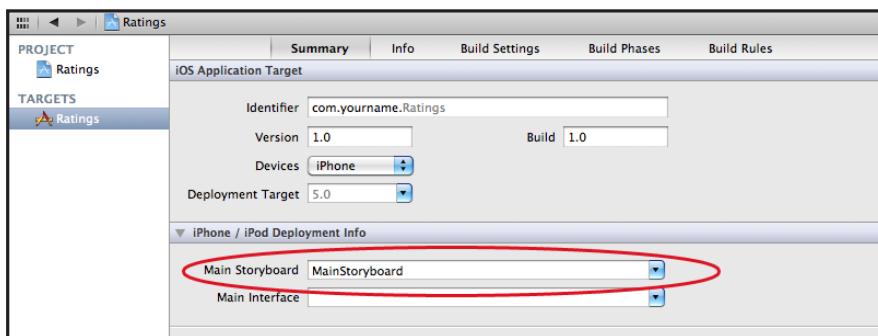




In nib-based projects there was a key in Info.plist named NSMainNibFile, or “Main nib file base name”, that instructed UIApplication to load MainWindow.xib and hook it into the app. Our Info.plist no longer has that setting.

Instead, storyboard apps use the key UIMainStoryboardFile, 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]));
    }
}
```



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

Previously, the last parameter for UIApplicationMain() was nil but now it is NSStringFromClass([AppDelegate class]).

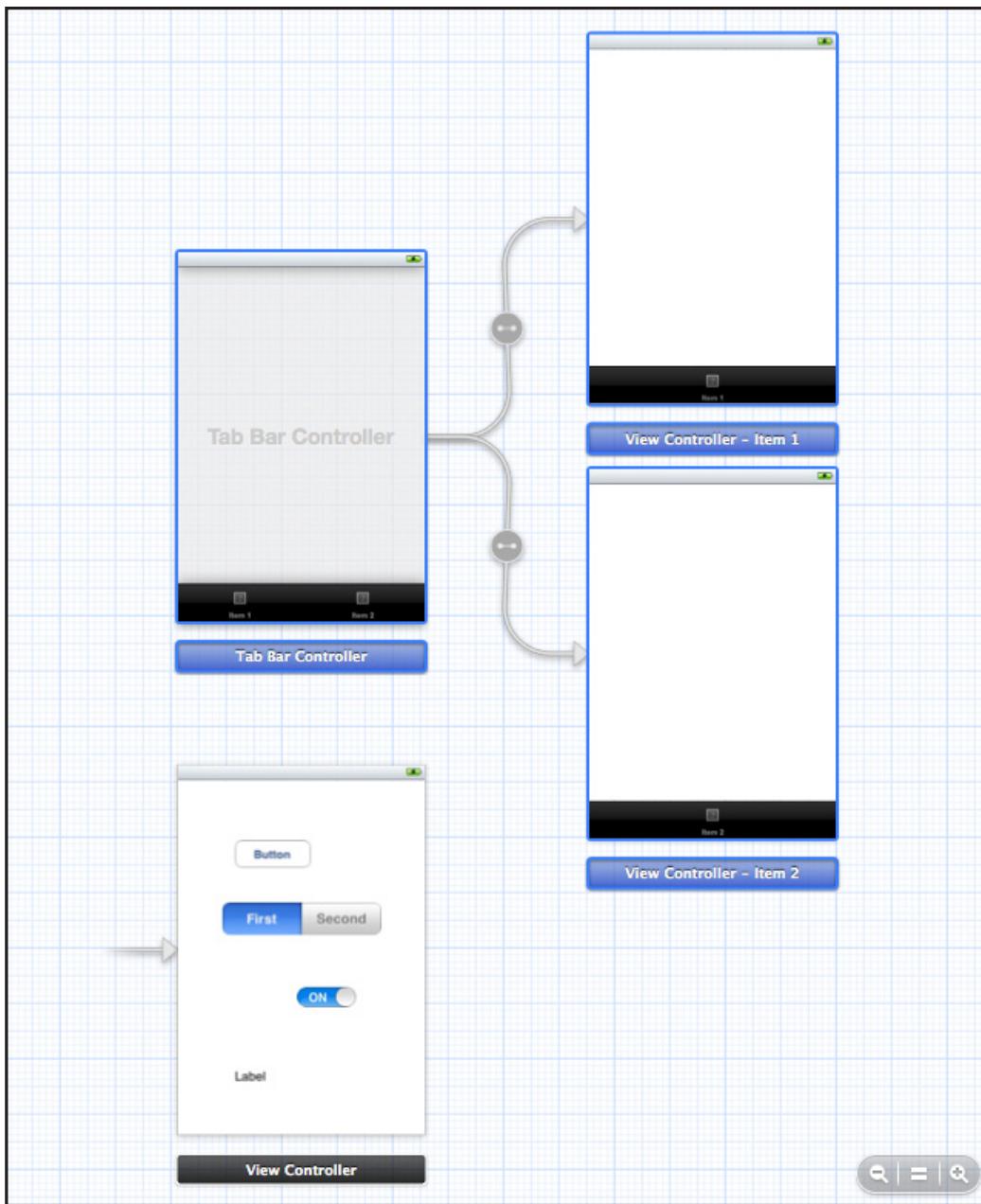
A big difference with having a MainWindow.xib is that the app delegate is not part of the storyboard. Because the app delegate is no longer being loaded from a nib (nor from the storyboard), we have to tell UIApplicationMain specifically what the name of our app delegate class is, otherwise it won't be able to find it.

Just Add It To My Tab

Our 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.

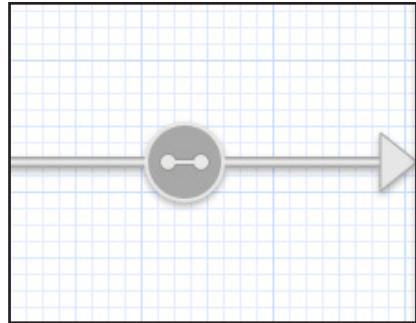




The new Tab Bar Controller comes pre-configured with two other view controllers, one for each tab. UITabBarController is a so-called container view controller because it contains one or more other view controllers. Two other common containers are the Navigation Controller and the Split View Controller (we'll see both of them later). 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 tutorial 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.



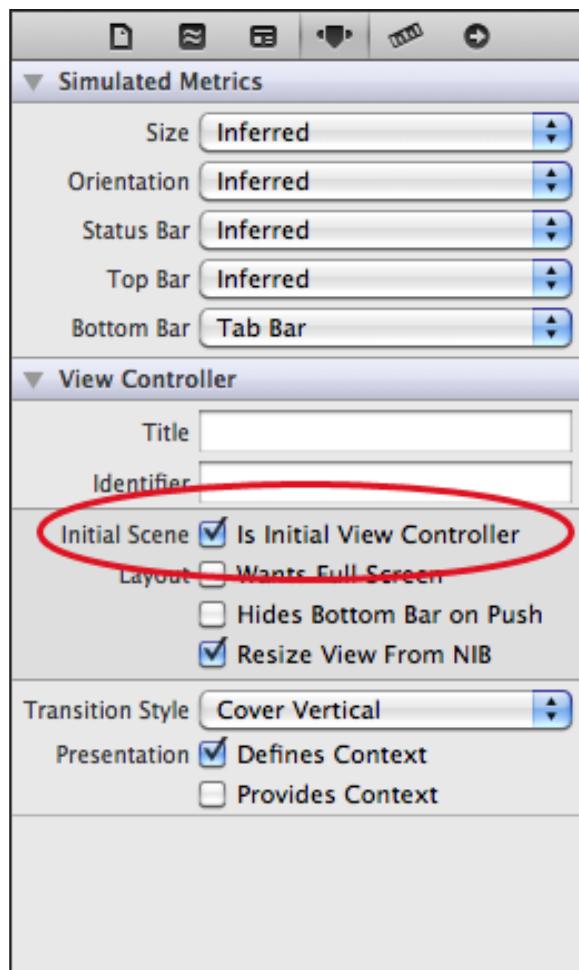


Note: If you want to move the Tab Bar controller and its attached view controllers as a group, you can Cmd-click (or shift-click) to select multiple scenes and then move them around together. (Selected scenes have a thick blue outline.)

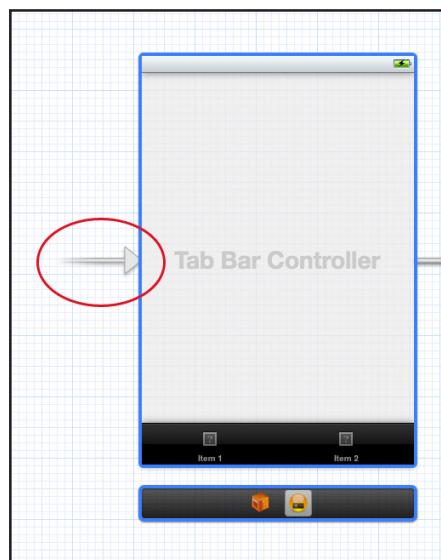
Drag a label into the first view controller and give it the text "First Tab". Also drag a label into the second view controller and name it "Second Tab". This allows us to actually see something happen when you switch between the tabs.

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.

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 of our app.

The storyboard always has a single view controller that is designated the initial view controller, that serves as the entry point into the storyboard.

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 we could have used, but it's good to know how this works so you can also create one by hand if you have to.

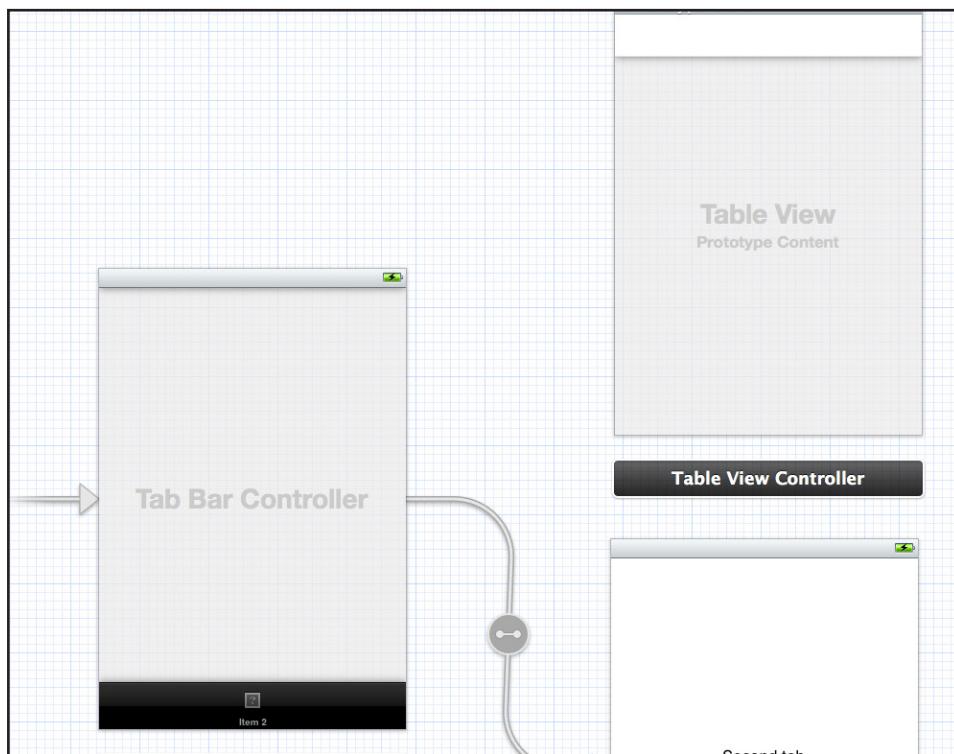
You can remove the view controller that was originally added by the template as we'll no longer be using it. The storyboard now contains just the tab bar and the two scenes for its tabs.

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

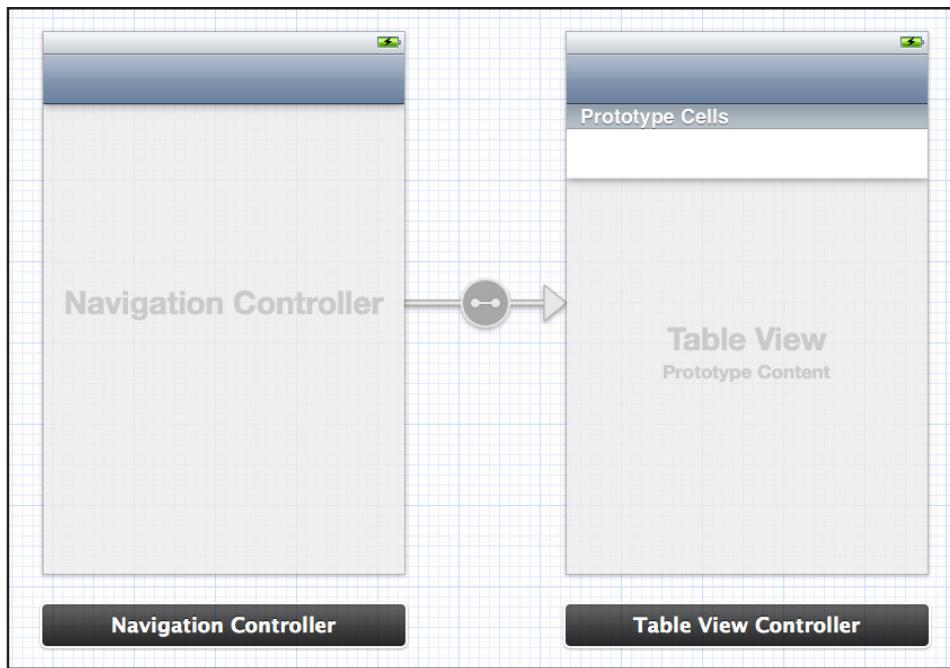
The two scenes that are currently attached to the Tab Bar Controller are both regular UIViewControllers. I want to replace the scene from the first tab with a UITableViewController instead.

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 scene used to be:



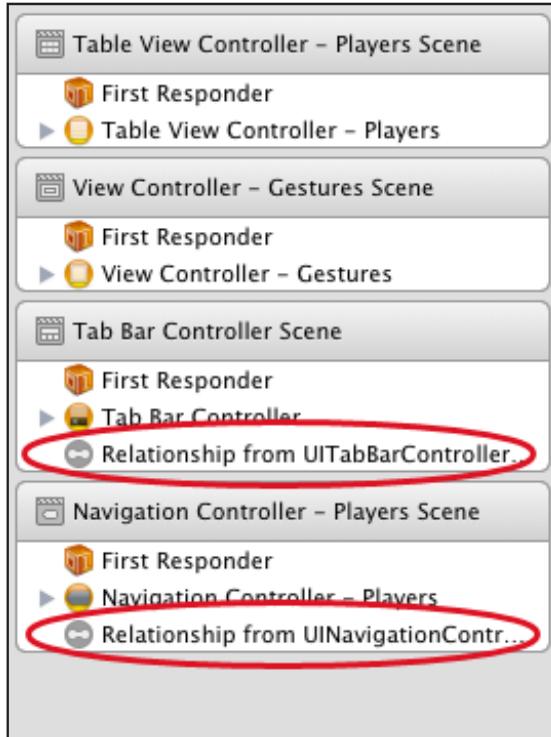
With the Table View Controller selected, choose **Editor\Embed In\Navigation Controller** from Xcode's menubar. This adds yet another view controller to the canvas:





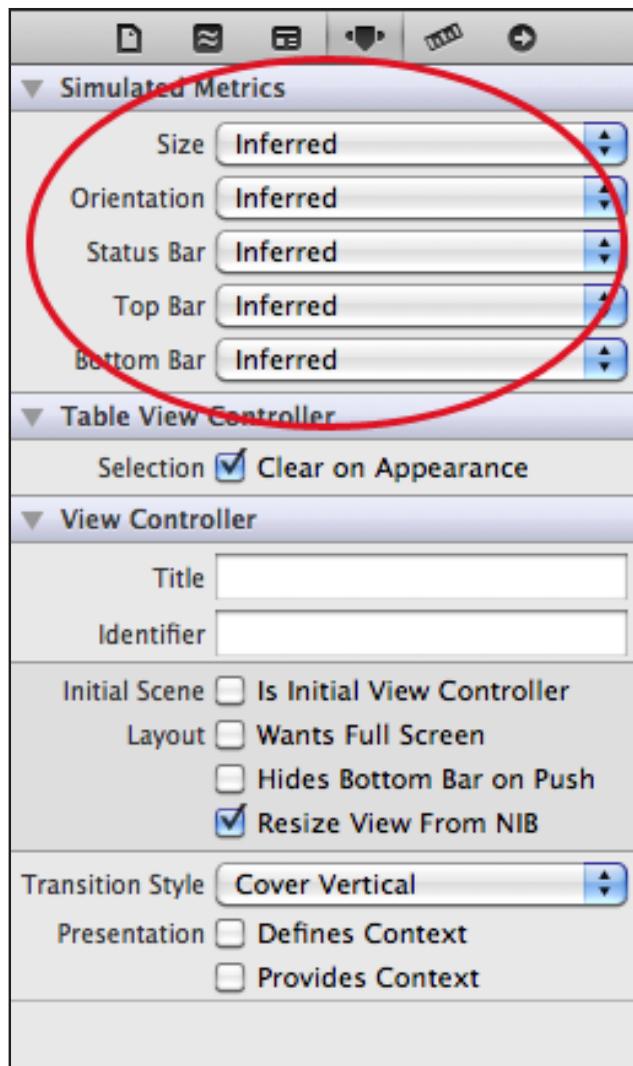
You could also have dragged in a Navigation Controller from the Object Library, but this Embed In command is just as easy.

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. The Storyboard Editor 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.

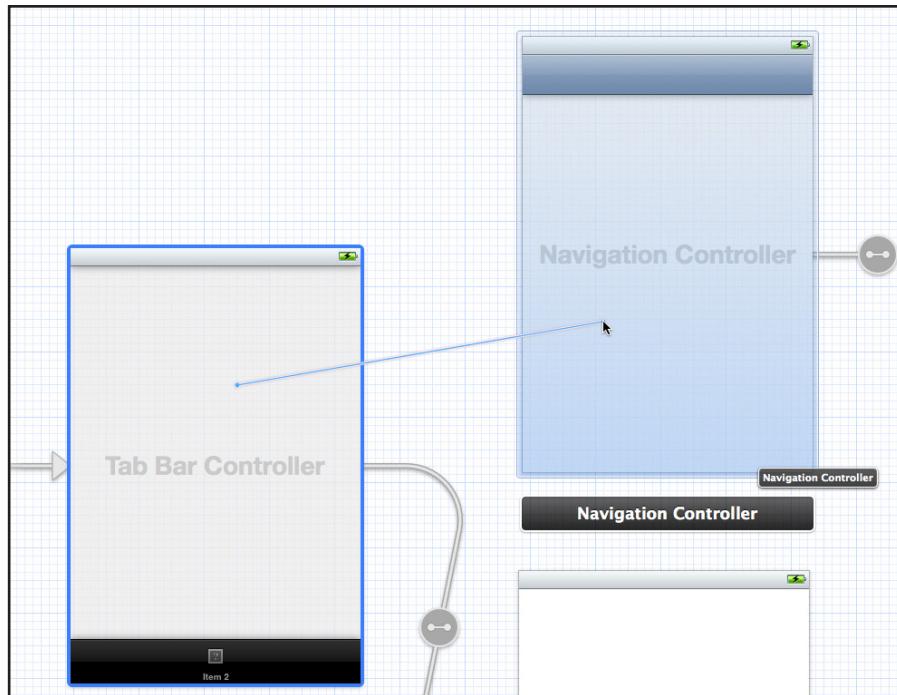
If you look at the Attributes Inspector for the Table View Controller, you'll see the Simulated metrics section at the top:



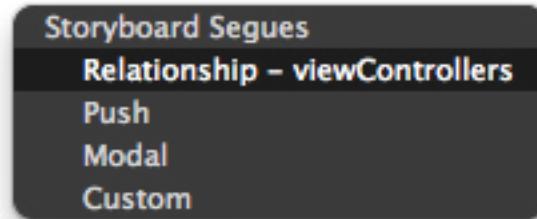
"Inferred" is the default setting for storyboards and it means the scene will show a navigation bar when it's inside of a navigation controller, a tab bar when it's inside of a tab bar controller, and so on. You could override these settings if you wanted to, but keep in mind they are here only to help 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 new scenes to our Tab Bar Controller. Ctrl-drag from the Tab Bar Controller to the Navigation Controller:

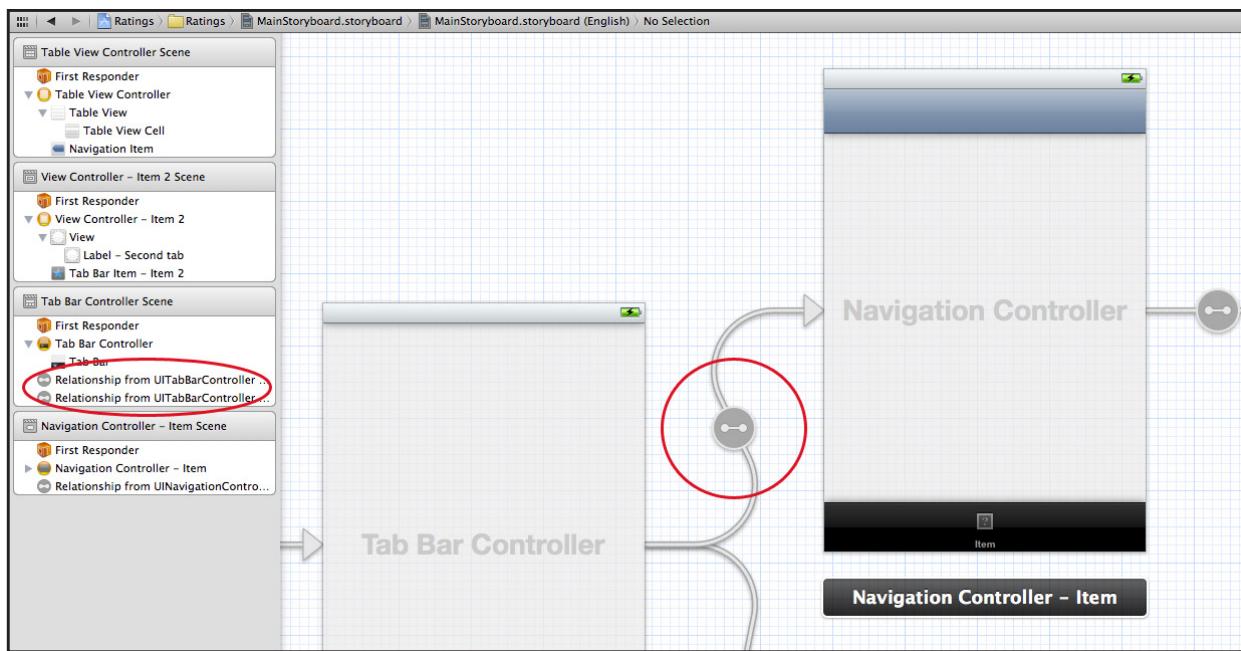




When you let go, a small popup menu appears:

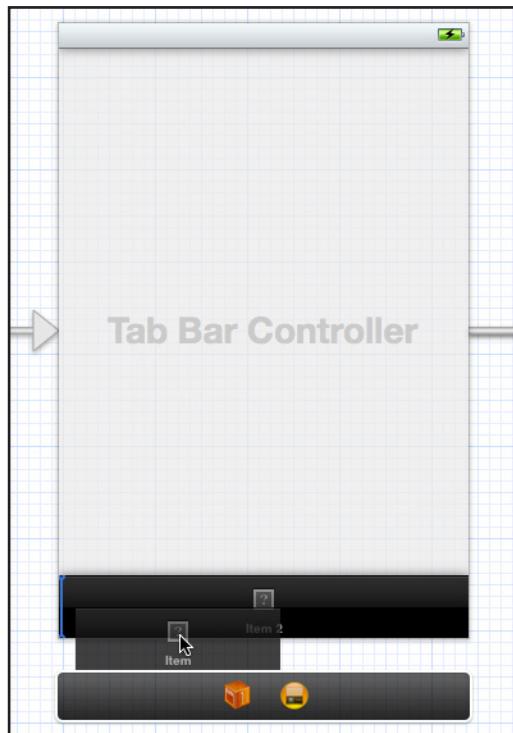


Choose the **Relationship - viewControllers** option. This creates a new relationship arrow between the two scenes:



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 we made this new connection, a new tab was added to the Tab Bar Controller, simply named "Item". I want this new scene to be the first tab, so drag the tabs around to change their order:



Run the app and try it out. The first tab now contains a table view inside a navigation controller.

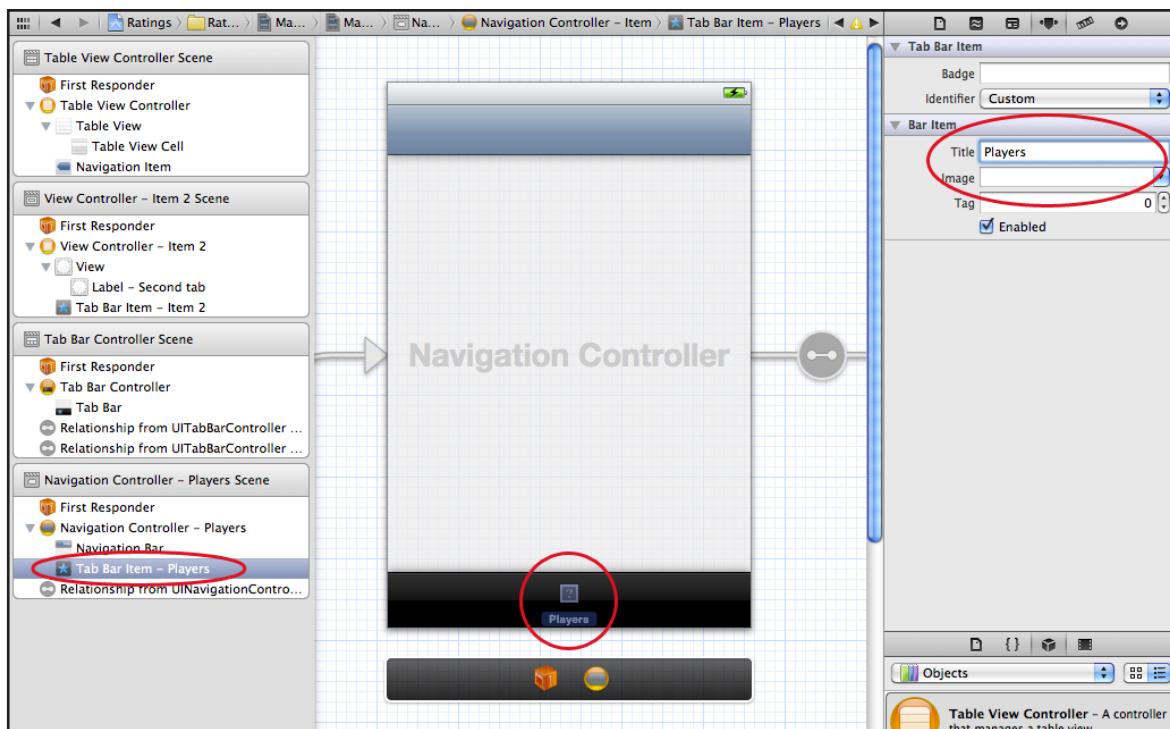


Before we put some actual functionality into this app, let's clean up the storyboard a little. I want to name the first tab "Players" and the second "Gestures". You do not change this on the Tab Bar Controller itself, but in the view controllers that are connected to these tabs.

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":





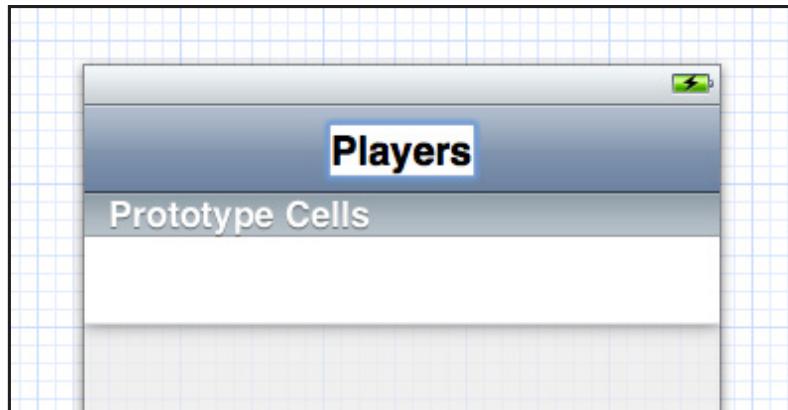
Rename the Tab Bar Item for the view controller from the second tab to "Gestures".

We should also put some pictures on these tabs. The resources for this chapter contains a subfolder named Images. Add that folder to the project.

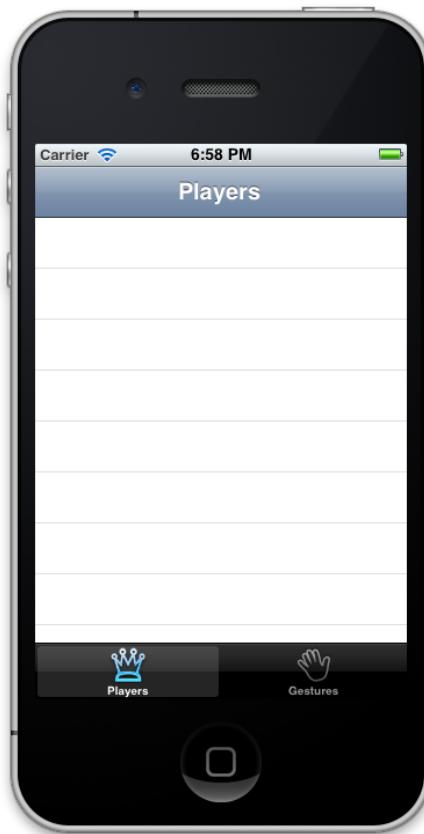
Then, in the Attributes Inspector for the Players Tab Bar Item, choose the Players.png image. You probably guessed it, but give the Gestures item the image Gestures.png.

Similarly, a view controller 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 simply 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.)



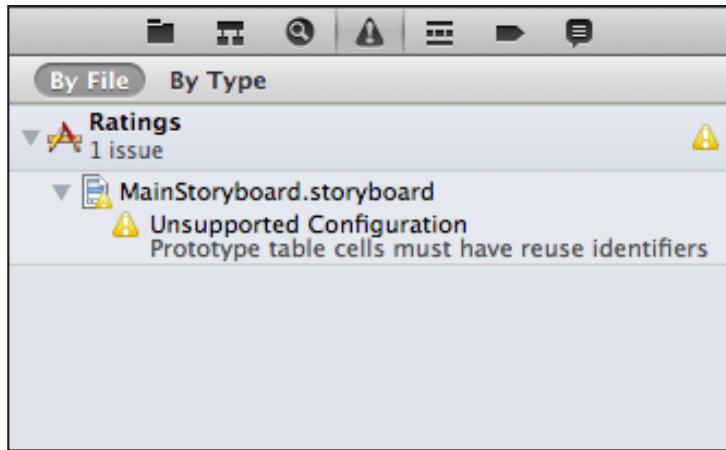
Run the app and marvel at our pretty tab bar, all without writing a single line of code!



Prototype cells

You may have noticed that ever since we added the Table View Controller, Xcode has been complaining:



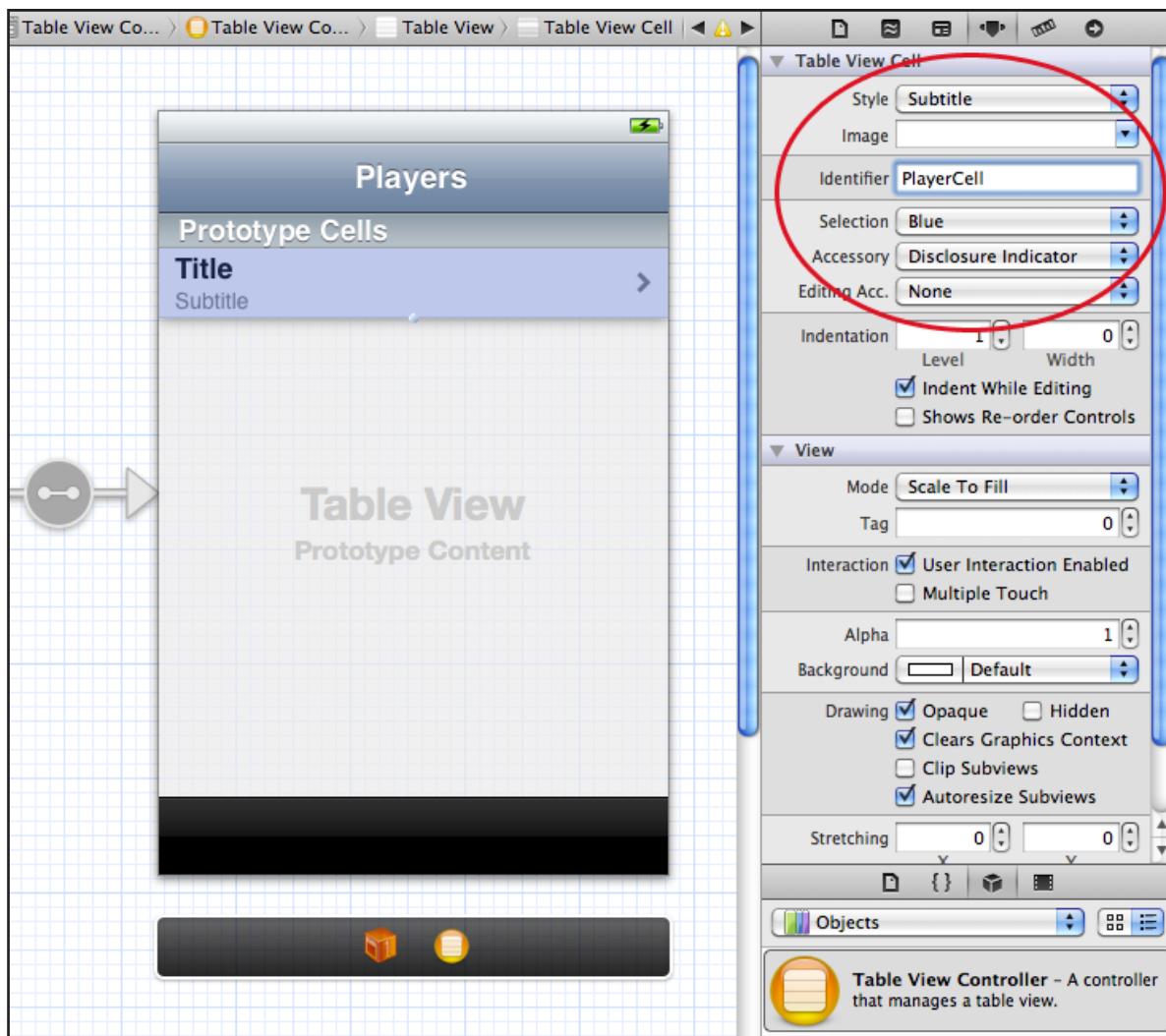


The warning message is, “Unsupported Configuration: Prototype table cells must have reuse identifiers”. When you add a Table View Controller to a storyboard, it wants to use prototype cells by default but we haven’t properly configured this yet, hence the warning.

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. iOS5 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.

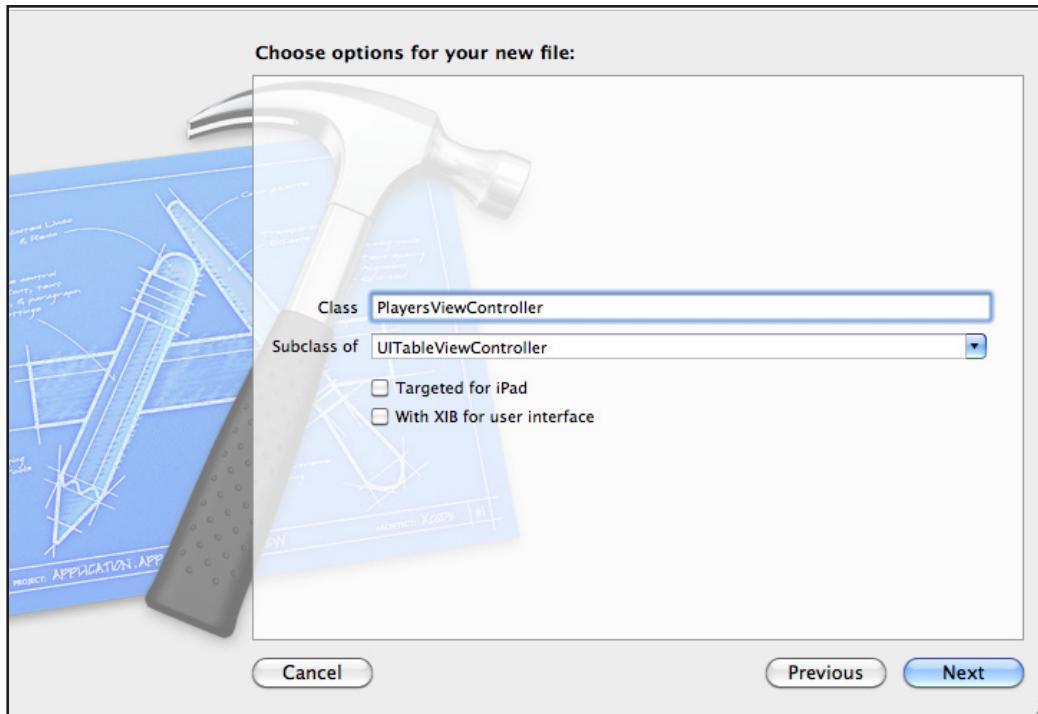
The Table View Controller comes with a blank prototype cell. Click on that cell to select it and in the Attributes Inspector set Style to Subtitle. This immediately changes the appearance of the cell to include two labels. If you’ve used table views before and created your own cells by hand, you may recognize this as the UITableViewCellStyleSubtitle style. With prototype cells you can either pick one of the built-in cell styles as we just did, or create your own custom design (which we’ll do shortly).



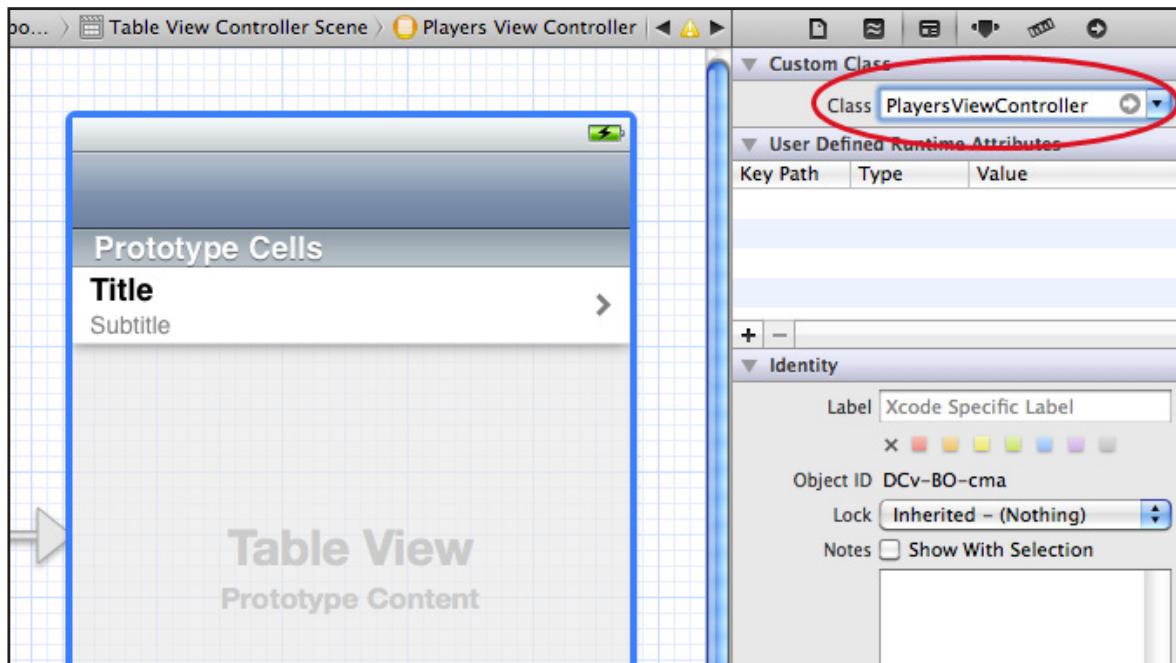
Set the Accessory attribute to Disclosure Indicator and give the cell the Reuse Identifier "PlayerCell". That will make Xcode shut up about the warning. All prototype cells are still regular UITableViewCells objects and therefore should have a reuse identifier. Xcode is just making sure we don't forget (at least for those of us who pay attention to its warnings).

Run the app, and... nothing has changed. That's not so strange, we 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 **UIViewController subclass** template. Name the class **PlayersViewController** and make it a subclass of **UITableViewController**. The **With XIB for user interface** option should be unchecked because we already have the design of this view controller in the storyboard. No nibs today!



Go back to the Storyboard Editor 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 our 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 our app. It contains Player objects. You also need to synthesize this in **PlayersViewController.m**:

```
@synthesize players;
```

Let's make that Player class right now. 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
```

And change **Player.m** to:

```
#import "Player.h"

@implementation Player

@synthesize name;
@synthesize game;
@synthesize 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 (1 to 5 stars).

We'll make the array and some test Player objects in our App Delegate and then assign it to the PlayersViewController's players property.

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;
}

// Rest of file...
```

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] objectAtIndex:0];
    PlayersViewController *playersViewController =
        [[navigationController viewControllers] objectAtIndex: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] objectAtIndex:0];
PlayersViewController *playersViewController =
    [[navigationController viewControllers] objectAtIndex:0];
```



```
playersViewController.players = players;
```

Yikes, what is that?! We 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 I find annoying. With Interface Builder 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 we can always get those references programmatically.

```
UITabBarController *tabBarController = (UITabBarController *)  
    self.window.rootViewController;
```

We know that the storyboard's initial view controller is a Tab Bar Controller, so we can look up the window's rootViewController and cast it.

The PlayersViewController sits inside a navigation controller in the first tab, so we look up that UINavigationController object:

```
UINavigationController *navigationController = [[tabBarController  
    viewControllers] objectAtIndex:0];
```

and then ask it for its root view controller, which is the PlayersViewController that we are looking for:

```
PlayersViewController *playersViewController =  
    [[navigationController viewControllers] objectAtIndex:0];
```

Unfortunately, UINavigationController has no rootViewController property so we'll have to delve into its viewControllers array. (It does have a topViewController property but that points to the top-most controller on the stack and we're looking for the bottom-most one. At this point the app has just launched so technically we could have used topViewController, but that is not always the case.)

Now that we have an array full of Player objects, we can continue building the data source for PlayersViewController.

Open up **PlayersViewController.m**, and change the table view data source methods to the following:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView  
{  
    return 1;  
}
```



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

The real work happens in `cellForRowAtIndexPath`. The version from the Xcode template looks 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;
}
```

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 objectAtIndex: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 re-



use identifier that you set on the prototype cell in the storyboard editor, in our case "PlayerCell". Don't forget to set that identifier, or this little scheme won't work!

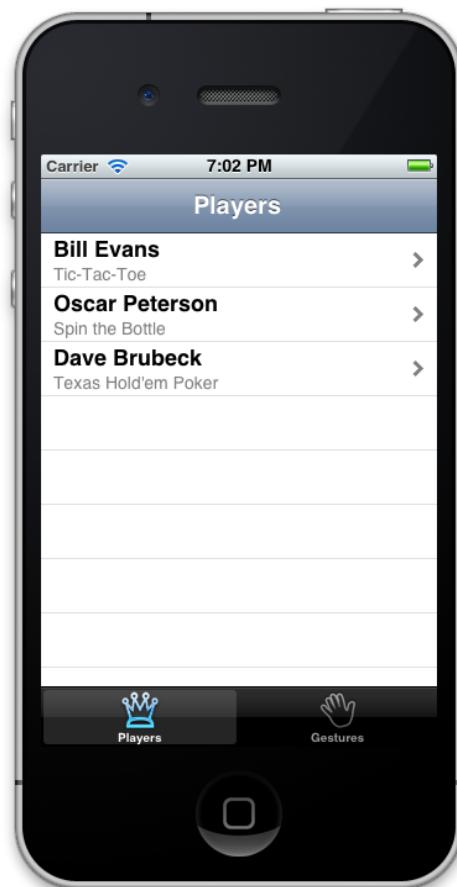
Because this class doesn't know anything about the Player object yet, it needs an #import at the top of the file:

```
#import "Player.h"
```

And we should not forget to synthesize the property that we added earlier:

```
@synthesize players;
```

Now you can run the app, and lo and behold, the table view has players in it:



Note: In this app we'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.



It just takes one line of code to use these newfangled prototype cells. I think that's just great!

Designing Our Own Prototype Cells

Using a standard cell style is OK for many apps, but I want to add an image on the right-hand side of the cell that shows the player's rating (in stars). Having an image view in that spot is not supported by the standard cell styles, so we'll have to make a custom design.

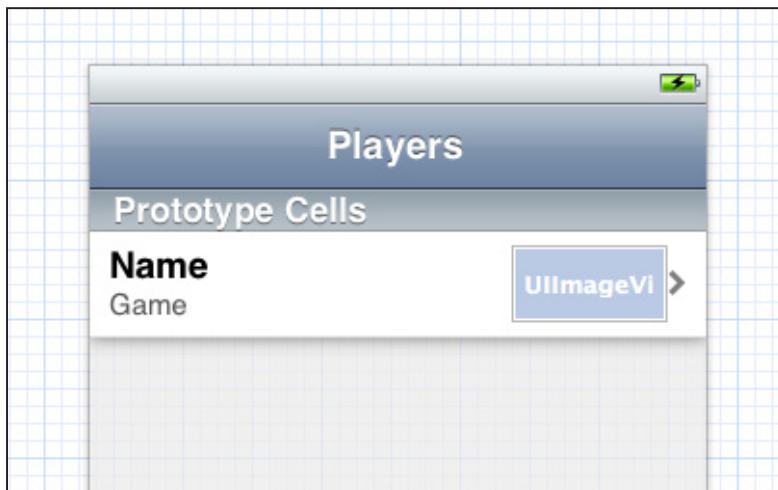
Switch back to MainStoryboard.storyboard, select the prototype cell in the table view, and set its Style attribute to Custom. The default labels now disappear.

First make the cell a little taller. Either drag its handle at the bottom or change the Row Height value in the Size Inspector. I used the latter method to make the cell 55 points high.

Drag two Label objects from the Objects Library into the cell and place them roughly where the labels were previously. Just play with the font and colors and pick something you like. Do 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 we put into this view is not stretched.

I made the labels 210 points wide so they don't overlap with the image view. The final design for the prototype cell looks something like this:



Because this is a custom designed cell, we 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 our cell anymore. Instead, we 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` from `PlayersViewController` to:

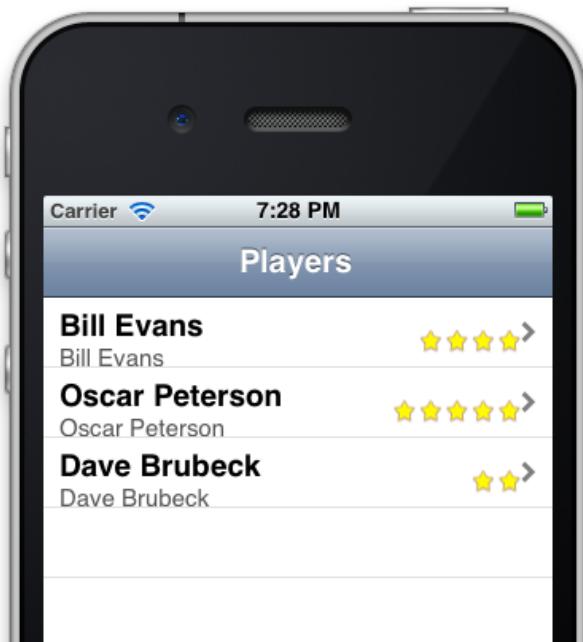
```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"PlayerCell"];
    Player *player = [self.players objectAtIndex:indexPath.row];
    UILabel *nameLabel = (UILabel *)[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`. Add that method above `cellForRowAtIndexPath`:

```
- (UIImage *)imageForRating:(int)rating
{
    switch (rating)
    {
        case 1: return [UIImage imageNamed:@"1StarSmall.png"];
        case 2: return [UIImage imageNamed:@"2StarsSmall.png"];
        case 3: return [UIImage imageNamed:@"3StarsSmall.png"];
        case 4: return [UIImage imageNamed:@"4StarsSmall.png"];
        case 5: return [UIImage imageNamed:@"5StarsSmall.png"];
    }
    return nil;
}
```

That should do it. Now run the app again.

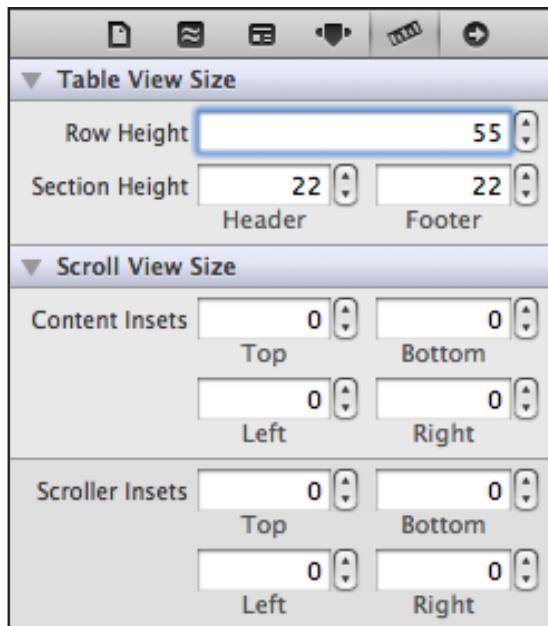




Hmm, that doesn't look quite right. We did change the height of the prototype cell but the table view doesn't automatically take that into consideration. There are two ways to fix it: we can change the table view's Row Height attribute or implement the `heightForRowAtIndexPath` method. The former is much easier, so let's do that.

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.

Back in **MainStoryboard.storyboard**, in the Size Inspector of the Table View, set Row Height to 55:



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 too. So it may have worked correctly for you the first time around.

If you run the app now, it looks a lot better!

Using a Subclass for the Prototype Cell

Our 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 we could connect these labels to outlets and then use the corresponding properties. As it turns out, we 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, strong) IBOutlet UILabel *nameLabel;
@property (nonatomic, strong) IBOutlet UILabel *gameLabel;
@property (nonatomic, strong) IBOutlet UIImageView
    *ratingImageView;

@end
```



Replace the contents of **PlayerCell.m** with:

```
#import "PlayerCell.h"

@implementation PlayerCell

@synthesize nameLabel;
@synthesize gameLabel;
@synthesize ratingImageView;

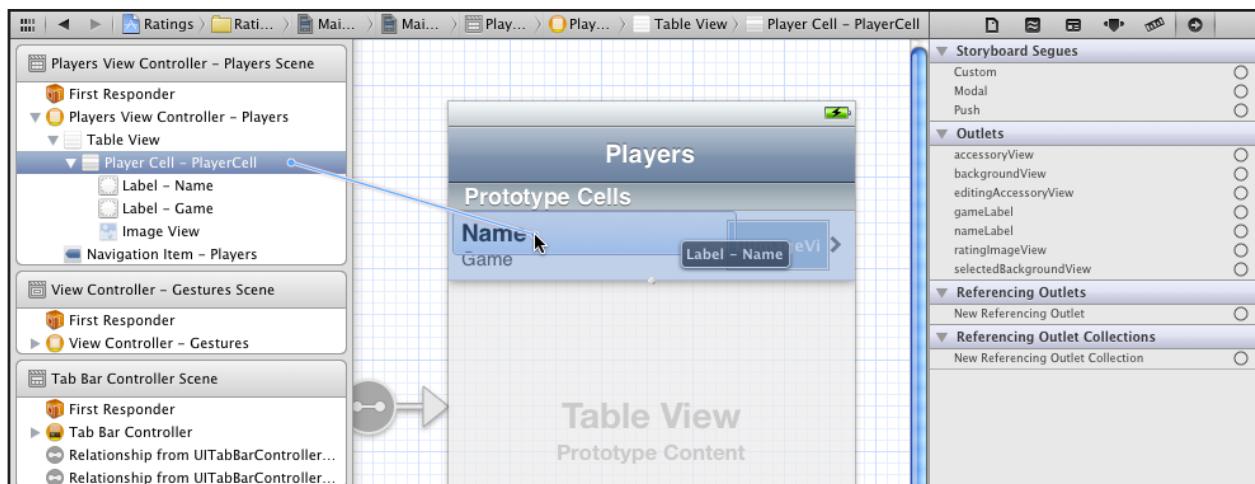
@end
```

The class itself doesn't do much, it just adds properties for nameLabel, gameLabel and ratingImageView.

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 UITableViewCells.

Note that I gave this class the same name as the reuse identifier -- they're both called PlayerCell -- but that's only because 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 you can 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.)

Now that we've hooked up the properties, we can simplify our data source code one more time:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    PlayerCell *cell = (PlayerCell *)[tableView
        dequeueReusableCellWithIdentifier:@"PlayerCell"];
    Player *player = [self.players objectAtIndex: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. We now cast the object that we receive from dequeueReusableCellWithIdentifier to a PlayerCell, and then we can simply use the properties that are wired up to the labels and the image view. I really like how using prototype cells makes table views a whole lot less messy!

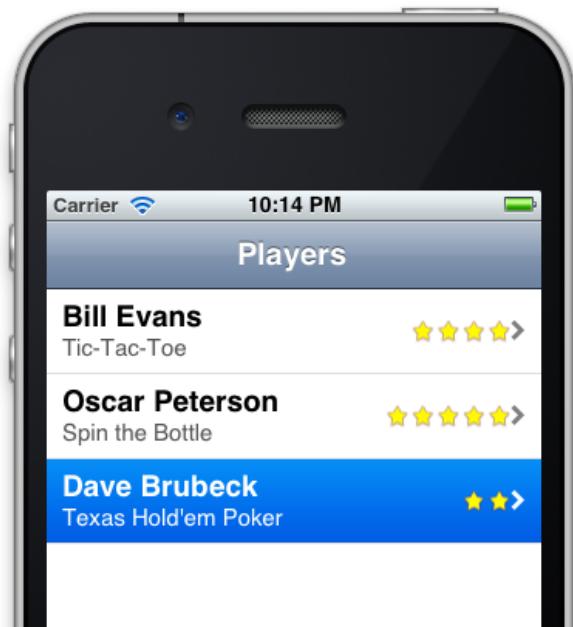
You'll need to import the PlayerCell class to make this work:

```
#import "PlayerCell.h"
```

Run the app and try it out. When you run the app it should still look the same as before, but behind the scenes we're now using our 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 of the labels so that they look good when the user taps the row:



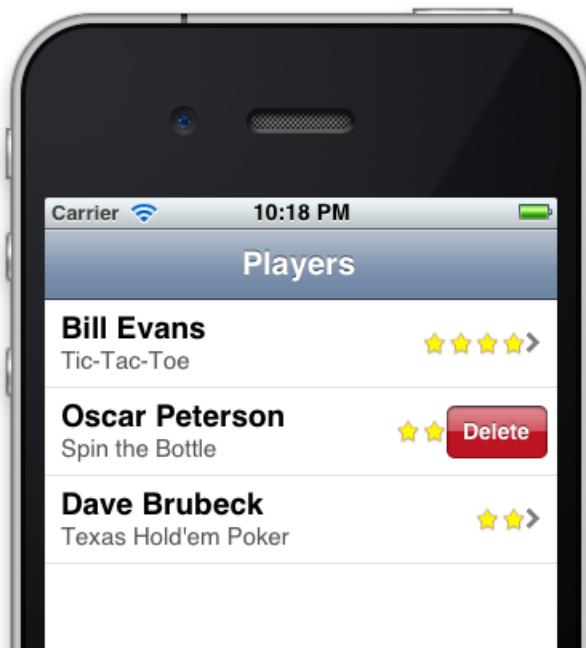


Second, you should make sure that the content you add is flexible so that when the table view cell resizes, the content sizes along with it. Cells will resize when you add the ability to delete or move rows, for example.

Add the following method to **PlayersViewController.m**:

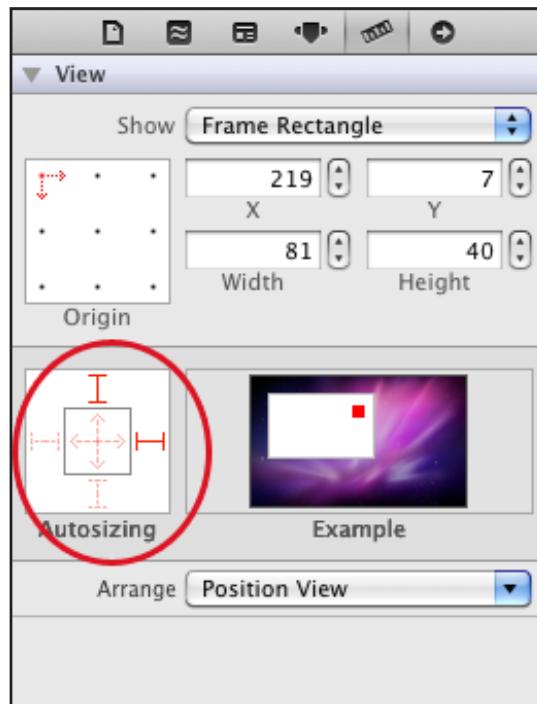
```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        [self.players removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:
         [NSArray arrayWithObject: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.

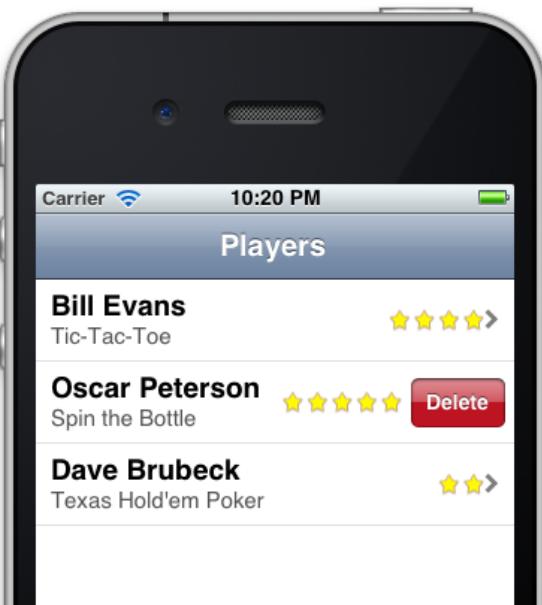
To fix this, open **MainStoryboard.storyboard**, select the image view in the table view cell, and in the Size Inspector change the Autosizing so it sticks to its super-view's right edge:



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 our storyboard. We're going to create a screen that allows users to add new players to the app.

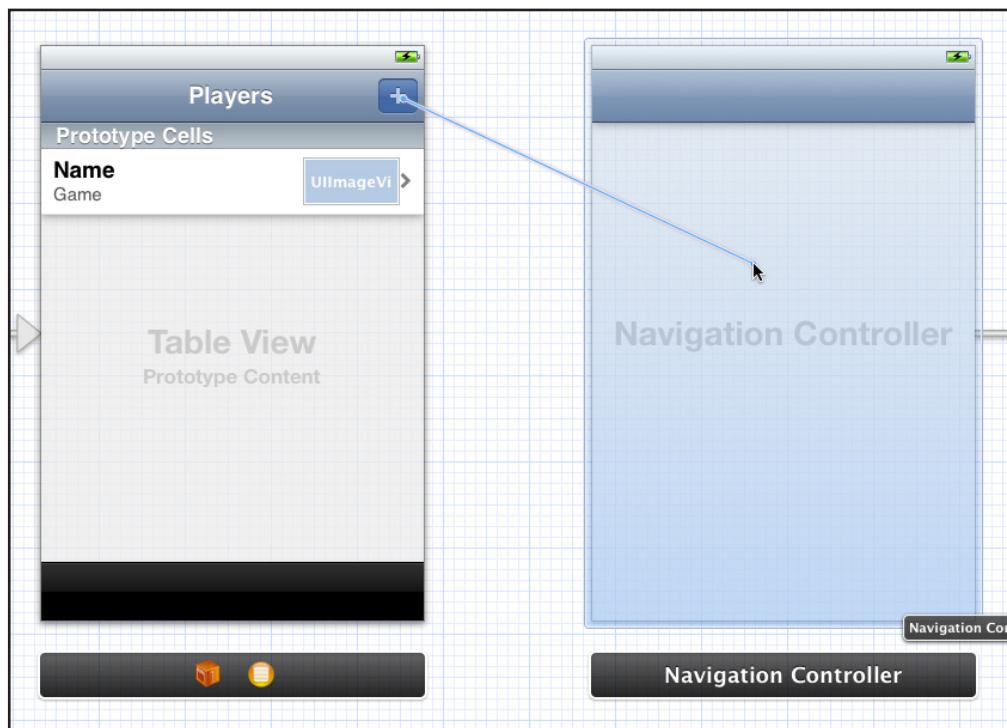
Drag a Bar Button Item into the right slot of the navigation bar on the Players screen. In the Attributes Inspector change its Identifier to Add to make it a standard + button. When you tap this button we'll make a new modal screen pop up that lets you enter the details for a new player.



Drag a new Table View 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.

Keep the new Table View Controller selected and embed it in a Navigation Controller (in case you forgot, from the menubar pick **Editor\Embed In\Navigation Controller**).

Here's the trick: Select the + button that we 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:

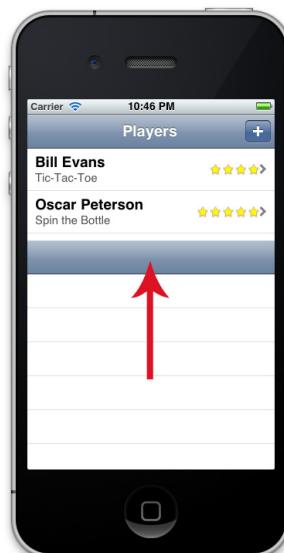




This type of connection is known as a segue (pronounce: seg-way) and represents a transition from one screen to another. The connections we had so far were relationships and they described one view controller containing another. A segue, on the other hand, changes what is on the screen. They are triggered 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 we just did, dragging from the Bar Button Item to the next screen, is enough to create the transition. (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!



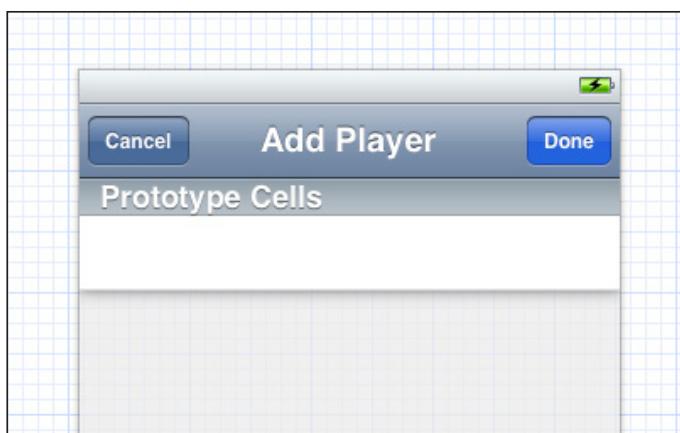
This is a so-called “modal” segue. The new screen completely obscures the previous one. The user cannot interact with the underlying screen until they close the modal screen first. Later on we’ll also see “push” segues that push new screens on the navigation stack.

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, we have to use the delegate pattern. For that, we first have to give this new scene its own class. Add a new **UITableViewController** subclass to the project and name it **PlayerDetailsViewController**.

To hook it up to the storyboard, switch back to **MainStoryboard.storyboard**, 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 we’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 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
```



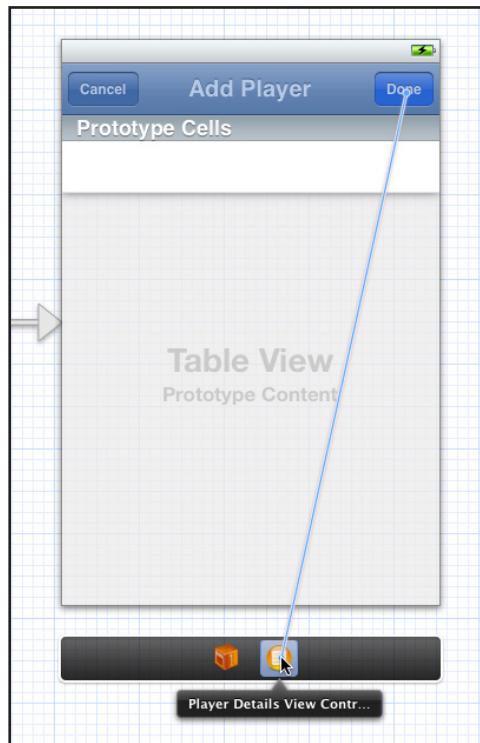
```
@property (nonatomic, weak) id <PlayerDetailsViewControllerDelegate>
delegate;

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

@end
```

This defines a new delegate protocol that we'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 the Storyboard Editor, 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:



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 that 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 PlayerDetailsViewController. That way the delegate always knows which object sent the message.

Don't forget to synthesize the delegate property:

```
@synthesize delegate;
```

Now that we've given the PlayerDetailsViewController a delegate protocol, we 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 we'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 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 the Storyboard Editor just by dragging a line between the two. Unfortunately, that is not possible. To pass data to the new view controller during a segue, we still need to write some code.

Add the following method to PlayersViewController (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]
             objectAtIndex: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 we can use this opportunity to send data to it. (You never call prepareForSegue yourself, it's a message from UIKit to let you know that a segue has just been triggered.)

Note that the destination of the segue is the Navigation Controller, because that is what we connected to the Bar Button Item. To get the PlayerDetailsViewController instance, we have to dig through the Navigation Controller's viewControllers property to find it.

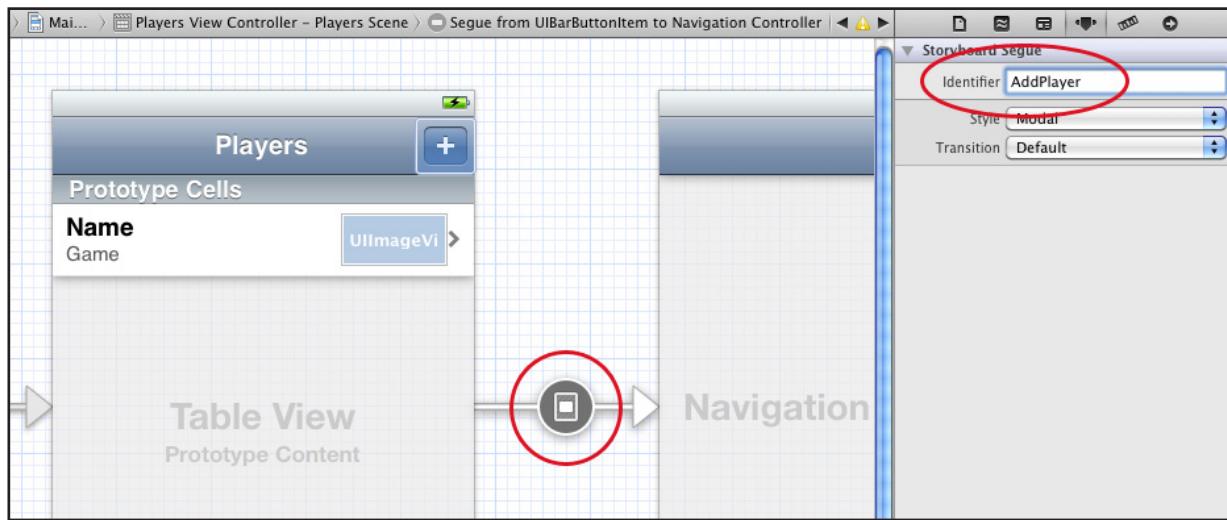
Run the app, press the + button, and try to close the Add Player screen. It still doesn't work!

That's because we 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 we'll do later in this tutorial).

To fix this issue, go into the Storyboard Editor 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":

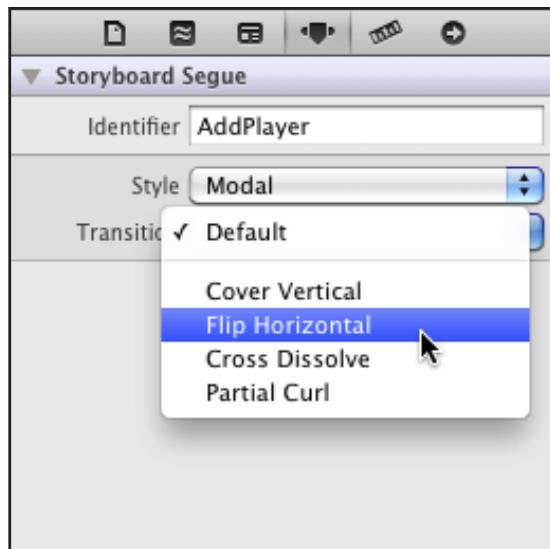


If you run the app again, tapping Cancel or Done will now properly close the screen and return you to the list of players.

Note: It is perfectly possible to call `dismissViewControllerAnimated:completion:` from the modal screen. There is no requirement that says this must be done by the delegate. I personally prefer to let the delegate do 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. 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!

We'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](#):

1. 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.)
2. Give the segue a unique Identifier. (It only has to be unique in the source scene; different scenes can use the same identifier.)
3. Create a delegate protocol for the destination scene.
4. 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.
5. Make the source scene implement the delegate protocol. It should dismiss the destination view controller when Cancel or Done is pressed.
6. Implement `prepareForSegue` in the source view controller and do `destination.delegate = self;`.

Delegates are necessary because 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 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.

Static Cells

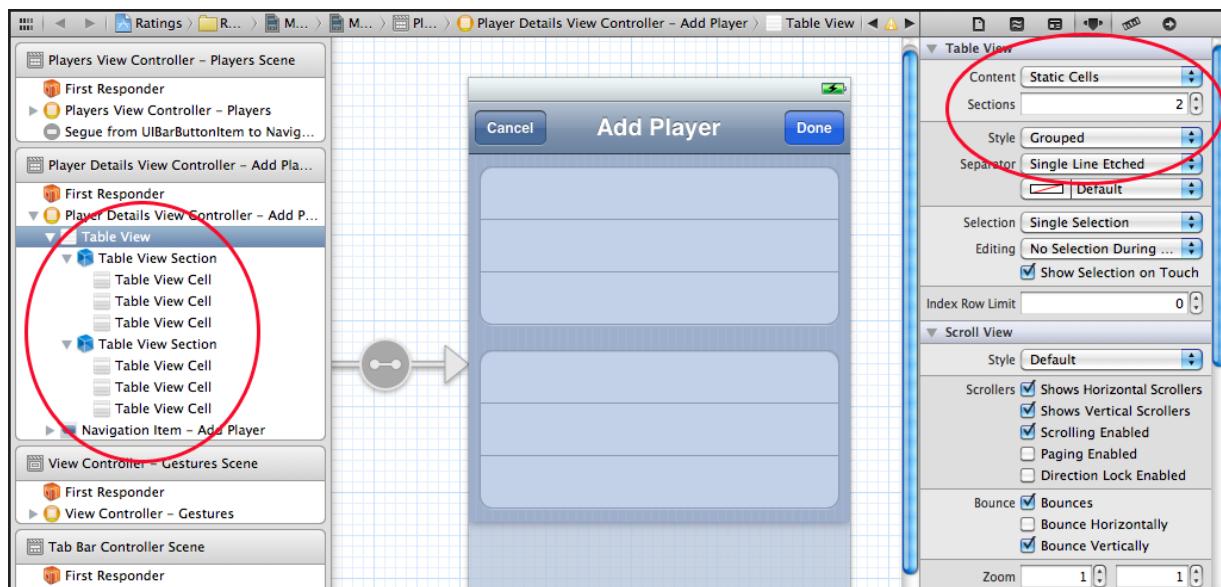
When we're finished, the Add Player screen will look like this:



That's a grouped table view, of course, but the new thing is that we don't have to create a data source for this table. We can design it directly in the Storyboard Editor, 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.

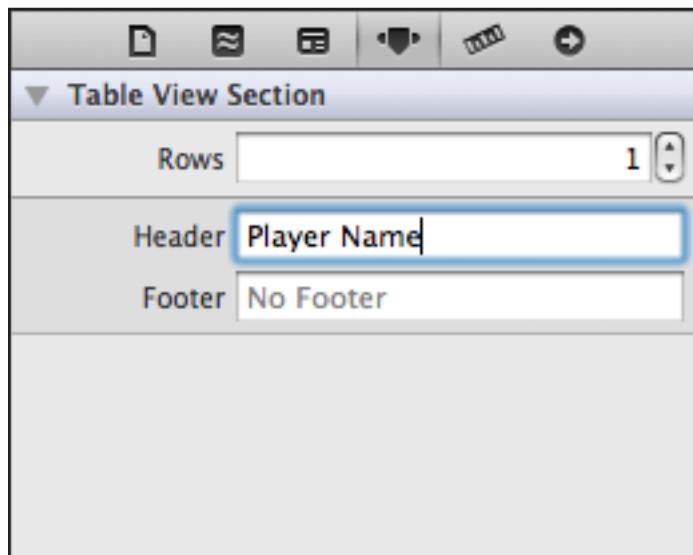




When you change the value of the Sections attribute, the editor will clone the existing section. (You can also select a specific section in the Document Outline on the left and duplicate it.)

Our 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".

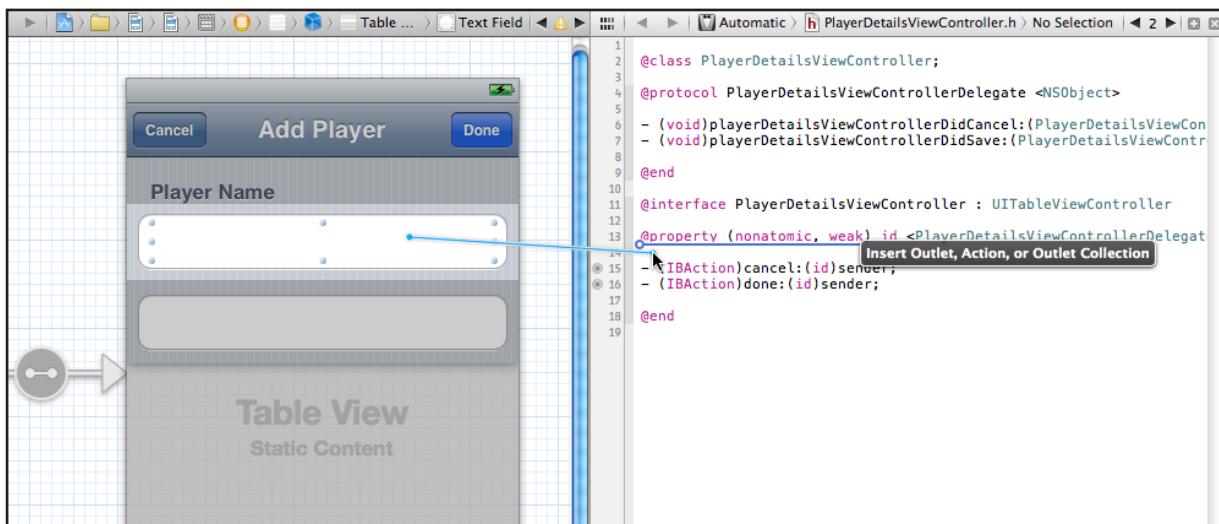


Drag a new Text Field into the cell for this section. Remove its border so you can't see where the text field begins or ends. Set the Font to System 17 and uncheck Adjust to Fit.

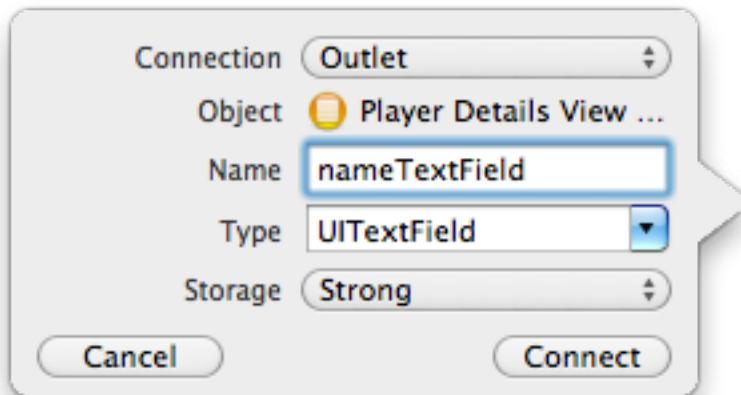


We're going to make an outlet for this text field on the PlayerDetailsViewController using the Assistant Editor feature of Xcode. Open the Assistant Editor with the button from the toolbar (the one that looks like a tuxedo / alien face). It should automatically open on PlayerDetailsViewController.h.

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`. After you click Connect, Xcode will add the following property to **PlayerDetailsViewController.h**:

```
@property (strong, nonatomic) IBOutlet UITextField *nameTextField;
```

It has also automatically synthesized this property and added it to the `viewDidUnload` method in the .m file.

This 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 us 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.

The final design of the Add Player screen looks like this:



When you use static cells, your table view controller doesn't need a data source. Because we used an Xcode template to create our PlayerDetailsViewController class, it still has some placeholder code for the data source, so let's remove that code now that we have no use for it. Delete anything between the

```
#pragma mark - Table view data source
```

and

```
#pragma mark - Table view delegate
```

lines. That should silence Xcode about the warnings it has been giving us ever since we added this class.

Run the app and check out our new screen with the static cells. All without writing a line of code -- in fact, we threw away a bunch of code!

We can't avoid writing code altogether, though. When you added the text field to 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, we should let a tap anywhere in that row bring up the keyboard. That's pretty easy to do, just add 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 we activate the text field (there is only one cell in the section so we'll just test for the section index). This 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 we'll 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, we should create a new Player object and fill in its properties. Then we should tell the delegate that we've added a new player, so it can update its own screen.



So inside **PlayerDetailsViewController.m**, 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];
}
```

This requires an import for Player:

```
#import "Player.h"
```

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, we now have "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:
        [NSArray arrayWithObject: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. We 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 our initial view controller is a Tab Bar Controller, the two view controllers that it contains are also loaded (the Players scene 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 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 the following methods to **PlayerDetailsViewController.m**:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        NSLog(@"init PlayerDetailsViewController");
    }
    return self;
}

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

We're overriding the initWithCoder and dealloc methods and making them log a message to the Debug Area. 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. The Storyboard Editor will let you add them to a Table View object inside a regular



UIViewController, but 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 regular view controller. Neither work in Interface Builder, 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 here [<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 the Storyboard Editor with the scroll gesture on the mouse or trackpad (2 finger swipe). This might not be immediately obvious, but it works quite well.

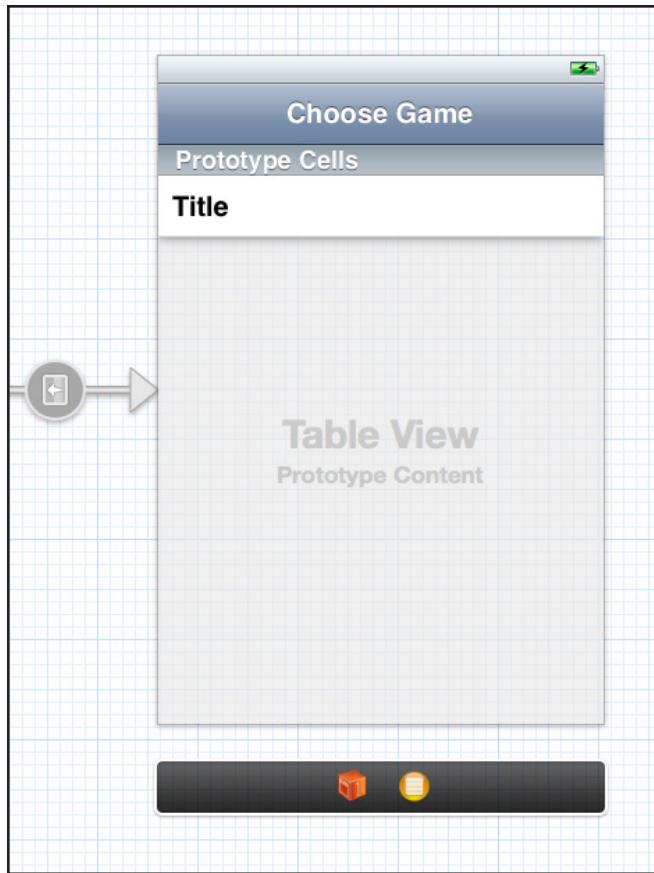
The Game Picker Screen

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 we'll be adding yet another table view controller, although this time we'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 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 we need to do for the design of this screen:





Add a new **UITableViewController** subclass file to the project and name it **GamePickerController**. Don't forget to set the Class of the Table View Controller in the storyboard so that your new GamePickerController is associated with the Table View Controller in the storyboard.

First we shall give this new screen some data to display. Add a new instance variable to **GamePickerController.h**:

```
@interface GamePickerController : UITableViewController {  
    NSArray * games;  
}
```

Then switch to **GamePickerController.m**, and fill up this array in `viewDidLoad`:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
    games = [NSArray arrayWithObjects:  
        @"Angry Birds",  
        @"Chess",  
        @"Russian Roulette",  
        @"Spin the Bottle",  
        nil];
```



```
        @"Texas Hold'em Poker",
        @"Tic-Tac-Toe",
        nil];
}
```

Because we create this array in viewDidLoad, we have to release it in viewDidUnload:

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

Even though viewDidUnload will never actually be called on this screen (we never obscure it with another view), it's still good practice to always balance your allocations with releases.

Replace the data source methods from the template with:

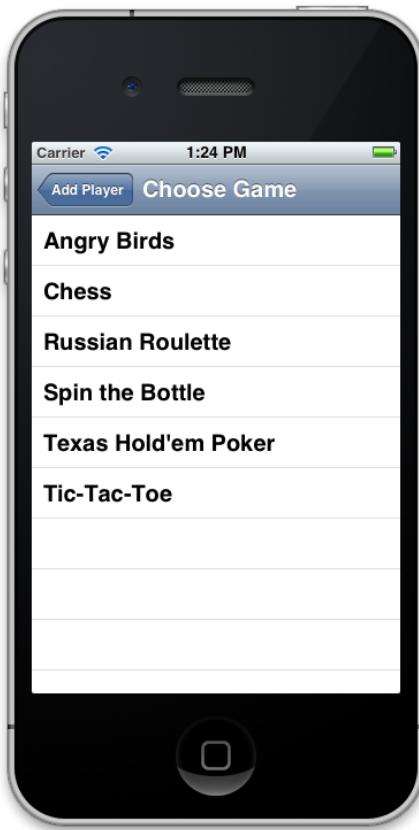
```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (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 objectAtIndex: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? We didn't have to write any code to invoke this new screen. We just dragged from the static table view cell to the new scene and that was it. (Note that 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 we'll have to add a new delegate for that. Add the following to **GamePickerController.h**:

```
@class GamePickerController;

@protocol GamePickerControllerDelegate <NSObject>
- (void)gamePickerController:
    (GamePickerController *)controller
    didSelectGame:(NSString *)theGame;
@end

@interface GamePickerController : UITableViewController

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

```
@end
```

We've added a delegate protocol with just one method, and a property that will hold the name of the currently selected game.

Change the top of **GamePickerController.m** to:

```
@implementation GamePickerController
{
    NSArray *games;
    NSUInteger selectedIndex;
}
@synthesize delegate;
@synthesize game;
```

This adds a new ivar, `selectedIndex`, and synthesizes the properties.

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 we figure out what the index is for that game in the list of games. We'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 we'll do this in the caller's `prepareForSegue`, which takes place before `viewDidLoad`.

Change `cellForRowIndexPath` to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"GameCell"];
    cell.textLabel.text = [games objectAtIndex: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. I'm sure this small gesture will be appreciated by the users of the app.

Replace the placeholder `didSelectRowAtIndexPath` method from the template with:

```
- (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 *theGame = [games objectAtIndex:indexPath.row];
    [self.delegate gamePickerController:self
        didSelectGame:theGame];
}

```

First we deselect the row after it was tapped. That makes it fade from the blue highlight color back to the regular white. Then we remove the checkmark from the cell that was previously selected, and put it on the row that was just tapped. Finally, we return 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 we 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 we 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;
}
```

We use this ivar to remember the selected game so we can store it in the Player object later. We should give it 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 *)theGame
{
    game = theGame;
```



```
    self.detailLabel.text = game;
    [self.navigationController popViewControllerAnimated:YES];
}
```

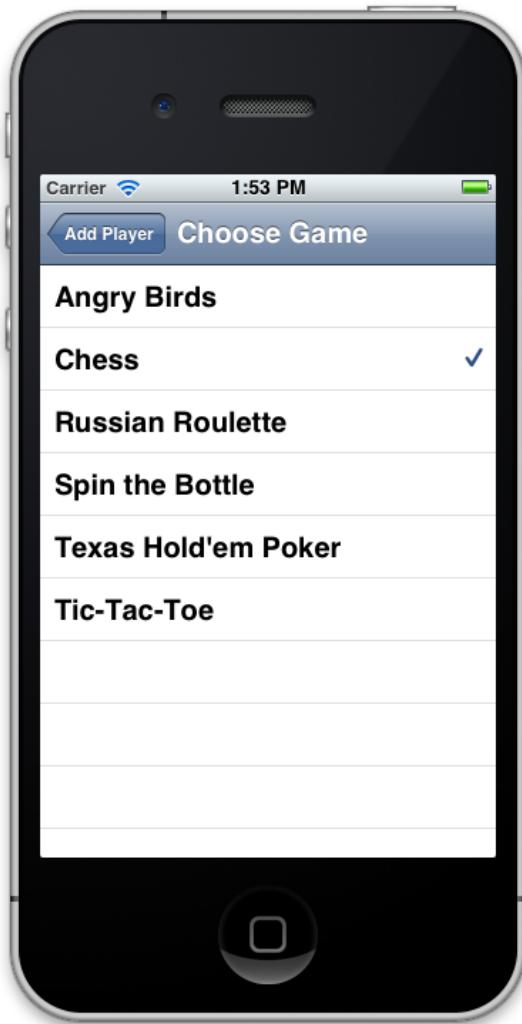
This is pretty straightforward, we put the name of the new game into our game ivar and the cell's label, and then we close the Choose Game screen. Because it's a push segue, we have to pop this screen off the navigation stack to close it.

Our done method can now put the name of the chosen game into the new Player object:

```
- (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, we now have a functioning Choose Game screen!





Where To Go From Here?

Congratulations, you now know the basics of using the Storyboard Editor, and can create apps with multiple view controllers transitioning between each other with segues!

If you want to learn more about Storyboards in iOS5, keep reading the next tutorial, 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.



Intermediate Storyboards

by Matthijs Hollemans

In the last chapter, you got some basic experience with Storyboards. 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, we're going to show you 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, add custom segues, use Storyboards on the iPad, and much more!

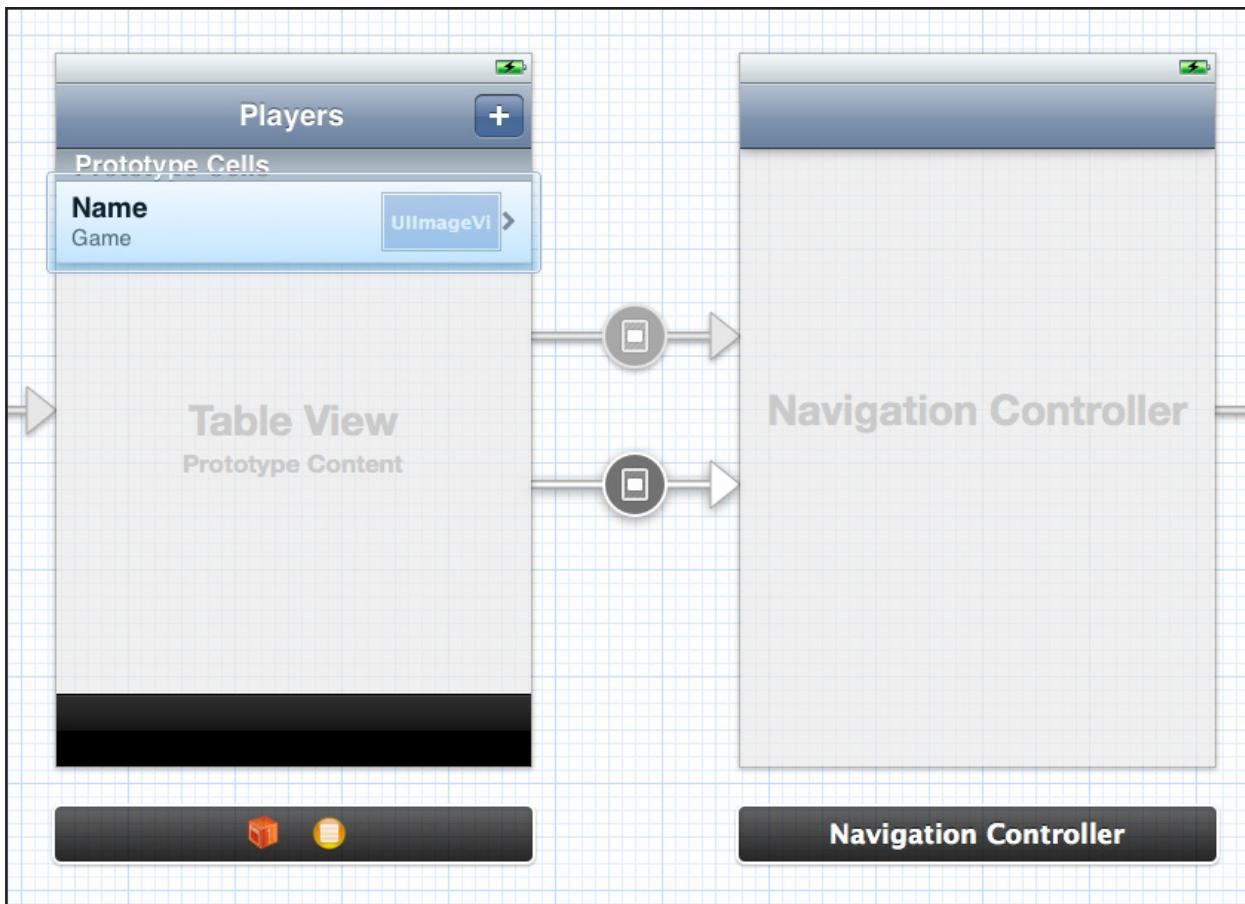
This chapter picks up where we left off last time, open up your Ratings project in Xcode and let's get started!

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 we 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:





We can distinguish between these two segues because we'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.

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] objectAtIndex:0];
        playerDetailsViewController.delegate = self;
    }
    else if ([segue.identifier isEqualToString:@"EditPlayer"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        PlayerDetailsViewController *playerDetailsViewController =
            [[navigationController viewControllers] objectAtIndex:0];
        playerDetailsViewController.delegate = self;
    }
}
```



```

        [[navigationController viewControllers] objectAtIndex:0];
playerDetailsViewController.delegate = self;

NSIndexPath *indexPath =
    [self.tableView indexPathForCell:sender];
Player *player = [self.players objectAtIndex: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 we now pass along a Player object in the new playerToEdit property.

To find the index-path for the cell that was tapped, we 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. We put the segue on the prototype cell, which means it can be triggered from any cell that is copied from the prototype. The storyboard magically takes care of this behind the scenes.

Add the new playerToEdit property to **PlayerDetailsViewController.h**:

```

@property (strong, nonatomic) Player *playerToEdit;

```

And to **PlayerDetailsViewController.m**:

```

@synthesize playerToEdit;

```

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. We also fill 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, we're not quite done with our changes. If you were to press Done now, a new Player object would still be added to the list. We have to change that part of the logic to update the existing Player object instead.

First, 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 we're 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:
        [NSArray arrayWithObject: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 we were able to re-use our 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, you are interacting with separate instances of this class.

I have mentioned a few times that `prepareForSegue` is called before `viewDidLoad`. We use that to our advantage to set the `playerToEdit` property on the destination view controller. In `viewDidLoad` we get the values from `playerToEdit` and put them into our labels.



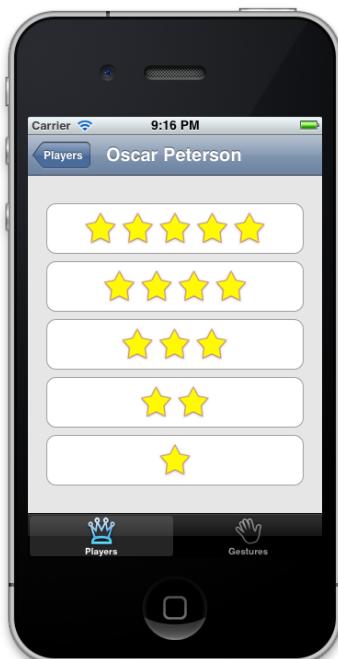
Note: There is no "didPerformFromSegue" method on the destination scene that lets the scene know it was invoked by a segue. In fact, that view controller doesn't know anything about the segue at all. To tell the destination view controller that it was launched from a segue, or to perform additional configuration, you'll need to set a property or call a method from prepareForSegue. You could 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 we haven't done much with those ratings except show a few stars here and there. We will now add a new screen that lets you pick a rating for a player:



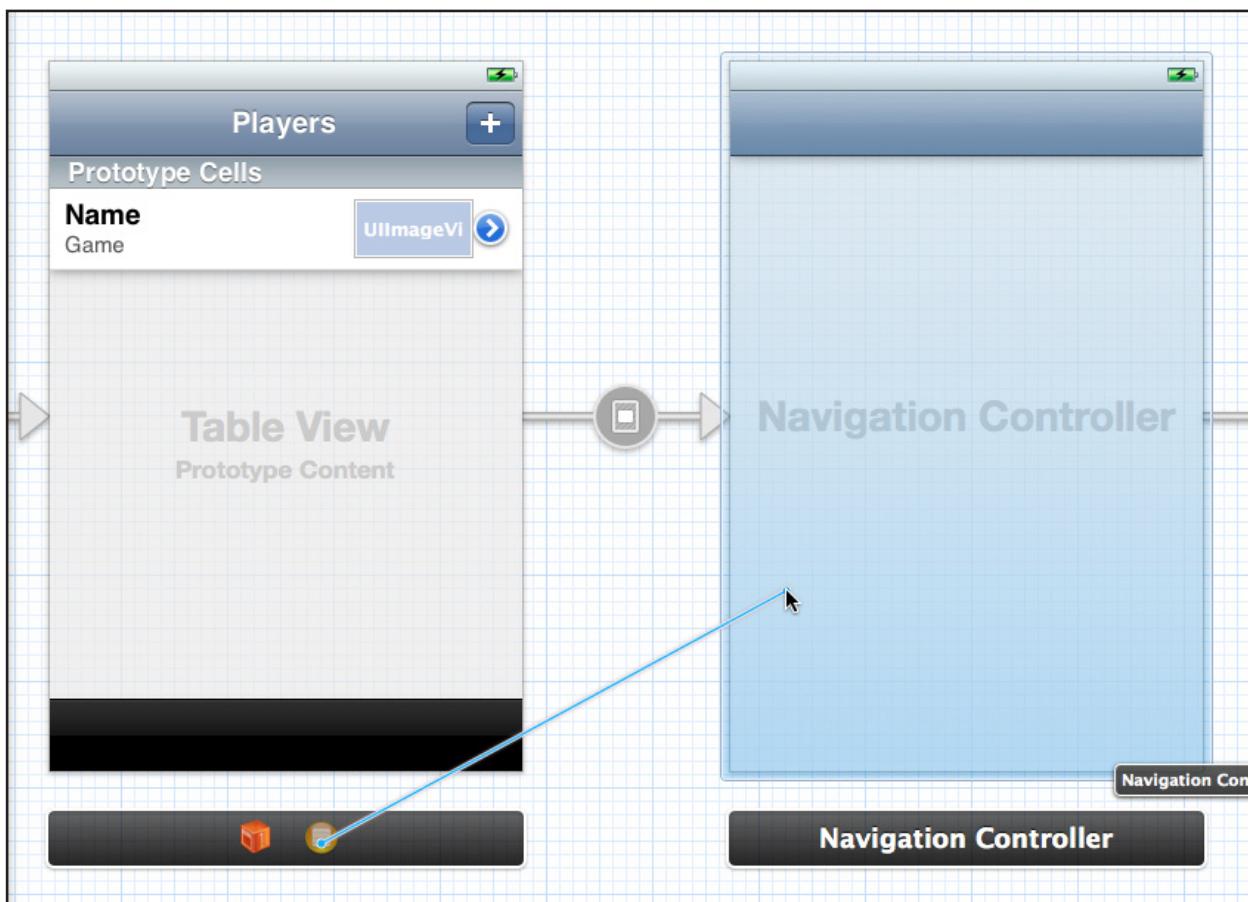
Drag a new View Controller into the canvas and put it roughly below the Add Player screen. This is a regular view controller, not a table view controller.

There is a bit of a problem, we 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 we'll have to find a way to distinguish between rating a player and editing a player. The way we'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 EditPlayer segue. We should make a new segue from the detail disclosure button to the Add/Edit Player screen, but unfortunately the storyboard editor does not support this. What we'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. Name it "EditPlayer". Note that this segue is connected to the Players view controller itself, not to any specific control inside it.



While we're here, ctrl-drag from the prototype cell to the new view controller that we just added. Make this a push segue named "RatePlayer". There are now three outgoing segues from the Players screen. Double-click the 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. We'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 we 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. We need to make a small change to that method. Previously, the sender parameter contained the UITableViewCell object that triggered the segue. Now, however, we're not sending along a table view cell but an NSIndexPath (because that's what we just put in the sender parameter of performSegueWithIdentifier).

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 send along anything you want. I chose to send the NSIndexPath because that was the least amount of work. (We could also have sent the Player object from that row, for example.)

Run the app. Tapping the blue disclosure button now brings up the Edit Player screen (modally) and tapping the row slides in the Rate Player screen (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 two 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.

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 we're using the delegate pattern to communicate back to the source view controller.

The top of **RatePlayerViewController.m** should look like this, it's just the imports and synthesize statements for the properties:

```
#import "RatePlayerViewController.h"
#import "Player.h"

@implementation RatePlayerViewController

@synthesize delegate;
@synthesize player;
```

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 "Rate Player").

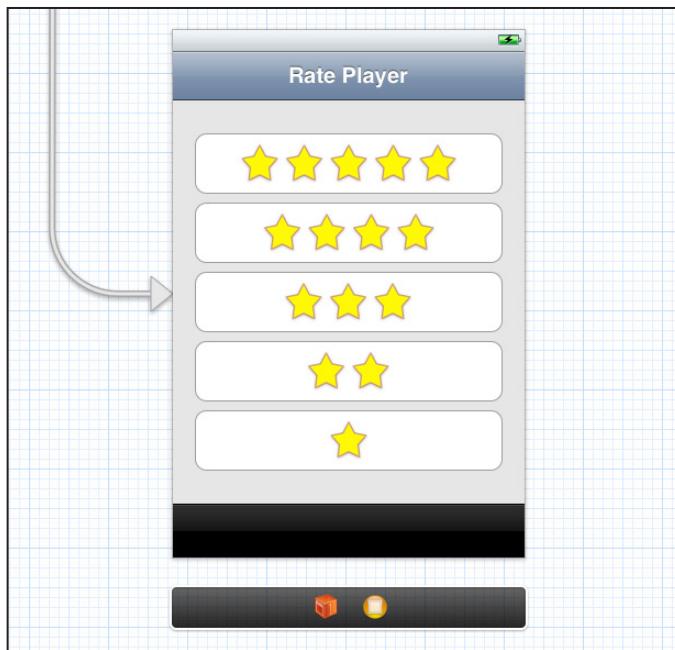


The interesting part in this view controller is the rateAction method, so add it next:

```
- (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 sender is a UIButton. What we'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 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.

I also made the background color for the scene's main view light gray so the buttons stand out a bit more.

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)ratePlayerViewController:
(RatePlayerViewController *)controller
didPickRatingForPlayer:(Player *)player
{
    NSUInteger index = [self.players indexOfObject:player];
    NSIndexPath *indexPath =
    [NSIndexPath indexPathForRow:index inSection:0];
    [self.tableView reloadRowsAtIndexPaths:
    [NSArray arrayWithObject:indexPath]
    withRowAnimation:UITableViewRowAnimationAutomatic];

    [self.navigationController popViewControllerAnimated:YES];
}
```

Again, we simply redraw the table view cell for the player that was changed and pop the Rate Player screen off the navigation stack.

Of course, we can't forget `prepareForSegue:`

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    // ...existing code...

    else if ([segue.identifier isEqualToString:@"RatePlayer"])
    {
        RatePlayerViewController *ratePlayerViewController =
            segue.destinationViewController;
        ratePlayerViewController.delegate = self;

        NSIndexPath *indexPath =
        [self.tableView indexPathForCell:sender];
        Player *player = [self.players objectAtIndex:indexPath.row];
        ratePlayerViewController.player = player;
    }
}
```



```
    }  
}
```

Because this scene has three outgoing segues, there are also three if-statements in `prepareForSegue`.

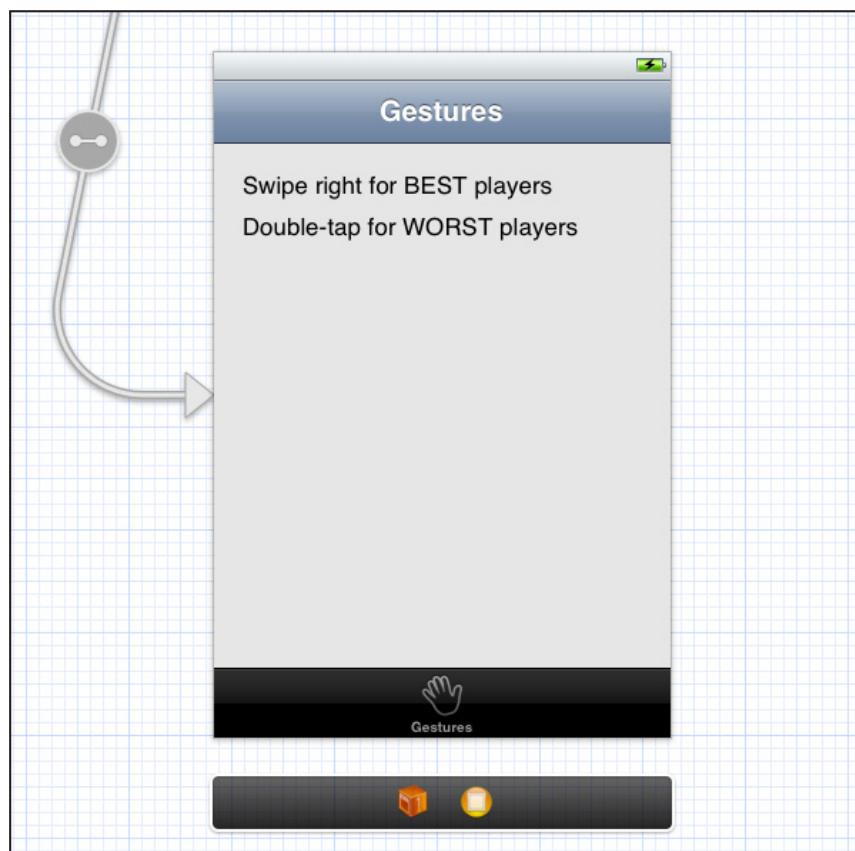
Run the app and verify that it works.

Gestures

We've been neglecting the second tab of our app. Let's give it something to do. The project has a class named `ViewController` that was originally generated by the Xcode template, but we haven't used it until now. Rename that class to `GesturesViewController.h/m`.

In the storyboard, go to the view controller that is hooked up to the second tab and set its class to "GesturesViewController".

Drag in some labels and a Navigation Bar to make it look somewhat like this:



We're not going to push any screens on the navigation stack here, so we don't need to embed this scene into a Navigation Controller. Just putting a Navigation Bar sub-view at the top is enough.

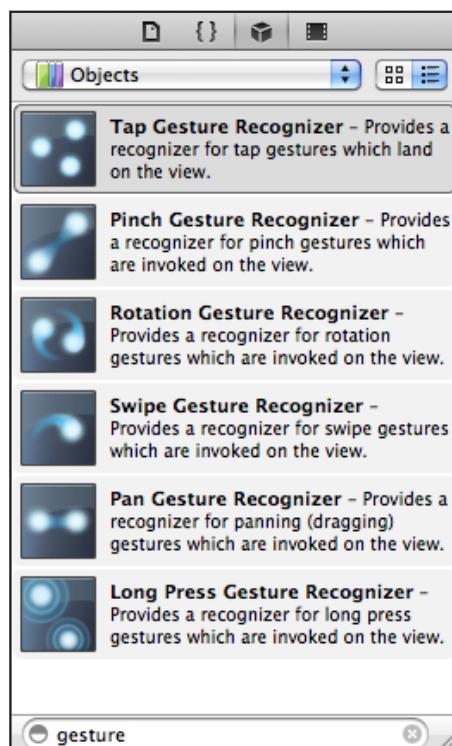
As the text on the labels indicates, we're going to add gestures to this screen. If you swipe to the right, we'll pop up a new screen that lists all the best players (5 stars); if you double-tap we'll list the worst players (1 star) instead. We need some place to list those best/worst players, and we'll add a new table view controller for that.

Drag a Navigation Controller from the Object Library into the canvas. That actually gives you two new scenes: the Navigation Controller itself and a default Root View Controller attached to it. Delete that Root View Controller as we don't need it.

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 - rootViewController" 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 we 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:

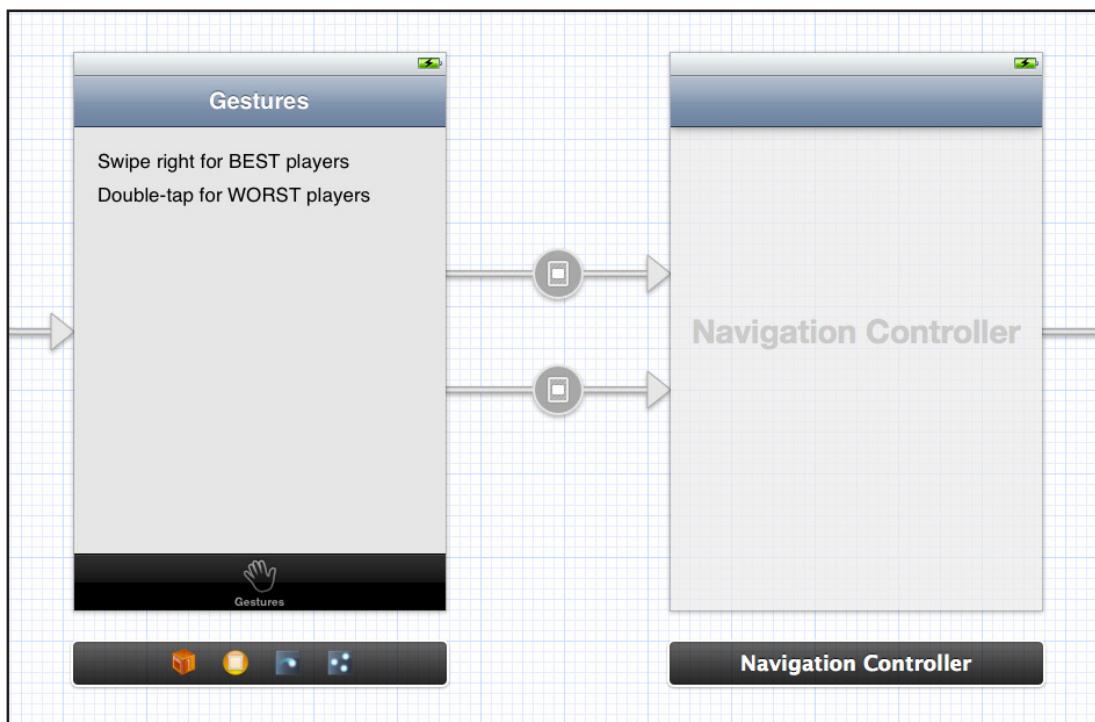


Drag the Swipe Gesture Recognizer into the Gestures screen. This will add an icon for the gesture recognizer to the dock:



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 should crash :]

Note: At the time of writing this chapter, there is a bug in iOS where adding a gesture recognizer to a view controller inside a Tab Bar Controller crashes the app. As a workaround, you can use regular buttons to trigger the segues.

Now that we can make it appear, let's make the Ranking screen do something. Create a new UITableViewController subclass named "RankingViewController".

Replace **RankingViewController.h** with:

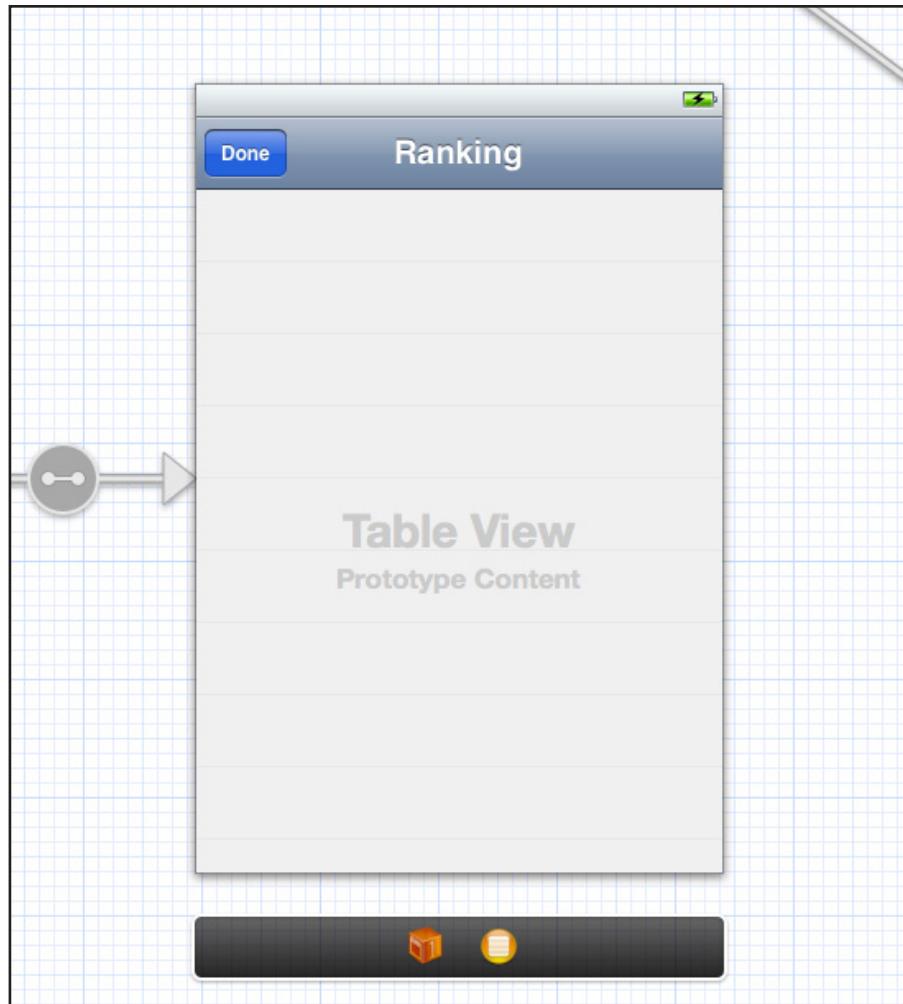
```
@interface RankingViewController : UITableViewController  
  
@property (nonatomic, strong) NSMutableArray *rankedPlayers;  
  
- (IBAction)done:(id)sender;  
  
@end
```

Go to the storyboard and set the class for the Ranking screen 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 we'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:





Replace the table view data source methods in **RankingViewController.m** with the following:

```
- (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 objectAtIndex:indexPath.row];
cell.textLabel.text = player.name;
cell.detailTextLabel.text = player.game;

return cell;
}
```

Naturally, we need to import the Player class and synthesize our rankedPlayers property:

```
#import "Player.h"

@implementation RankingViewController

@synthesize rankedPlayers;
```

Finally, add 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. We still need to give it the list of ranked players.

Add the following property to **GesturesViewController.h**:

```
@property (nonatomic, strong) NSArray *players;
```

Synthesize it in **GesturesViewController.m**:

```
@synthesize players;
```

This file will also need these two 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] objectAtIndex:0];

        rankingViewController.rankedPlayers =
            [self playersWithRating:5];
        rankingViewController.title = @"Best Players";
    }
    else if ([segue.identifier isEqualToString:@"WorstPlayers"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        RankingViewController *rankingViewController =
            [[navigationController viewControllers] objectAtIndex:0];

        rankingViewController.rankedPlayers =
            [self playersWithRating:1];
        rankingViewController.title = @"Worst Players";
    }
}
```

For both segues we'll first get the Navigation Controller that is on the other end of the segue, and from that we can obtain the `RankingViewController` instance. Then we give it the list of ranked players and a title.

The `playersWithRating` method is also new. Add it above `prepareForSegue`:

```
- (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.

Add the following import to AppDelegate.m:

```
#import "GesturesViewController.h"
```

Add the following to the bottom of didFinishLaunchingWithOptions:

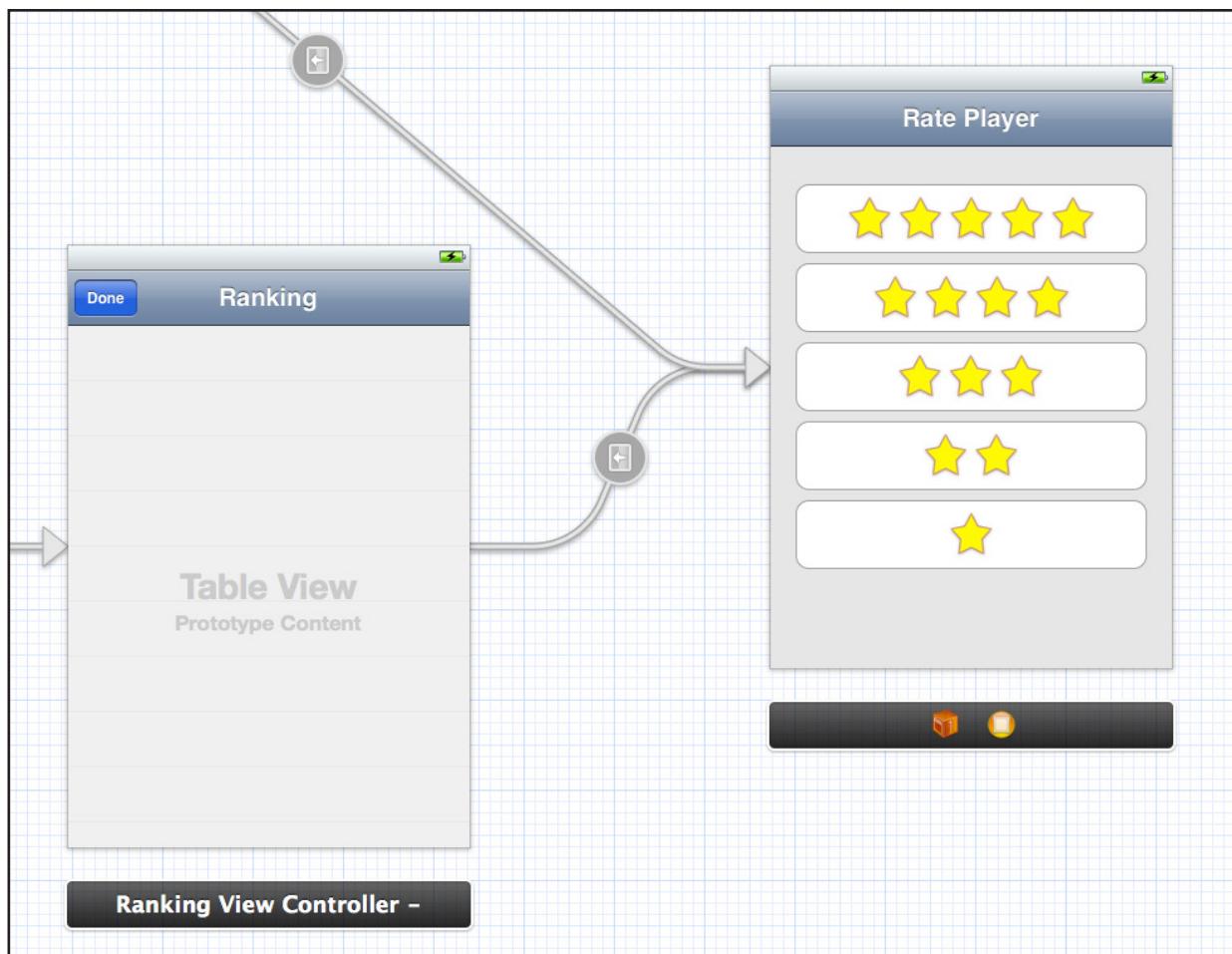
```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
  
    // ...existing code...  
  
    GesturesViewController *gesturesViewController =  
        [[tabBarController viewControllers] objectAtIndex:1];  
    gesturesViewController.players = 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.

We're not done yet. We are also going to connect the Ranking screen to the Rate Player screen. Not for any good reason, but simply because we 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.

If we had hardcoded the relationship between RatePlayerViewController and PlayersViewController -- instead of using a delegate -- 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 we don't have a prototype cell in the Ranking screen, we'll have to perform the segue manually. In **RankingViewController.m**, change didSelectRowAtIndexPath: to the following:

```
- (void)tableView:(UITableView *)tableView  
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```



```
{
    Player *player = [self.rankedPlayers objectAtIndex:indexPath.row];
    [self performSegueWithIdentifier:@"RatePlayer" sender:player];
}
```

First we find the Player object in question and then send it along with performSegueWithIdentifier in its sender parameter. That may be abusing the intent of sender a little (it's supposed to contain the object that initiated the segue), but it works just fine.

Of course we 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 the Storyboard Editor. 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, we have to remove the player from the Ranking screen if the rating changes, because when that happens he 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 we'll set this property's value to 5, for the list of worst players we'll set it to 1.

Then synthesize it in **RankingViewController.m**:

```
@synthesize requiredRating;
```

And change the `RatePlayerViewControllerAnimated` method to the following:

```
- (void)ratePlayerViewControllerAnimated:(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:
            [NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
    }

    [self.navigationController popViewControllerAnimated:YES];
}
```

If the rating changed, we remove the Player object from our array and from the table.

In `GesturesViewController`, we have to set `requiredRating` to the proper value before we transition 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;  
    }  
}
```

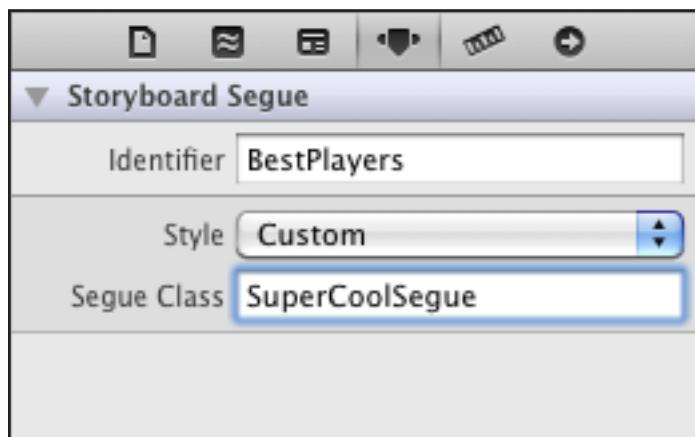
Try it out and see what happens.

Note: We 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 we'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 "SuperSegue", subclass of `UIStoryboardSegue`.

All we need to add to this class is a "perform" method. To start, add the simplest possible implementation inside **SuperSegue.m**:

```
@implementation SuperSegue

- (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. Previously you may have used the `presentModalViewControllerAnimated:` method for this, but as of iOS 5, this new method is the preferred way to present view controllers.

Try it out. When you perform the gesture, the Ranking screen appears 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 **SuperSegue.m** with:

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

@implementation SuperSegue

- (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 =
        UIGraphicsGetImageFromCurrentImageContext();
    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 -  
        destinationImageView.bounds.size.width, oldCenter.y);  
    destinationImageView.center = newCenter;  
  
    // Start the animation  
    [UIView animateWithDuration:0.5f delay:0  
        options:UIViewAnimationOptionCurveEaseOut  
        animations:^(void)  
    {  
        destinationImageView.transform = CGAffineTransformIdentity;  
        destinationImageView.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.

We add that UIImage as a 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 we start the animation.



When the animation is done, we remove the image view and we 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, 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.

Storyboards and the iPad

We're going to make the Ratings app universal so that it also runs on the iPad.

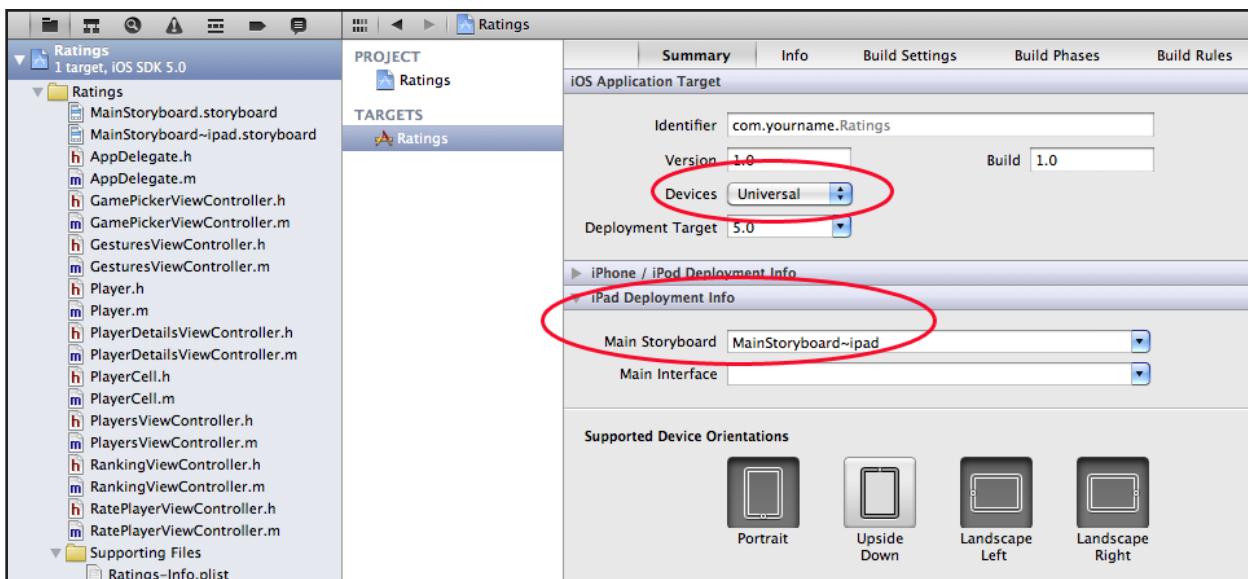
In the project's Target Summary screen, under iOS Application Target, change the Devices setting to Universal. This adds a new iPad Deployment Info section to the screen.

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. You'll 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.

Back in your target settings, in the iPad Deployment Info section on the Target Summary screen, choose `MainStoryboard~ipad` as the Main Storyboard:





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] objectAtIndex:0];
        PlayersViewController *playersViewController =
            [[navigationController viewControllers] objectAtIndex:0];
        playersViewController.players = players;

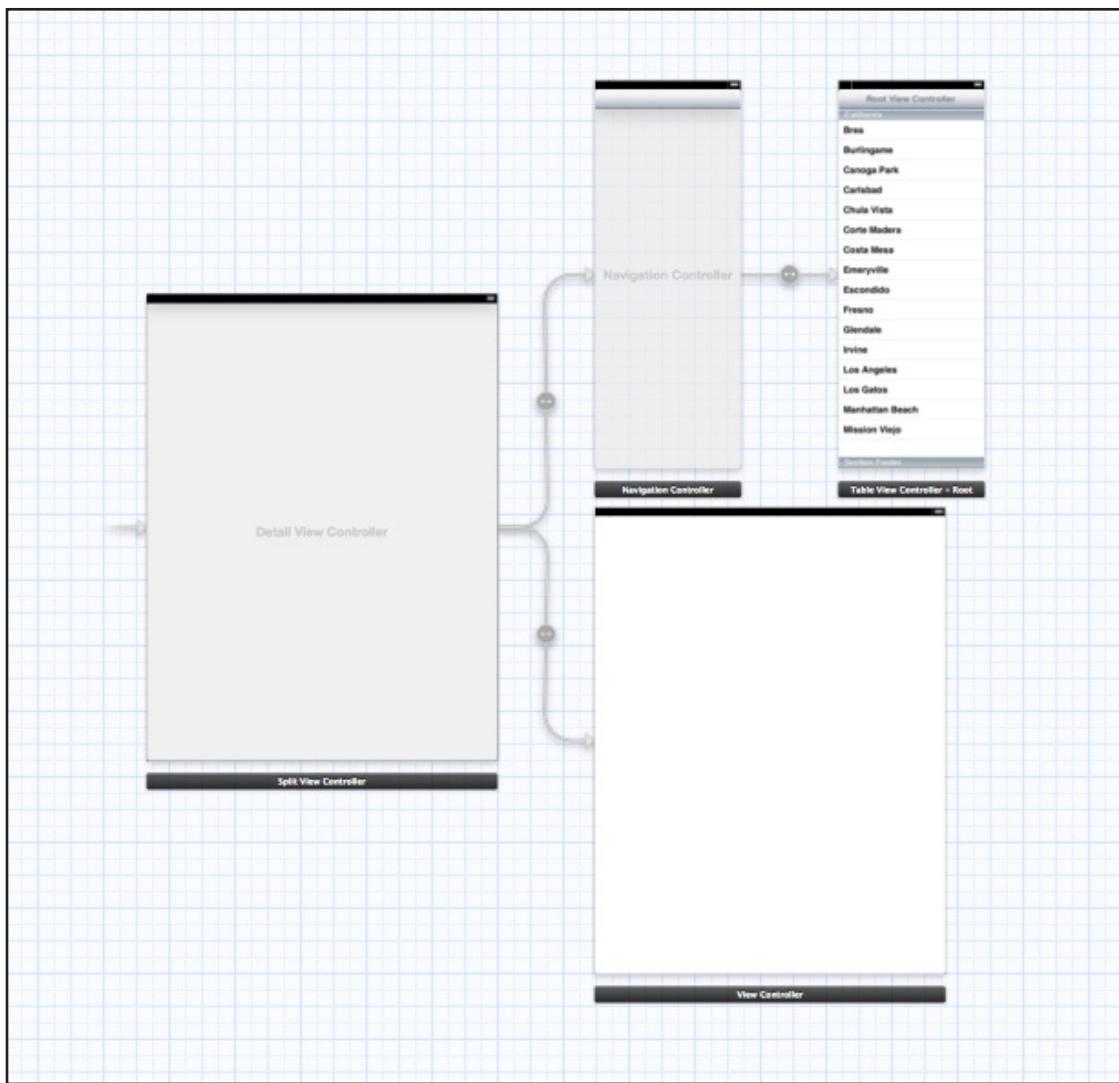
        GesturesViewController *gesturesViewController =
            [[tabBarController viewControllers] objectAtIndex:1];
        gesturesViewController.players = players;
    }
    return YES;
}
```

We don't want to do any of that stuff when we're 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. 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.

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 needed 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 **UIViewController subclass** file to the project and name it **DetailViewController**. 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, strong) IBOutlet UIToolbar *toolbar;

@end
```

We're making this class the delegate for the Split View Controller so we will be notified whenever the device is rotating.

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

```
#import "DetailViewController.h"

@implementation DetailViewController
{
    UIPopoverController *masterPopoverController;
}

@synthesize toolbar;

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.toolbar = nil;
}

- (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;
}

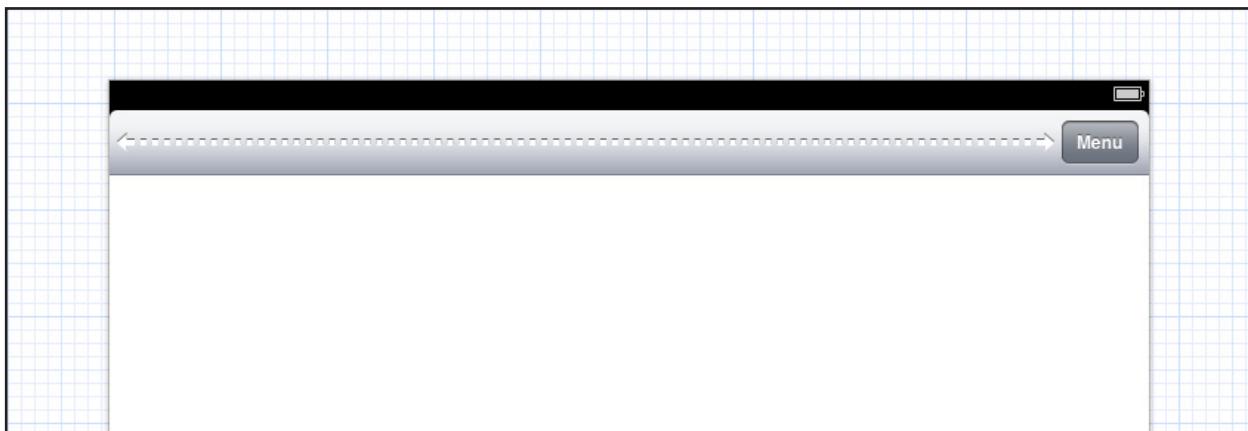
- (void)splitViewController:(UISplitViewController *)splitController
    willShowViewController:(UIViewController *)viewController
    invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
{
    NSMutableArray *items = [[self.toolbar items] mutableCopy];
    [items removeObject:barButtonItem];
    [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.

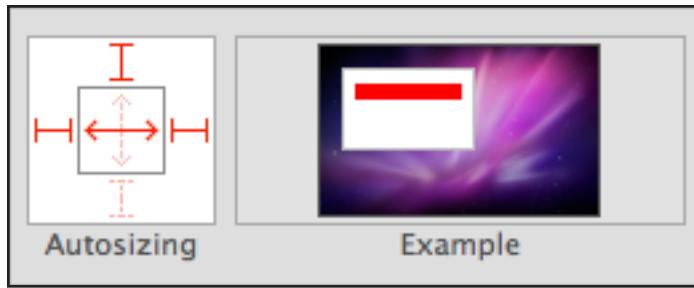
In the storyboard, set the Class of the big scene to "DetailViewController". Drag a Toolbar into this scene. The toolbar comes with a default Bar Button Item named simply "Item". Rename it to "Menu" (this is for a popover we'll be adding later) and add a flexible space in front of it.

The design should look like this:



Connect the Toolbar to the view controller's toolbar property. 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 we want this one to sit at the top at all times, or it won't look right when the device is rotated.

We're not done yet. We 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 we 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**. If you check the "Targeted for iPad" option, then this will properly fill in the `shouldAutorotateToInterfaceOrientation` method already.

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 we're not going to worry about that.

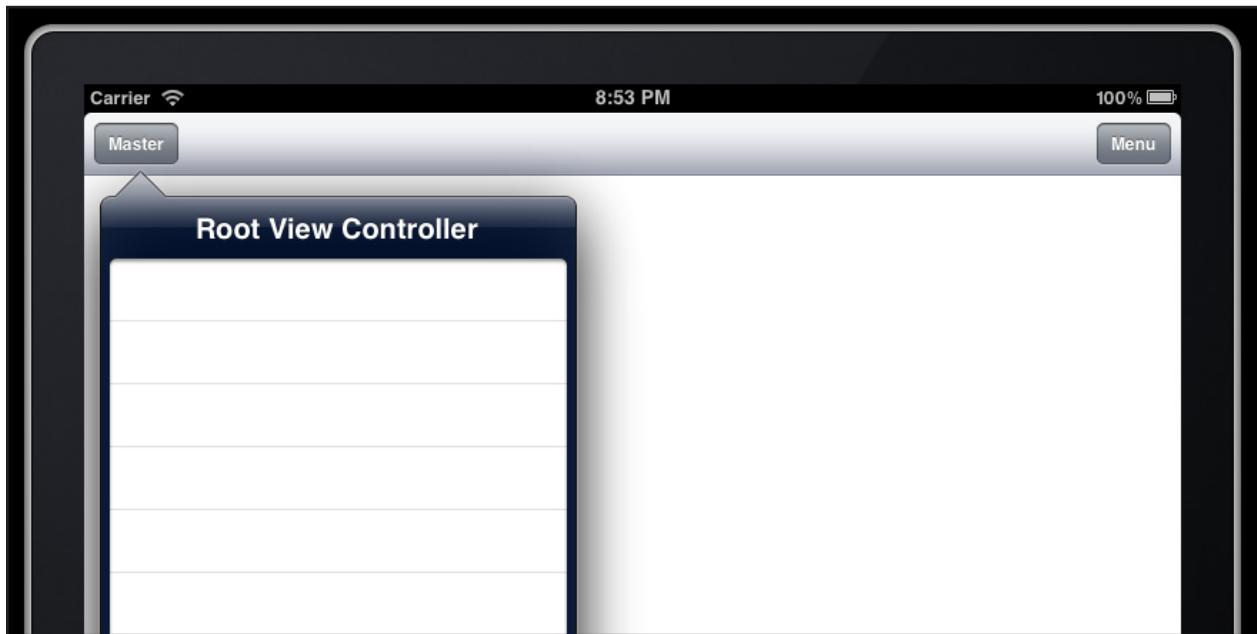
There is one more thing to do. The DetailViewController class is the delegate for the split view controller, but we haven't set up that delegate relationship anywhere yet. As you know you cannot make these kinds of connections directly in the Storyboard Editor (unfortunately!) so we'll have to write 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];
    }
}
```

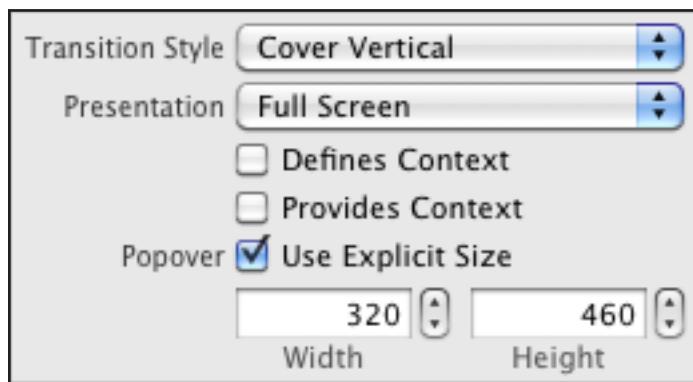
```
    return YES;  
}
```

Run the app and you should now have a fully functional split view controller!



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.

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:



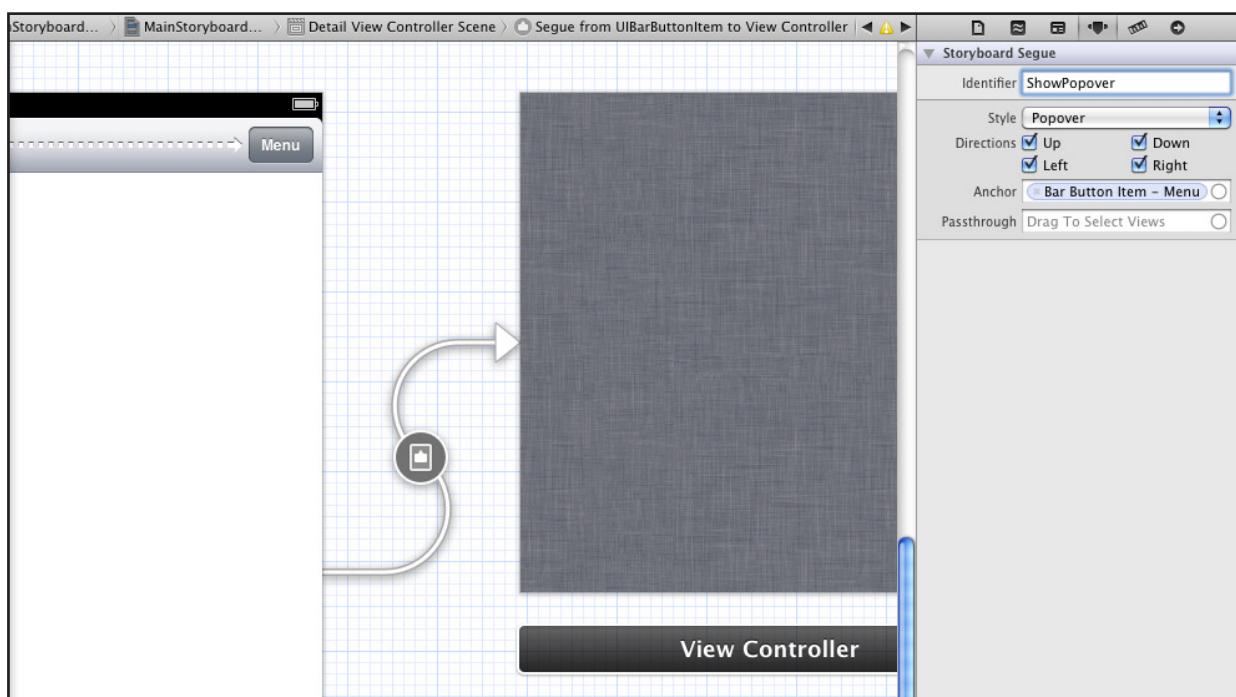
We can also easily create our own popovers. You add a new scene to the storyboard and then simply make a "Popover" segue to it.



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 we can see the popover actually works, change the Background Color of the view to something other than white (I chose 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:



If you run the app, you'll 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. We should really capture that `popoverController` object in an instance variable so that we 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 now very familiar `prepareForSegue` method:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"ShowPopover"])
    {
        menuPopoverController =
            ((UIStoryboardPopoverSegue *)segue).popoverController;
        menuPopoverController.delegate = self;
    }
}
```

Here we will put the value from the segue's `popoverController` property into our own `menuPopoverController` variable and make ourselves the delegate for the popover controller.

Add the delegate method:

```
#pragma mark - UIPopoverControllerDelegate

- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)thePopoverController
{
    menuPopoverController.delegate = nil;
    menuPopoverController = nil;
}
```

It simply sets the ivar back to `nil` when the popover is dismissed.

Now that we have an instance variable that refers to the segue's popover controller when it is visible, we 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 work-around 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 currently open we do have a reference to it in our 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];
        }

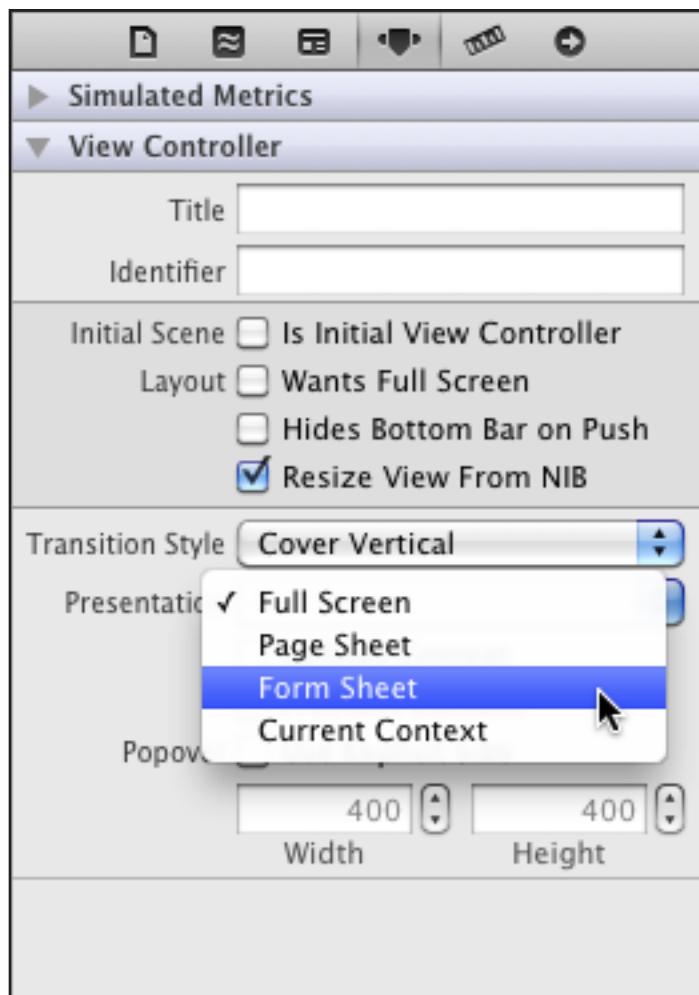
        menuPopoverController =
            ((UIStoryboardPopoverSegue *)segue).popoverController;
        menuPopoverController.delegate = self;
    }
}
```

No matter how many times you tap the Menu button now, only one popover is ever visible.

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 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:



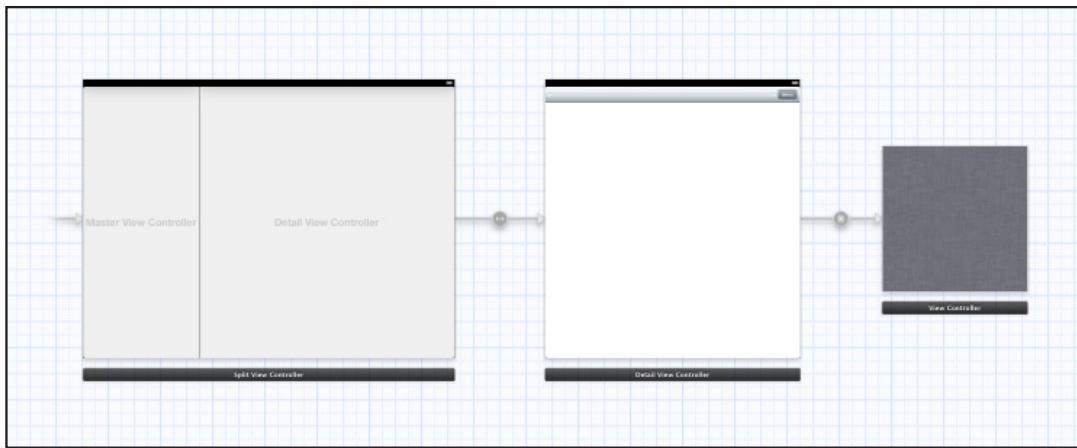


Manually Loading Storyboards

The iPad version of our 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. We'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 separated. However, for the purposes of this tutorial it's 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 from the project if you want. The storyboard now looks like this:



We'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 we attempt to load the iPhone storyboard using:

```
UIStoryboard *storyboard =  
    [UIStoryboard storyboardWithName:@"MainStoryboard" bundle:nil];
```

then this will actually load the `MainStoryboard~ipad` file because we're a universal app running in iPad mode and the app will get totally confused. Normally you wouldn't do this so it's no big deal but for this particular section of the tutorial we'd better rename the `MainStoryboard~ipad` file. Let's 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  
{  
    // ...existing code ...  
    UITabBarController *tabBarController;  
    if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad)
```

```

{
    tabBarController = (
        UITabBarController *)self.window.rootViewController;
}
else
{
    UISplitViewController *splitViewController =
        (UISplitViewController *)self.window.rootViewController;

    UIStoryboard *storyboard = [UIStoryboard storyboardWithName:
        @"MainStoryboard" bundle:nil];
    tabBarController = [storyboard
        instantiateInitialViewController];

    NSArray *viewControllers = [NSArray arrayWithObjects:
        tabBarController,
        [splitViewController.viewControllers lastObject], nil];
    splitViewController.viewControllers = viewControllers;

    splitViewController.delegate =
        [splitViewController.viewControllers lastObject];
}

UINavigationController *navigationController =
    [[tabBarController viewControllers] objectAtIndex:0];
PlayersViewController *playersViewController =
    [[navigationController viewControllers] objectAtIndex:0];
playersViewController.players = players;

GesturesViewController *gesturesViewController =
    [[tabBarController viewControllers] objectAtIndex:1];
gesturesViewController.players = players;
return YES;
}

```

The new part is this:

```

UIStoryboard *storyboard = [UIStoryboard storyboardWithName:
    @"MainStoryboard" bundle:nil];
tabBarController = [storyboard instantiateInitialViewController];

```

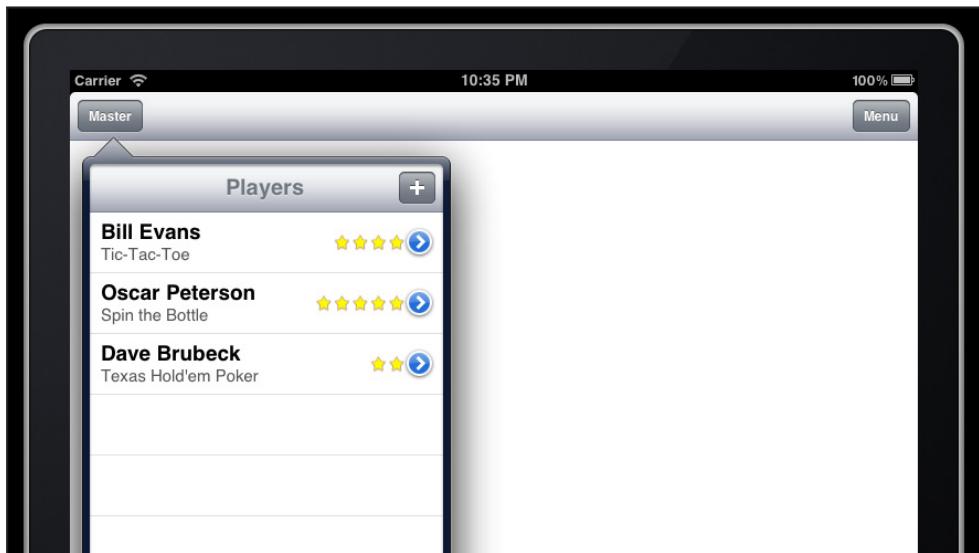
This loads the MainStoryboard file into a new UIStoryboard object and then calls instantiateInitialViewController to load its initial view controller, which in our case is the Tab Bar Controller.

Once we have the Tab Bar Controller, we need to put it into the Split View Controller. Currently the split-view only contains the Detail View Controller, so we add the Tab Bar Controller to its viewControllers property:



```
NSArray *viewControllers = [NSArray arrayWithObjects:  
    tabBarController,  
    [splitViewController.viewControllers lastObject], nil];  
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 we never configured the scenes from our original storyboard to work on the iPad.

Go through the .m files for all the view controllers and replace shouldAutorotateToInterfaceOrientation with:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:  
    (UIInterfaceOrientation)interfaceOrientation  
{  
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)  
        return YES;  
  
    return (interfaceOrientation !=  
        UIInterfaceOrientationPortraitUpsideDown);  
}
```

Now the app will properly rotate to landscape mode.

Unfortunately, the Storyboard Editor doesn't let us set the popover size for view controllers in an iPhone storyboard (which is understandable because the iPhone doesn't have popovers), so we'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.

There is one more minor problem with the app and that is that our modal view controllers are being presented full screen, which looks a little weird. This is related to the modal presentation style setting of the view controllers. If in `prepareForSegue`, you do,

```
navigationController.modalPresentationStyle =
    UIModalPresentationCurrentContext;
```

then the modal scene doesn't take over the whole screen - the modal view will appear just in the master controller.

`UIStoryboard`'s `instantiateViewControllerWithIdentifier` 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. (FYI: If you run the app now and press the disclosure button the app will crash with the error message: "Receiver (<PlayersViewController>) has no segue with identifier 'EditPlayer'".)

Replace the `accessoryButtonTappedForRowWithIndexPath` method in **PlayersViewController.m** with the following:

```
- (void)tableView:(UITableView *)tableView
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
    UINavigationController *navigationController =
        [self.storyboard instantiateViewControllerWithIdentifier:
            @"PlayerDetails"];
    PlayerDetailsViewController *playerDetailsViewController =
        [[navigationController viewControllers] objectAtIndex:0];
    playerDetailsViewController.delegate = self;

    Player *player = [self.players objectAtIndex:indexPath.row];
    playerDetailsViewController.playerToEdit = player;

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

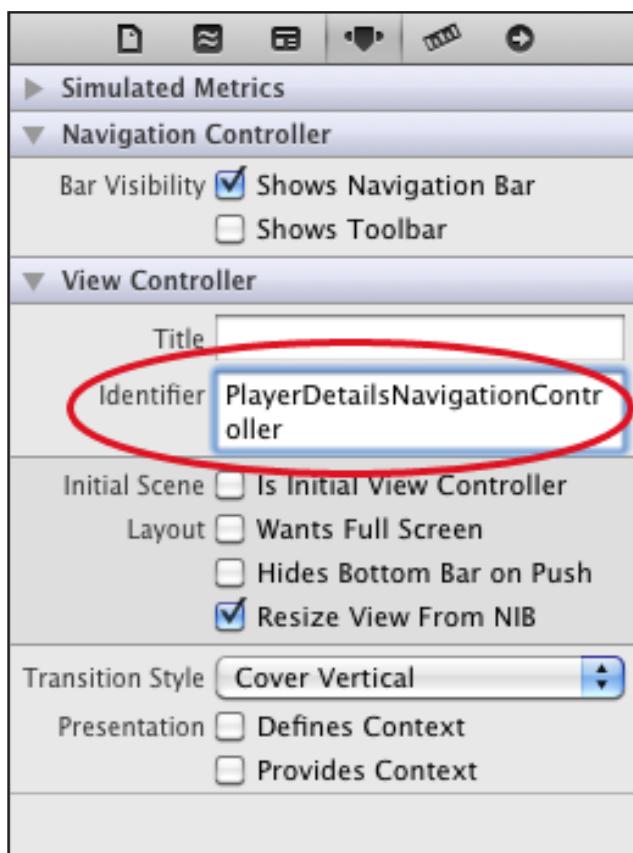
This is very similar to what we did in `prepareForSegue`, except for the first line:



```
UINavigationController *navigationController =  
[self.storyboard instantiateViewControllerWithIdentifier:  
@"PlayerDetailsNavigationController"];
```

The `self.storyboard` property refers to the `UIStoryboard` object that this view controller was loaded from. We 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 on the view controller in its Attributes Inspector:



Note: We 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 we no longer have 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 and the View Controller. Gesture recognizers are also added to the dock. In theory you can drag any other objects that you want into the dock, just like you can with nibs. The problem is that the Storyboard Editor doesn't give you any means to edit these objects.

If you want to change the background image for a static cell, for example, 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 added to the canvas anywhere. I hope the Storyboard team adds in this functionality because it's really handy to load extra objects that way.

By the way, 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, for example. 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 OS 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 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 "stuff" 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 which 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, you 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 iCloud 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.



iTunes so far 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. And 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, calendars.

iCloud is great news for developers. By using a set of new APIs, you can configure your app to store/retrieve data on iCloud as well. Can you think of the advantages? The possibilities are just endless!

In this tutorial, we'll investigate iCloud by implementing a set of simple applications which 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 iOS devices running iOS 5 for testing, such as an iPhone and an iPad. The simulator does not currently have iCloud support.

An important note about Apple's policy. 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 the cloud if the user chooses to enable backup for the application. If your data can be recreated somehow just don't store it in "Documents", otherwise backups are not efficient and your app wastes user's iCloud storage space. 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 we begin, let's talk about how iCloud works.

In iOS each application has its data stored in a local directory, and each app can only access data in its own directory. This prevents apps from reading or modifying data from other apps (although there are some alternate methods of transferring data between apps built into the OS).

iCloud allows you to upload your local data to central servers on the net, and receive updates from other devices. The replication of content across different devices is achieved by means of a continuous background process (daemon) which detects changes to a resource (document) and uploads them to the central storage.



This works real-time and enables another interesting feature: notifications. For example, whenever there is a conflict about a document, the application can be aware of that and you can implement a resolution policy.

If you ever tried to create something like this with your own apps, you know there are several major challenges 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? You somehow 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 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 pushed to the cloud, 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 app 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" in this list.

Configuring iCloud in iOS 5

The first time you install iOS 5, 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 iOS 5 installed on 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 use <http://www.icloud.com> to see what's in your calendar.

Once you're sure iCloud is working OK on your device, let's try it out in an app of our own creation!

Enabling iCloud in your App

In this tutorial, we'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 we 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 we 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 we will be naming the project. For example, you could enter com.yourname.dox.

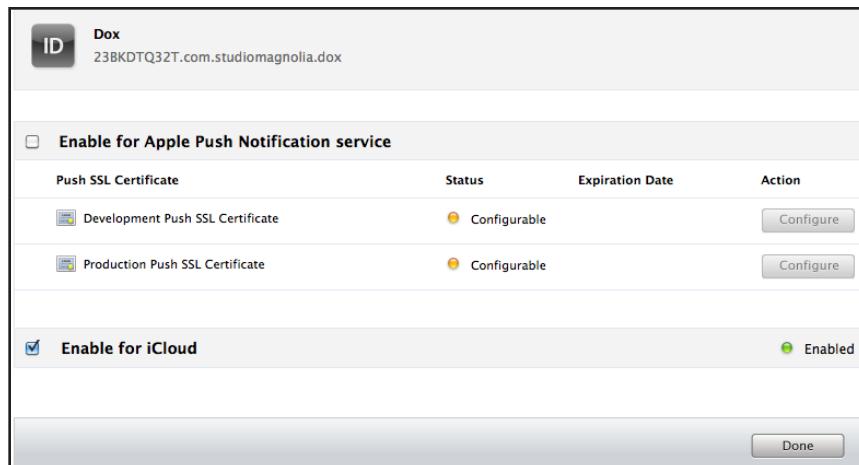
The screenshot shows the 'Create App ID' page in the Provisioning Portal. The left sidebar has a 'App IDs' tab selected. The main area has sections for 'Description' (containing 'Dox iCloud App'), 'Bundle Seed ID (App ID Prefix)' (with a dropdown set to 'Use Team ID'), and 'Bundle Identifier (App ID Suffix)' (containing 'com.studiomagnolia.dox'). At the bottom are 'Cancel' and 'Submit' buttons.

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 that 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 below screenshot:

The screenshot shows the 'Modify iOS Development Provisioning Profile' page. The 'Profile name' is set to 'Dox'. Under 'Certificates', 'Cesare Rocchi' is selected. The 'App ID' dropdown is set to 'Dox'. In the 'Devices' section, several devices are listed with checkboxes: 'Etta' and 'jeff' are checked, while others like 'iPhone 5S' and 'iPhone 6' are not. At the bottom are 'Cancel' and 'Submit' buttons.

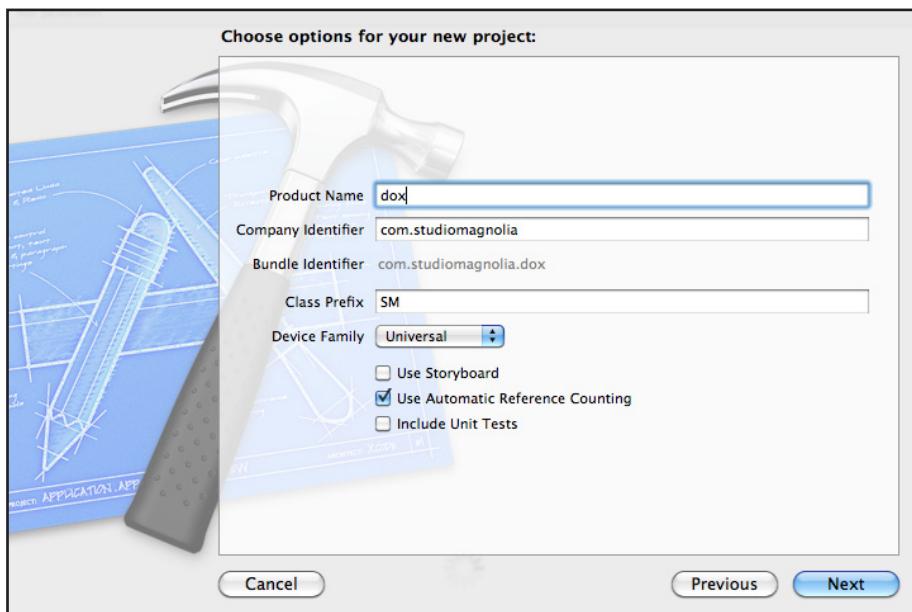
After creating the profile, refresh the page until it is ready for download, and then download it to your machine. Once it's downloaded, double click it to bring it into Xcode, and verify that it is visible in Xcode's Organizer:





3) Configure your Xcode project for iCloud

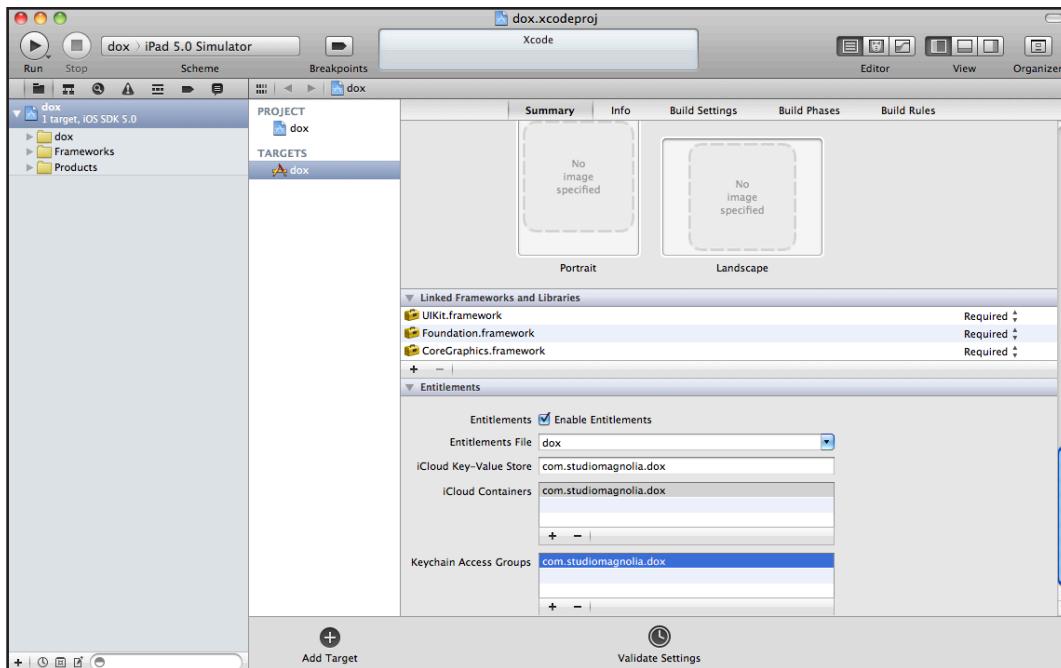
Start up Xcode and create a new project with the **iOS\Application\Single View Application** template. Enter **dox** for the product name, enter the company identifier you used when creating your App ID, enter **SM** for the class prefix, set the device family to **Universal**, and make sure **Use Automatic Reference Counting** is 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, click the **Enable Entitlements** checkbox, and it will auto-populate the other 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. We will discuss the key-value store more in the next chapter.
- The **iCloud 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 Access Groups** includes keys needed by applications which 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're ready to go! If you like you can edit the same settings by editing the file `dox.entitlements` which is included in your project.



Checking for iCloud Availability

When building an application which 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 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 **SAppDelegate.m**, and add the following code at the bottom of `application:didFinishLaunchingWithOptions` (before the return YES):

```
NSURL *ubiq = [[NSFileManager defaultManager]
                 URLForUbiquityContainerIdentifier:nil];
if (ubiq) {
    NSLog(@"iCloud access at %@", ubiq);
    // TODO: Load document...
} else {
    NSLog(@"No iCloud access");
}
```

Here we use a new method you haven't seen yet called `URLForUbiquityContainerIdentifier`. This method allows you to pass in a container identifier (like you set up earlier in the iCloud Containers section) and it will return to you a URL to use to access files in iCloud storage.

You need to call this for each container you want to access to give your app permission to access the URL. If you pass in `nil` to the method (like we do here), it automatically returns the first iCloud Container set up for the project. Since we only have one container, this makes it nice and easy.

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 at file:///localhost/private/var/mobile/Library/Mobile%20
Documents/D2CD3DR3GP~com~studiomagnolia~dox/
```

Note that the URL this returns is actually a local URL on the system! This is because the iCloud daemon transfers files from the central servers to a local directory on the device on your behalf. Your application can then retrieve files from this directory, or send updated versions to this directory, and the iCloud daemon will synchronize everything for you.

This directory is outside of your app's directory, but as mentioned above the act of calling `URLForUbiquityContainerIdentifier` gives your app permission to access this directory.



iCloud API Overview

Before we proceed further with the code, let's take a few minutes to give an overview of the APIs we'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 iOS 5 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 our case will be the text of a note). In your apps, you'd 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 we want to open a document, which has been already created on iCloud. We'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 we can do something else on the UI so the application is not blocked. Once the reading is done we 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 our simple notes app:

```
- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName
```

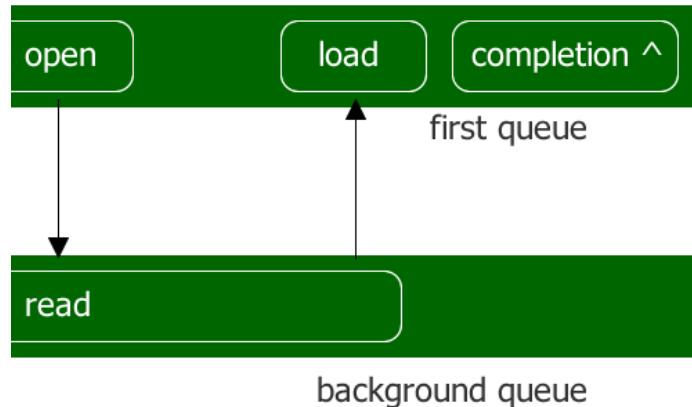


```
error:(NSError ***)outError
{
    self.noteContent = [[NSString alloc]
        initWithBytes:[contents bytes]
        length:[contents length]
        encoding:NSUTF8StringEncoding];
    return YES;
}
```

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.

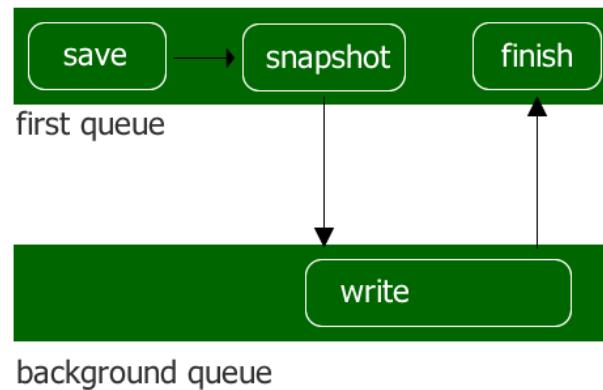
After `loadFromContents:type:error` completes, you'll receive the callback you provided in the `openWithCompletionHandler:` block, as shown in the diagram below:



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 we have to parse an NSData instance, but while writing we have to convert our 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 autosaving 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 which is called when the writing procedure is done.

When asked to write, the background queue asks for a snapshot of the contents of our `UIDocument` subclass. This is accomplished by overriding another method of `UIDocument` - `contentsForType:error`.

Here you should return an `NSData` instance which describes the current model to be saved. In our notes application, we'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 we 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. We'll cover using `NSFileWrapper` in the next chapter.

As we 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 a change is registered via an undo or redo action the `updateChangeCount:` is called automatically. We will cover 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 we 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 our 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, our 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:

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

@interface SMNote : UIDocument

@property (strong) NSString * noteContent;
```



```
@end
```

As we have learned above we have two override points, one when we read and one when we write. Add the implementation of these by replacing **SMnote.m** with the following:

```
#import "SMNote.h"

@implementation SMNote

@synthesize noteContent;

// Called whenever 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 first created, assign some default content
        self.noteContent = @"Empty";
    }

    return YES;
}

// Called whenever 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 we load a file we need a procedure to 'transform' the NSData contents returned by the background queue into a string. Conversely, when we save we have to encode our string into an NSData object. In both cases we 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 we need to model the document is already over! Now we can move to the code related to loading and updating.

Opening an iCloud File

First of all we should decide a file name for our document. For this tutorial, we'll start by creating a single filename. Add the following #define at the top of **SMAppDelegate.m**:

```
#define kFILENAME @"mydocument.dox"
```

Next, let's extend the application delegate to keep track of our document, and a metadata query to look up the document in iCloud. Modify **SMAppDelegate.h** to look like the following:

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

@class SMViewController;

@interface SMAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) SMViewController *viewController;
@property (strong) SMNote * doc;
@property (strong) NSMetadataQuery *query;

- (void)loadDocument;

@end
```

Then switch to **SMAppDelegate.m** and synthesize the new properties:

```
@synthesize doc = _doc;
@synthesize query = _query;
```

We've already added code into `application:didFinishLaunchingWithOptions` to check for the availability of iCloud. If iCloud is available, we want to call the new method we're about to write to load our document from iCloud, so add the following line of code right after where it says "TODO: Load document":

```
[self loadDocument];
```



Next add the `loadDocument` method. Let's put it together bit by bit so we can discuss all the code as we go.

```
- (void)loadDocument {  
  
    NSMetadataQuery *query = [[NSMetadataQuery alloc] init];  
    _query = query;  
  
}
```

Note that before we can load a document from iCloud, we first have to check what's there. Remember that we can't simply enumerate the local directory returned to us by `URLForUbiquityContainerIdentifier`, because there may be files in iCloud not yet pulled down locally.

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 we 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 CoreData or even arrays you probably know the approach. Basically, you build a predicate and set it as parameter of a query/search.

In our case we 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:  
        NSMetadataItemFSNameKey]];
```



```
    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.

Now the query is ready to be run, but since it is an asynchronous process we need to set up an observer to catch a notification when it completes.

The specific notification we are interested in has a pretty long (but descriptive) name: `NSMetadataQueryDidFinishGatheringNotification`. This is posted when the query has finished gathering info from iCloud.

So here is the final implementation of our method:

```
- (void)loadDocument {
    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];
}
```

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.

We 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 our case, we are working with just one file so we are just interested in the first element.

An `NSMetadataItem` is like a dictionary, storing keys and values. It has a set of pre-defined keys that you can use to look up information about each file:

- `NSMetadataItemURLKey`
- `NSMetadataItemFSNameKey`
- `NSMetadataItemDisplayNameKey`
- `NSMetadataItemIsUbiquitousKey`
- `NSMetadataUbiquitousItemHasUnresolvedConflictsKey`
- `NSMetadataUbiquitousItemIsDownloadedKey`
- `NSMetadataUbiquitousItemIsDownloadingKey`
- `NSMetadataUbiquitousItemIsUploadedKey`



- NSMetadataUbiquitousItemIsUploadingKey
- NSMetadataUbiquitousItemPercentDownloadedKey
- NSMetadataUbiquitousItemPercentUploadedKey

In our case, we are interested in NSMetadataItemURLKey, which points to the URL that we need to build our 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. We call that here, passing in the URL of the located file, and store it away in an instance variable.

Now we 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");  
            } 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 our 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, we need to write some code to create a doc. We will append this to the `loadData` method that we defined a few seconds ago. When the query returns zero results, we should:

- Retrieve the local iCloud directory
- Initialize an instance of document in that directory
- Call the `saveToURL` method
- When the save is successful we can call `openWithCompletionHandler`.

So add an else case to the if statement in `loadData` as follows:

```
else {  
  
    NSURL *ubiq = [[NSFileManager defaultManager]  
        URLForUbiquityContainerIdentifier:nil];  
    NSURL *ubiquitousPackage = [[ubiq URLByAppendingPathComponent:  
        @"Documents"] URLByAppendingPathComponent:kFILENAME];  
  
    SMNote *doc = [[SMNote alloc] initWithFileURL:ubiquitousPackage];  
    self.doc = doc;  
  
    [doc saveToURL:[doc fileURL]  
        forSaveOperation:UIDocumentSaveForCreating  
        completionHandler:^(BOOL success) {  
        if (success) {  
            [doc openWithCompletionHandler:^(BOOL success) {  
                NSLog(@"new document opened from iCloud");  
            }];  
        }  
    }];  
}
```

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.

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 our application is almost ready. The iCloud part is over, and we just need to set up the UI!



Setting Up the User Interface.

The Xcode project template we chose already set up an empty view controller for us. We will extend it by adding the current document and a UITextView to display the content of our note.

Start by modifying **SMViewController.h** to look like the following:

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

@interface SMViewController : UIViewController <UITextViewDelegate>

@property (strong) SMNote * doc;
@property (weak) IBOutlet UITextView * noteView;

@end
```

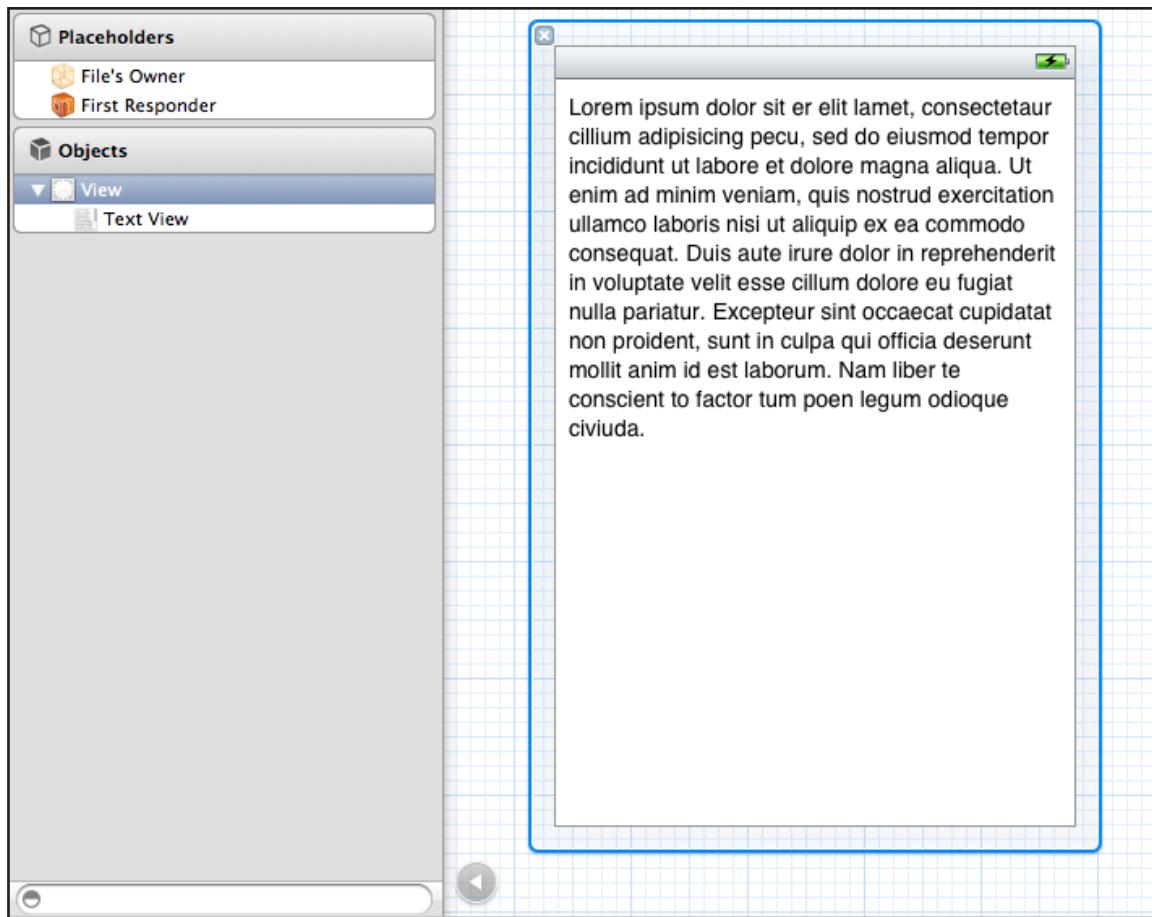
We have also marked the view controller as implementing UITextViewDelegate so that we can receive events from the text view.

Next, open up **SMViewController_iPhone.xib** and make the following changes:

1. Drag a Text View into the View, and make it fill the entire area.
2. Control-click the File's Owner, and drag a line from the noteView outlet to the Text View.
3. Control-click the Text View, and drag a line from the delegate to the File's Owner.

At this point your screen should look like this:





When you are done, repeat these steps for **SMViewController_iPad.xib** as well.

Next, open up **SMViewController.m** and synthesize your new properties as follows:

```
@synthesize doc;
@synthesize noteView;
```

Then modify `viewDidLoad` to register for the notification our code will send when our document changes (we'll add the code to send this notification later):

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

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(dataReloaded:)
        name:@"noteModified" object:nil];
}
```

Next, implement the method that gets called when the notification is received as follows:

```
- (void)dataReloaded:(NSNotification *)notification {  
  
    self.doc = notification.object;  
    self.noteView.text = self.doc.noteContent;  
  
}
```

This simply stores the current document and updates the text view according to the new content received.

In general substituting the old content with the new one is NOT a good practice. When we receive a notification of change from iCloud we should have a conflict resolution policy to enable the user to accept/refuse/merge the differences between the local version and the iCloud one. We'll discuss more about conflict resolution in the next chapter, but for now to keep things simple we'll just overwrite each time.

Next, implement `textViewDidChange:` to notify iCloud when the document changes, and modify the app to refresh the data in `viewWillAppear:` as well:

```
- (void)textViewDidChange:(UITextView *)textView {  
  
    self.doc.noteContent = textView.text;  
    [self.doc updateChangeCount:UIDocumentChangeDone];  
  
}  
  
- (void)viewWillAppear:(BOOL)animated  
{  
    [super viewWillAppear:animated];  
    self.noteView.text = self.doc.noteContent;  
}
```

As above this is not a great practice, because we are going to notify each iCloud about every single change (i.e. each time a character is added or deleted). For efficiency, it would be better to just tell iCloud every so often, or when the user has finished a batch of edits.

There's just one last step remaining - we need to add the code to send the "note-Modified" notification we registered for in `viewDidLoad`. The best place in this case is the `SMNote` class's `loadFromContents:ofType:error`, method which is called whenever data are read from the cloud.

So open up **SMNote.m** and add this line of code to the bottom of `loadFromContents:ofType:error` (before the return YES):



```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"noteModified"
    object:self];
```

Now we are really ready! The best way to test this application is the following: install it on two devices and run it on both. You should be able to edit on one and see the changes periodically propagated to the other!

Note that the propagation of changes is not immediate and might depend on your connectivity. In general, for our examples, it should take around 5-30 seconds.

Another way to check the correctness it to browse the list of files in your iCloud, which is a bit hidden in the menu. Here is the sequence:

Settings -> iCloud -> Storage and Backup -> Manage Storage -> Documents & Data -> Unknown

If the application works correctly you should see the note we created in our app:



The 'unknown' label comes from the fact that the application has not been uploaded and approved on the Apple Store yet.



Also note that users can delete files from iCloud from this screen at-will (without having to go through your app). So keep this in mind as you're developing.

Congrats - you have built your first iCloud-aware application!

Handling Multiple Documents

Cool, our example works and we are a bit more acquainted with the capabilities of iCloud. But what we have right now isn't enough to impress our users, or build an application that makes sense. Who wants to manage just one document?!

So next we are going to extend our application to manage more than one document at a time. The most natural development of our current prototype is to transform it into a notes application, as follows:

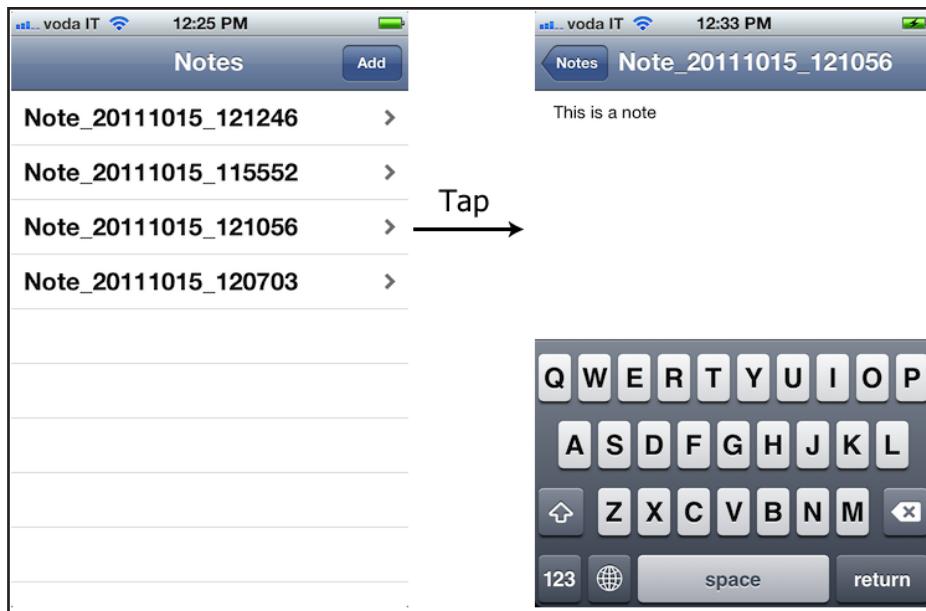
- The application will start with a view showing a list of notes
- Each note will have a unique id
- Tapping a note will show a single note view with the content
- Users can then edit the content
- The list of notes is updated when we launch the application or tap a refresh button

We will reuse some of the code of the previous project but we will need to reorganize it. Let's start by rearranging the user interface.

Reorganizing the User Interface

In our new project a single view is not enough, we'll need two. The first will be a table view which shows the list of notes. The second will be pretty similar to the main view of the previous project: it will show an editable text view.





Let's add an empty table view controller, and modify our app to show that first inside a navigation controller.

Create a new file with the **iOS\Cocoa Touch\UIViewController subclass** template, name the class **SMListViewController**, and make it a subclass of **UITableViewController**. You can leave both checkboxes unchecked.

Then open **SMAppDelegate.m**, and import **SMListViewController.h** at the top of the file:

```
#import "SMListViewController.h"
```

Then replace the lines up to "NSURL * ubiq = [[NSFileManager..." of application :**didFinishLaunchingWithOptions** with the following:

```
self.window =
    [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
SMListViewController * listViewController =
    [[SMListViewController alloc] initWithNibName:nil bundle:nil];
UINavigationController * navController =
    [[UINavigationController alloc] initWithRootViewController:
        listViewController];
self.window.rootViewController = navController;
[self.window makeKeyAndVisible];
```

This sets the app up to start with a navigation controller, with the new **SMListViewController** as the first thing inside.

You can compile and run at this point, and you'll see an empty table. A good start, but let's make the table show our iCloud docs! Modify **SMListViewController.h** to the following:

```
#import <UIKit/UIKit.h>
#import "SMNote.h"
#import "SMViewController.h"

@interface SMListViewController : UITableViewController

@property (strong) NSMutableArray * notes;
@property (strong) SMViewController * detailViewController;
@property (strong) NSMetadataQuery *query;

- (void)loadNotes;

@end
```

Here we've added an array to store the notes, a reference to the old view controller we created that we'll be pushing onto the stack, a metadata query we'll use to load the notes, and a `loadNotes` method we'll write later.

Next switch over to **SMListViewController.m** and synthesize properties at the top of the file:

```
@synthesize notes = _notes;
@synthesize detailViewController = _detailViewController;
@synthesize query = _query;
```

Then add the following code to the bottom of `viewDidLoad`:

```
self.notes = [[NSMutableArray alloc] init];
self.title = @"Notes";
UIBarButtonItem *addNoteItem = [[UIBarButtonItem alloc]
    initWithTitle:@"Add"
    style:UIBarButtonItemStylePlain
    target:self
    action:@selector(addNote:)];
self.navigationItem.rightBarButtonItem = addNoteItem;
```

This initializes the notes array to an empty list, sets up a title for this view controller, and adds a button to the navigation bar that says "Add". When the user taps this, the `addNote` method will be called, and we'll implement this to add a new note.

Unlike the previous project that had just one document (so always used the same filename each time), this time we're storing multiple documents (one for each note created), so we need a way to generate unique file names. As an easy solution, we will use the creation date of the file and prepend the 'Note_' string.



So add the implementation of addNote as follows:

```
- (void)addNote:(id)sender {  
  
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];  
    [formatter setDateFormat:@"yyyyMMdd_hhmss"];  
  
    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. We call the saveToURL method and, in case of success, we add the newly created note to the array which populates the table view.

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 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
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
        cell.accessoryType =
            UITableViewCellStyleDisclosureIndicator;
    }

    SMNote * note = [_notes objectAtIndex:indexPath.row];
    cell.textLabel.text = note.fileURL.lastPathComponent;

    return cell;
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        self.detailViewController = [[SMViewController alloc]
            initWithNibName:@"SMViewController_iPad" bundle:nil];
    } else {
        self.detailViewController = [[SMViewController alloc]
            initWithNibName:@"SMViewController_iPhone" bundle:nil];
    }
    SMNote * note = [_notes objectAtIndex:indexPath.row];
    self.detailViewController.doc = note;
    [self.navigationController
        pushViewController:self.detailViewController animated:YES];
}
```

Compile and run the application, and you should be able to add new notes when you tap the Add button. You can also go to the iCloud manager in settings to verify they are actually in iCloud.

But what if we quit the application and restart it? The list is empty! We need a way to load them at startup or when the application becomes active.



Loading Notes

To load notes, we'll follow a similar strategy to what we did earlier when loading a single note. However, this time we don't know the exact file name, so we have to tweak our search predicate to look for a file name like "Note_*".

So add this new method to **SMLViewController.m**:

```
- (void)loadNotes {  
    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");  
    }  
}
```

We might be tempted to place a call to this `loadNotes` method in `viewDidLoad`. That would be correct, but that is executed on initial app startup. If we want to reload data really each time the app is opened (even from the background), it's better to add an observer to listen when the application becomes active.

So add the following line of code to the end of `viewDidLoad`:

```
[[NSNotificationCenter defaultCenter] addObserver:self  
    selector:@selector(loadNotes)  
    name: UIApplicationDidBecomeActiveNotification object:nil];
```



This calls the `loadNotes` method we just wrote on startup (or when the application returns from the background). And we set up `loadNotes` to call `queryDidFinishGathering` when the metadata search completes, so add the code for that next:

```
- (void)queryDidFinishGathering:(NSNotification *)notification {  
  
    NSMetadataQuery *query = [notification object];  
    [query disableUpdates];  
    [query stopQuery];  
  
    [self loadData:query];  
  
    [[NSNotificationCenter defaultCenter] removeObserver:self  
        name:NSMetadataQueryDidFinishGatheringNotification  
        object:query];  
  
    self.query = nil;  
}
```

This is exactly the same as we added earlier in the tutorial. Next, add the implementation of `loadData` as follows (right above `queryDidFinishGathering`):

```
- (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");  
            }  
        }];  
    }  
}
```

The `loadData` method has to populate the array of notes according to the results



of the query. The implementation here fully reloads the list of notes as returned by iCloud.

One final change. For testing purposes, go ahead and add a 'refresh' button to the navigator item which triggers the `loadNotes` method when tapped, by adding this code to the bottom of `viewDidLoad`:

```
UIBarButtonItem *refreshItem = [[UIBarButtonItem alloc]
    initWithTitle:@"Refresh"
    style:UIBarButtonItemStylePlain
    target:self
    action:@selector(loadNotes)];
self.navigationItem.leftBarButtonItem = refreshItem;
```

That's it! Compile and run the app on one device, and create a few notes. Then start the app on another device, and see that the table is correctly populated with the same notes!

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.

We've just scratched the surface of iCloud - stay tuned for the next chapter, where we'll cover handling conflict resolution, using `NSFileWrapper`, storing simple key-value pairs, and using Core Data with iCloud!



Intermediate iCloud

by Cesare Rocchi

In the last chapter, you learned the basics of synchronizing your app's data with iCloud with 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, we're going to cover tons of new iCloud topics, including:

- Toggling between iCloud and local storage
- Having multi-file documents with `NSFileWrapper`
- Storing key-value information in iCloud
- Deleting files in iCloud
- Dealing with file changes and conflicts
- Using the Undo Manager with iCloud
- Exporting Data (via URLs) with iCloud
- Using 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 have hands-on experience that you can make use of in your own apps!

This chapter begins with the project that we created in the previous chapter. If you don't have this project already, you can find the code in the resources for the previous chapter.

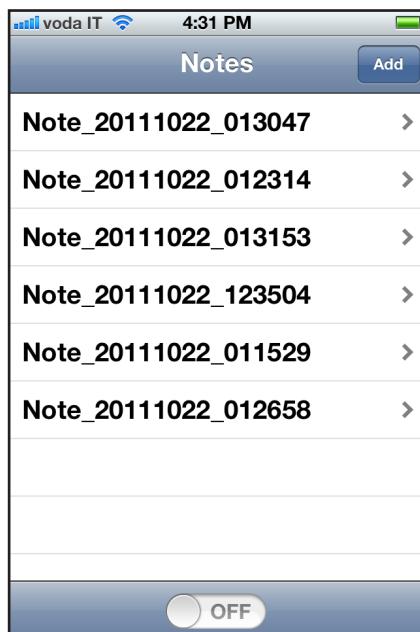


Toggling iCloud Support

So far we have assumed that our end 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 we 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 our application by just saving notes locally. Later he might change his mind and turn iCloud on. How do we cope with this case? Let's see.

First we have to update our user interface by adding a switch button that activates iCloud storage. We will place it in the toolbar at the bottom of the first view controller. The intended result is shown in the following screenshot:



To achieve this we have to refactor our code a bit.

Let's first add a boolean property to keep track when the iCloud functionality is enabled. Add the following property to **SMListViewController.h**:

```
@property (nonatomic) BOOL useiCloud;
```

Then switch to **SMListViewController.m** and synthesize the property:

```
@synthesize useiCloud = _useiCloud;
```

Next let's add a few private methods which will be needed throughout this new project (put this before the @implementation):



```
@interface SMLViewController ()  
  
- (UIView *) toolBarView;  
- (NSURL *) localNotesURL;  
- (NSURL *) ubiquitousNotesURL;  
- (void) setNotesUbiquity;  
  
@end
```

Also add a static string to use as a key for storing the switch button value (right after the @synthesize statements):

```
static NSString * const useiCloudKey = @"useiCloud";
```

Next we'll add the new methods. In the first method, toolBarView, we build a view which wraps a switch button (so we can add it to the toolbar). So add the following to the bottom of the file:

```
- (UIView *) toolBarView {  
  
    UIView *cloudView =  
        [[UIView alloc] initWithFrame:CGRectMake(0, 0, 160, 33)];  
    UISwitch *cloudSwitch =  
        [[UISwitch alloc] initWithFrame:CGRectMake(40, 4, 80, 27)];  
    cloudSwitch.on = _useiCloud;  
  
    [cloudSwitch addTarget:self  
        action:@selector(enableDisableiCloud:)  
        forControlEvents:UIControlEventValueChanged];  
  
    [cloudView addSubview:cloudSwitch];  
  
    return cloudView;  
}
```

Note that this method also registers a callback for when the switch value changes. Let's implement this callback next:

```
- (void) enableDisableiCloud: (id) sender {  
    self.useiCloud = [sender isOn];  
}
```

This way the value of the boolean property is always in synch with its visual counterpart.

Next, add the following code to the beginning of viewDidLoad to add the switch to the toolbar and initialize the useiCloud boolean:



```

_useiCloud =
[[NSUserDefaults standardUserDefaults] boolForKey:_useiCloudKey];

UIBarButtonItem *flexSpace1 = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
    target:nil action:NULL];

UIBarButtonItem *iCloudSwitchItem = [[UIBarButtonItem alloc]
    initWithCustomView:[self toolbarView]];

UIBarButtonItem *flexSpace2 = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
    target:nil action:NULL];

self.toolbarItems = [NSArray arrayWithObjects:
    flexSpace1, iCloudSwitchItem, flexSpace2, nil];

```

When the view is loaded we retrieve the value previously selected by the user and assign it to the boolean (we'll show how to store it later). To make the switch button centered we surround it with two flexible spaces (flexSpace1 and flexSpace2).

The navigation controller's toolbar is hidden by default, so we have to make it visible in `viewWillAppear`:

```

- (void) viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [self.navigationController setToolbarHidden:NO];
}

```

The last step for our first milestone is to persist the value of the switch button. We 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];
    }
}

```

If we compile now the switch button is correctly shown at the bottom and 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 UI 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, we will start with the case where the user starts working on notes without synching to iCloud.

You might remember we declared some private methods in `SMLViewController.m` that we haven't implemented yet:

- **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 we 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 we 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 we 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 `SMLViewController.m`:

```
- (NSURL *) localNotesURL {
    return [[[NSFileManager defaultManager]
        URLsForDirectory:NSDocumentDirectory
        inDomains:NSUserDefaultsDomainMask] 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 we are already familiar with.

So let's get back to our use case. The user starts the application, iCloud is off and the list of notes is empty. In the last chapter we already implemented a method to



add a note. We just have to modify it a bit to cover the iCloud-off case. To do this, we change the base url according to the value of the `_useiCloud` boolean property.

By default we initialize it as local and we change it if iCloud is enabled. The rest of the method is the same of the previous project.

```
- (void)addNote:(id)sender {  
  
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];  
    [formatter setDateFormat:@"yyyyMMdd_hhmss"];  
  
    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 we can do a first test. Run the application, make sure iCloud storage is off, and add a note as in the following screenshot:





Then check your iCloud storage on your device (via Settings/iCloud/Storage & Back-up/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's our file go?

The local file is still there but in the `loadNotes` method we are searching for iCloud documents, whether iCloud is enabled or not by the switch. We need to rework the method to cover the iCloud-off case.

So modify `loadNotes` to cover both cases as shown below:

```
- (void)loadNotes {
    NSURL *ubiq = [[NSFileManager defaultManager]
                    URLForUbiquityContainerIdentifier:nil];
    if (ubiq && _useiCloud) { // iCloud is on (from previous chapter)

        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:NSNotificationCenterMetadataQueryDidFinishGatheringNotification
            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 this case the retrieval is not asynchronous so we can just cycle over the returned array, build instances of notes according to the array and reload the content of the table view.

Compile and run the application, and this time you should see the note you created earlier in local storage (not anything currently in iCloud!)

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 **)errorOut;`

The idea is that we 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 we want to avoid!

To resolve this problem, the documentation suggests to perform this operation on a secondary thread. Here we have two options: using a GCD dispatch queue or an NSOperationQueue. We cover using GCD many other places within this book, so to switch things up we'll use NSOperationQueue this time.

In essence we 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 synch with data on the cloud. We 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 we instantiate a NSOperationQueue and we 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, we have to assign a different folder according to the value of the `_useiCloud` boolean. Then we cycle over the list of notes and for each one we call the `NSFileManager` method described above.

We should remember to notify the main thread that the operation on the secondary thread is complete. For now, we'll just have the callback 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` we 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 app's folder):

- file://localhost/var/mobile/Applications/75C93597-DD34-4F33-BE1D-DA092EB-CF1E2/Documents/Note_20111022_063557

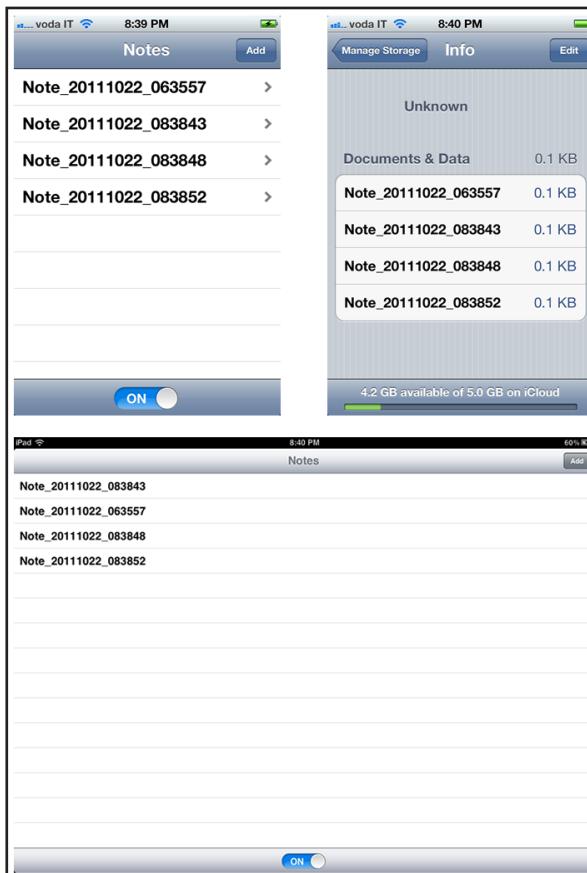
A document synched with iCloud has a different url (outside of the app's directory):

- file://localhost/private/var/mobile/Library/Mobile%20Documents/D2CD3DR3GP~com~studiomagnolia~dox/Documents/Note_20111022_063557

Compile 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 you'll see the notes you created locally are correctly synchronized!



But what happens if we 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 our 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 **SMLViewController.h** to mark the interface as implementing the **UIAlertViewDelegate** protocol, and add a property for the switch:

```
@interface SMLViewController : UITableViewController  
    <UIAlertViewDelegate>  
  
    @property (strong) UISwitch *cloudSwitch;
```



Then switch to **SMLViewController.m** and synthesize the property:

```
@synthesize cloudSwitch;
```

Also modify toolBarView to set the property as follows:

```
- (UIView *) toolBarView {
    UIView *cloudView =
        [[UIView alloc] initWithFrame:CGRectMake(0, 0, 160, 33)];
    cloudSwitch =
        [[UISwitch alloc] initWithFrame:CGRectMake(40, 4, 80, 27)];
    // Rest of method...
}
```

Next add a new method to wrap the "migration" procedure, which will be performed on a secondary thread (add this above the setUseiCloud method):

```
- (void) startMigration {
    NSOperationQueue *iCloudQueue = [NSOperationQueue new];
    NSInvocationOperation *oper = [[NSInvocationOperation alloc]
        initWithTarget:self
        selector:@selector(setNotesUbiquity)
        object:nil];
    [iCloudQueue addOperation:oper];
}
```

Then we 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:@"Yes", nil];
        }
    }
}
```



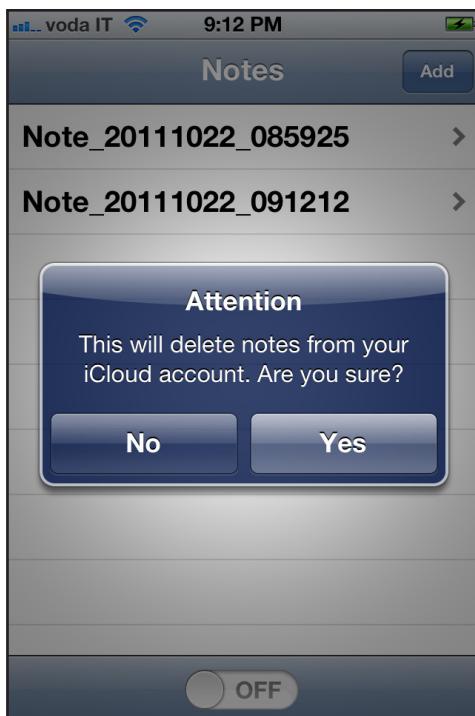
```
    [iCloudAlert show];  
}  
else {  
    [self startMigration];  
}  
}  
}  
}
```

Finally we 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!)

Now's a good time to take a break because we'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 we 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 your user wants to delete your app's data using the iCloud storage editor in Settings, but your app's list is full of tons of weirdly named small files. That might be quite annoying to have to delete each one individually!

In this part we are going to create an alternate version of our 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.

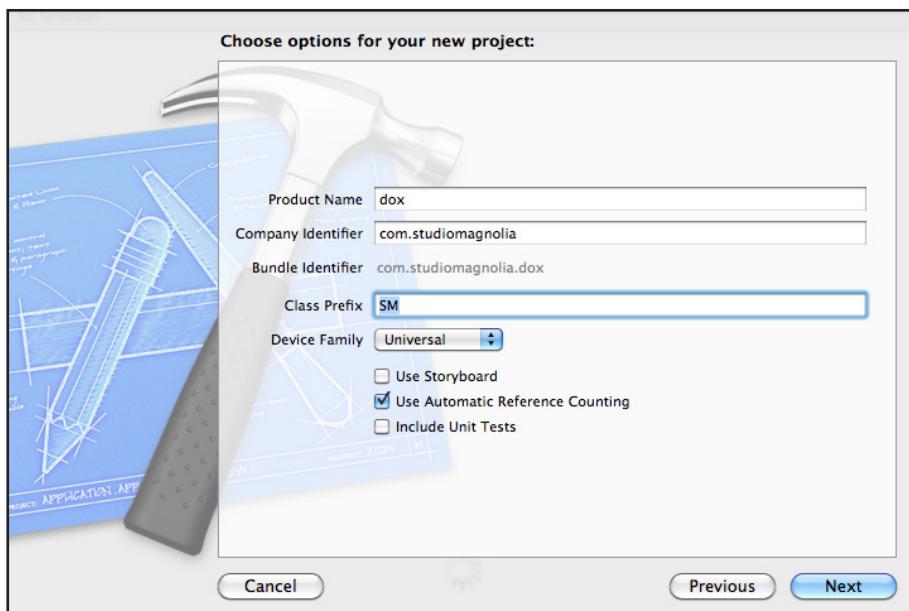


Rather than refactoring our existing app, it will be easier to create a new app from scratch. Be sure to put your current project somewhere safe, because we'll be coming back to it later on in this chapter.

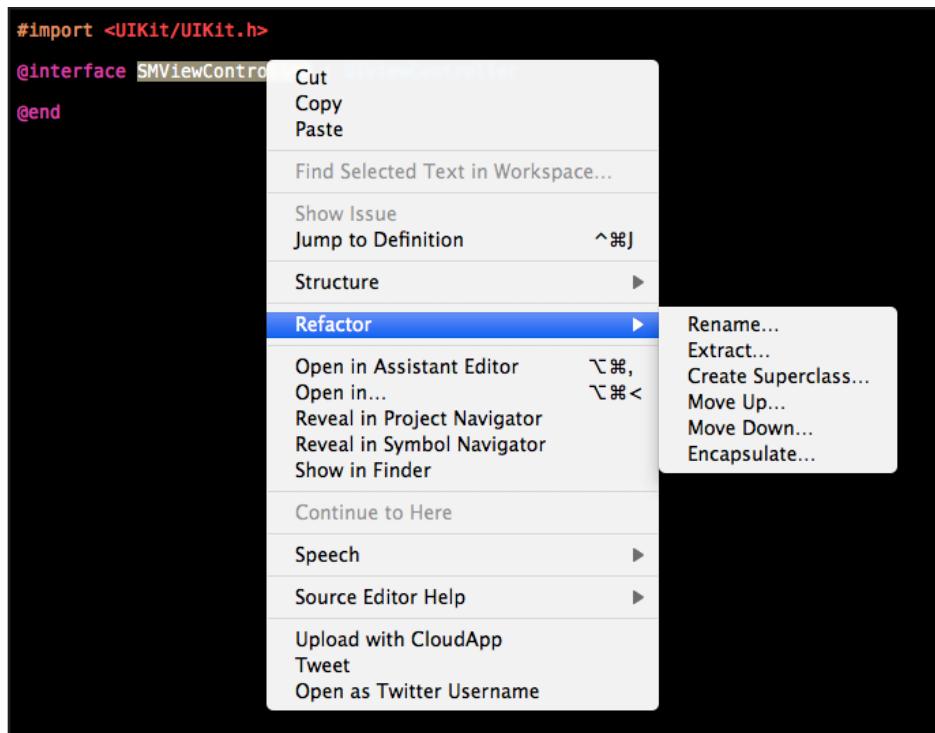
So let's get started by building the skeleton of the application.

As previously there will be two view controllers, one to show the list of notes and one to show and edit a single note. There will be also a few notifications to be posted and listened for, but first let's build the user interface.

Create a new project with the **iOS\Application\Single View Application** template. Name the new application **dox**, set the Class Prefix to **SM**, set the Device Family to **Universal**, and enable ARC (but not Storyboards). Again, be sure to save in a different directory so you don't overwrite your old project, because you'll need it later on.



First let's rename the automatically generated view controller to `SMNoteListViewController`. Select the header file, highlight the class name and control-click it. Select Refactor\Rename, and use the refactor wizard to automatically rename both file names and references.



As previously we have to create a navigation based application, so modify **SAppDelegate.h** as follows:

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

@interface SAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic)
    UINavigationController *navigationController;

@end
```

Instead of a simple view controller we use a UINavigationController. To finish the switch, modify **SAppDelegate.m** as follows:

```
#import "SAppDelegate.h"
#import "SMNoteListViewController.h"

@implementation SAppDelegate

@synthesize window = _window;
@synthesize navigationController = _navigationController;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
```



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

SMNoteListViewController *masterViewController = nil;

if ([[UIDevice currentDevice] userInterfaceIdiom] ==
    UIUserInterfaceIdiomPhone) {

    masterViewController = [[SMNoteListViewController alloc]
        initWithNibName:@"SMNoteListViewController_iPhone" bundle:nil];

} else {

    masterViewController = [[SMNoteListViewController alloc]
        initWithNibName:@"SMNoteListViewController_iPad" bundle:nil];

}

self.navigationController = [[UINavigationController alloc]
    initWithRootViewController:masterViewController];
self.window.rootViewController = self.navigationController;

[self.window makeKeyAndVisible];
return YES;
}

// Rest of file...

@end
```

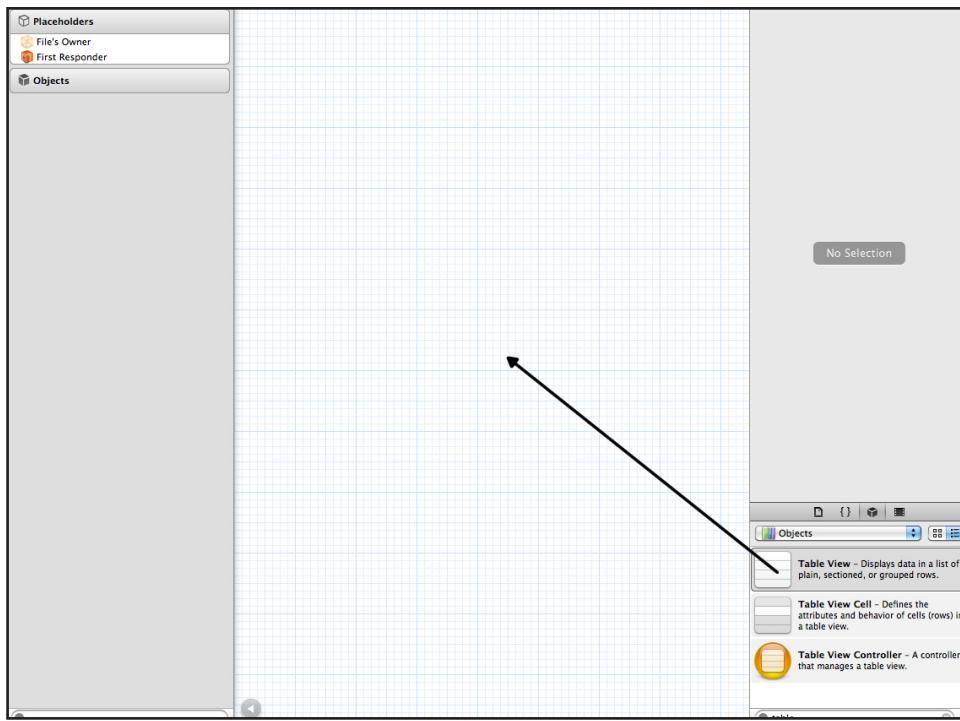
This way the controller is correctly instantiated according to the device. Now the first view controller to manage the application is a navigation controller, which enables the pushing of the single note view controller when a note is tapped.

The next step is to make our root controller a table view controller, so change the interface declaration in **SMNoteListViewController.h** as follows.

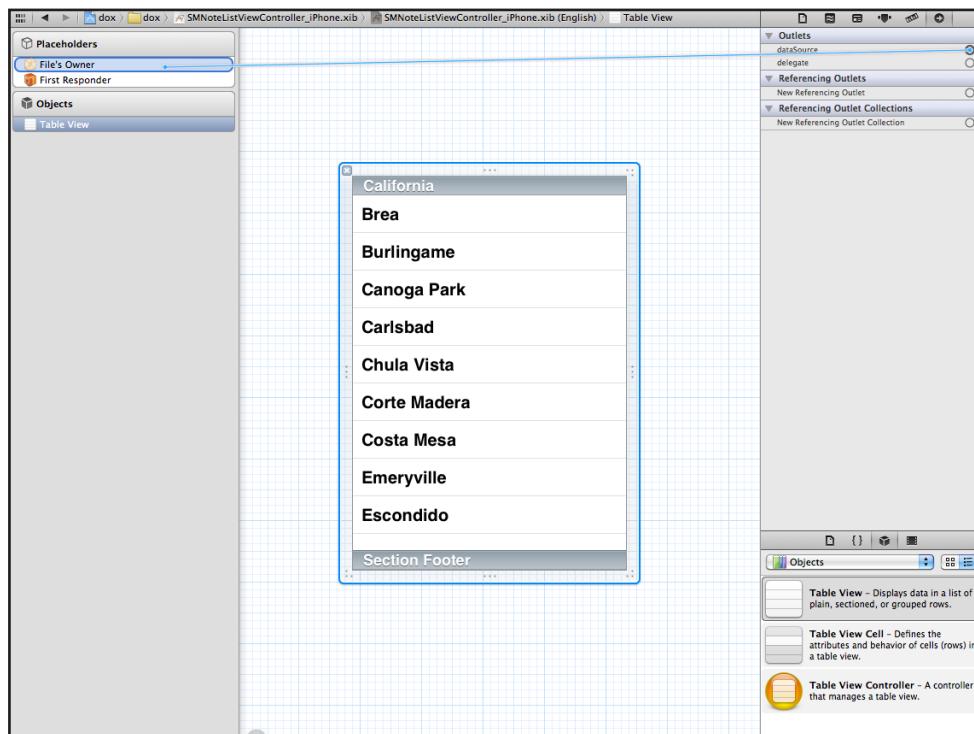
```
@interface SMNoteListViewController : UITableViewController
```

Then open **SMNoteListViewController_iPhone.xib**, delete the automatically inserted view and we drag on the stage a table view component:



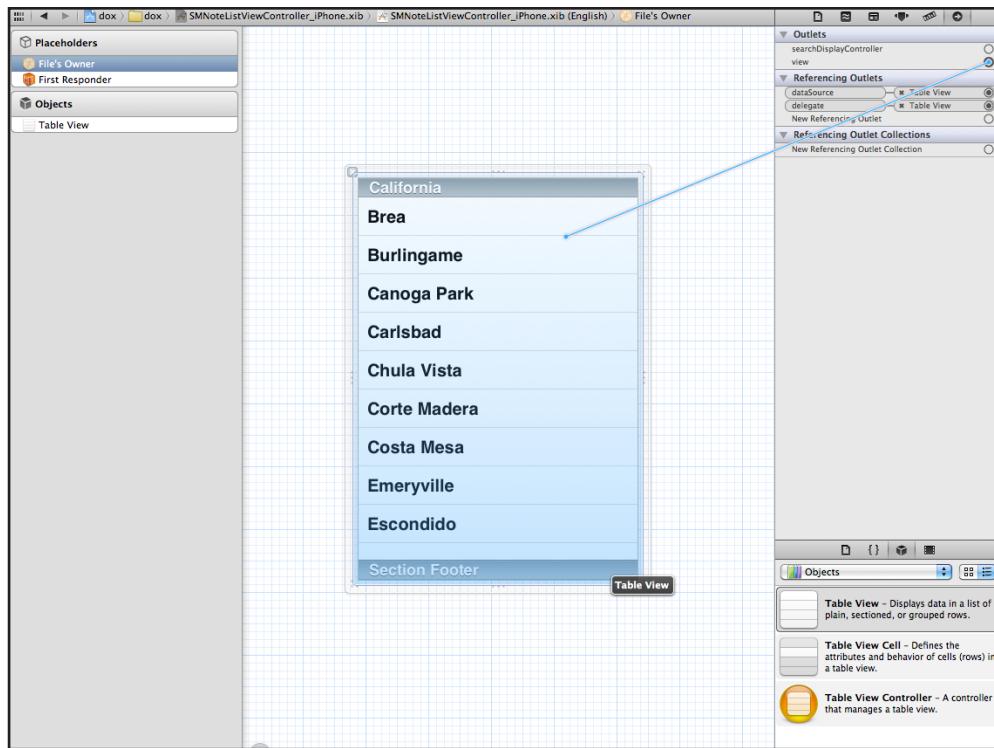


Now we have to hook up the connections. Select the table view, bring up the Connections Inspector (last tab in the sidebar) and drag a line from the dataSource and delegate outlets to File's Owner:



Similarly, select File's Owner and drag a line from the view outlet to the table view:





Finally, repeat this same process for **SMNoteListViewController_iPad.xib**.

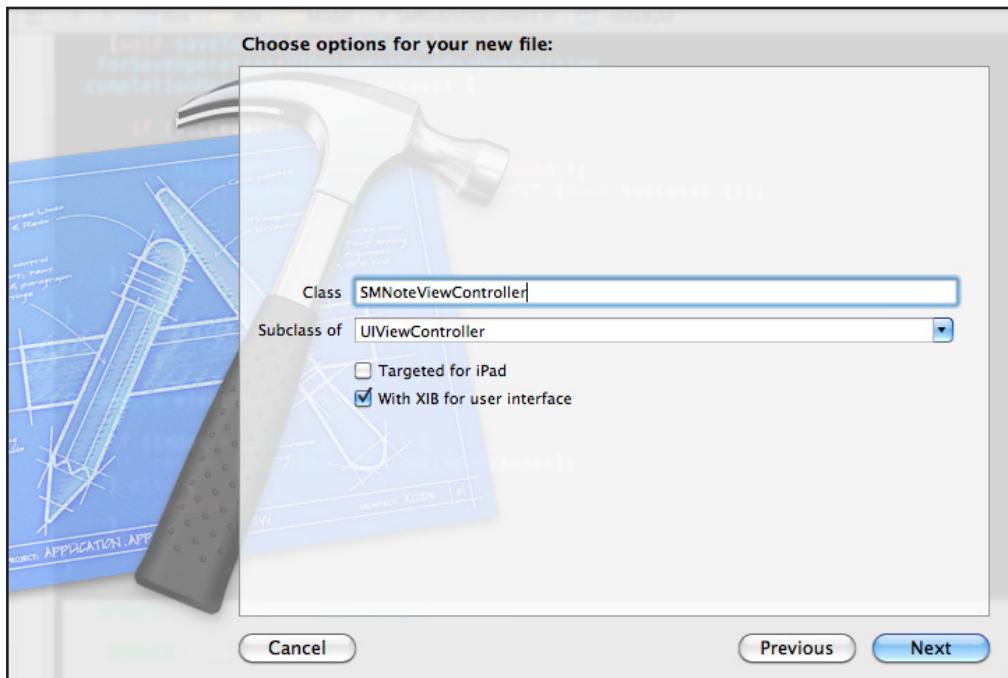
Compile and run, and you'll see an empty table as shown below:



The user interface for the single note view controller is exactly the same of previous projects. Create a new file with the **iOS\Cocoa Touch\UIViewController sub-**



class template, name it **SMNoteViewController**, make it a subclass of **UIViewController**, and make sure **With XIB for user interface** is selected.



Modify **SMNoteViewController.h** to mark the view controller as implementing the **UITextViewDelegate** and having a **UITextView** outlet:

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

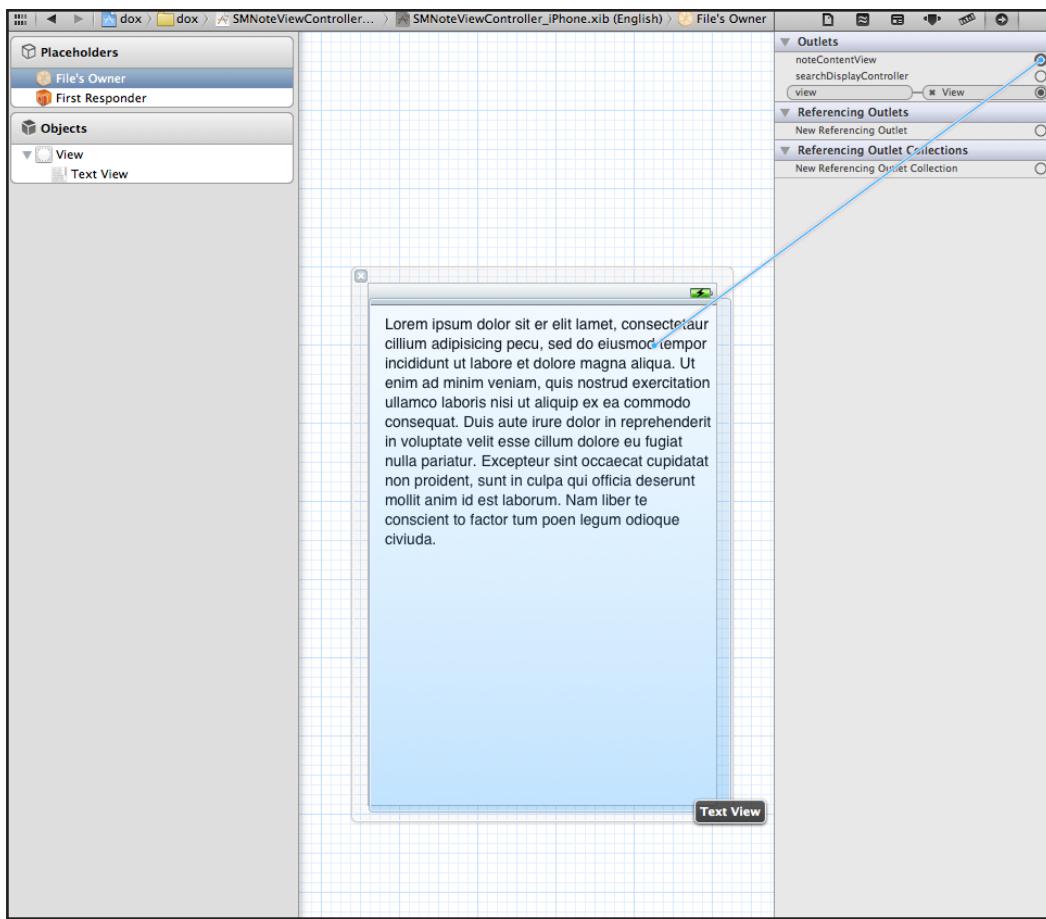
@interface SMNoteViewController : UIViewController
<UITextViewDelegate>

@property BOOL isChanged;
@property (strong, nonatomic) IBOutlet UITextView *noteContentView;

@end
```

Next rename the xib file to **SMNoteViewController_iPhone.xib**. Then open it up and add a text view to the view, and link the text view to its corresponding outlet as shown below:





The class generator in XCode has created just one xib optimized for the iPhone. Let's add the one for the iPad.

Create a new file with the **iOS\User Interface\Empty** template. Select **iPad** for the device family, and save it as **SMNoteViewController_iPad.xib**.

The xib for the iPad is empty so let's populate it. Unlike xibs generated during the creation of a UIViewController, this one has doesn't have a class associated for the File's Owner. So we have to declare which view controller will manage this view. Open the xib, select its 'File's Owner' and open the inspector on the right.

On the third tab we can specify the name of the class which will manage the xib. We enter SMNoteViewController. We then perform the following steps:

- Add a view, and connect it to the File's Owner's view outlet
- Add a text view to the view, and connect it to the File's Owner's noteContentView outlet

For the moment the work on the user interface is complete. Let's move to add some iCloud functionality!



Modeling the Custom Document

Now it's time to focus on the way we model data. Unlike previous projects here we 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 we can encode and decode information into a data buffer. For each note we will keep track of the following properties:

- id
- content
- creation date
- update date

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template, name the class **SMNote** and make it a subclass of **NSObject**. Then replace **SMNote.h** with the following:

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

@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

@synthesize noteId = _noteId;
@synthesize noteContent = _noteContent;
@synthesize createdAt = _createdAt;
@synthesize updatedAt = _updatedAt;

- (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 synthesizes the properties and 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 we'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**.

So again, 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 {  
  
    NSMutableArray *_entries;  
    NSFileWrapper *_fileWrapper;  
  
}  
  
@property (nonatomic, strong) NSMutableArray *entries;  
@property (nonatomic, strong) NSFileWrapper *fileWrapper;  
  
- (NSInteger) count;  
- (void) addNote:(SMNote *) note;  
- (SMNote *) entryAtIndex:(NSUInteger)index;  
  
@end
```

Here we create an `NSFileWrapper` so we 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  
  
@synthesize entries = _entries;  
@synthesize fileWrapper = _fileWrapper;  
  
- (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 we synthesize properties and override the `initWithFileURL:` method to initialize the array of entries, and we 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 we 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;
    }
}

- (NSUInteger ) count {
    return self.entries.count;
}
```

The entryAtIndex: method will be needed when we 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 we'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 we are left with the two override points to manage reading and writing on iCloud. This is the crucial step for this application.



Unlike the previous chapter, where we stored just a string, here we have a more complex data model: an array of SMNote instances. To encode this set of objects we 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 our 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 we 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 we have enough information to build the method to generate our 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 we 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 we'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 we 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 we 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. iCloud requires the application to register its custom document type in the Info.plist file.

In this section we are going to create a new UTI (Uniform Type Identifier) for our app's custom data format. An UTI is a string that uniquely identifies a class of objects (in our case an array of notes), which we 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 we can add new metadata for the application. Select **Document types**, as shown below:



Key	Type
Localization native development region	String
Bundle display name	String
Executable file	String
▶ Icon files	Array
Bundle identifier	String
InfoDictionary version	String
Bundle name	String
Bundle OS Type code	String
Bundle versions string, short	String
Bundle creator OS Type code	String
Bundle version	String
Application requires iPhone environment	Boolean
▶ Required device capabilities	Array
▶ Supported interface orientations	Array
▶ Supported interface orientations (iPad)	Array
Document types	String
▶ Document types	Dictionary
Environment variables	String
Executable architectures	String
Executable file	String
Exported Type UTIs	String
File quarantine enabled	Boolean
Fonts provided by application	String
Get Info string	String
Help Book directory name	String
Help Book identifier	String

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)	Diction...	(4 items)
Document is a package or bundle	Boolean	YES
▼ Document Content Type UTIs	Array	(1 item)
Item 0	String	com.studiomagnolia.dox.package
Document Type Name	String	Dox Package
Handler rank	String	Owner

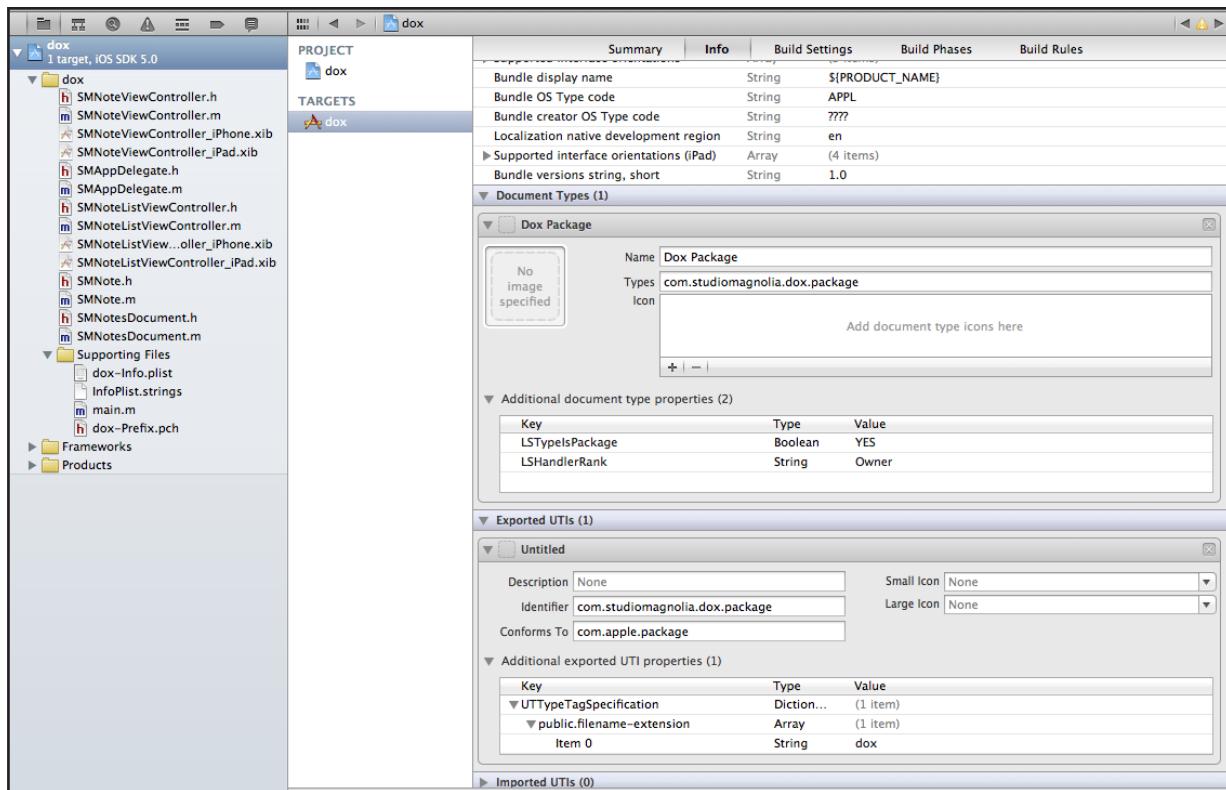
Now you might think we are done, but we are not! It is not enough to declare a new UTI, we have also to export it. We have to add another property to the plist file, called 'Exported Type UTIs'. Essentially we have to recreate a structure as in the following picture:

▼ Exported Type UTIs	Array	(1 item)
▼ Item 0 (com.studiomagnolia.dox.package)	Diction...	(3 items)
▼ Conforms to UTIs	Array	(1 item)
Item 0	String	com.apple.package
Identifier	String	com.studiomagnolia.dox.package
▼ Equivalent Types	Diction...	(1 item)
▼ public.filename-extension	Array	(1 item)
Item 0	String	dox

The identifier has to be the same provided above and 'dox' will be the extension of our file.

We should also check that all the information entered in the plist are replicated in the info section of our 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 we 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 we will follow an approach similar to last chapter. Replace **SMNoteListViewController.h** with the following:

```
#import "SMNote.h"
#import "SMNotesDocument.h"

@class SMNoteViewController;

@interface SMNoteListViewController : UITableViewController {
    SMNotesDocument *document;
    NSMetadataQuery *_query;
}

@property (strong, nonatomic)
    SMNoteViewController *detailViewController;
@property (strong, nonatomic) SMNotesDocument *document;

- (void) loadNotes;
- (void) loadData:(NSMetadataQuery *)query;
```



```
@end
```

The document property is a reference to the SMNotesDocument instance that manages the array of notes and iCloud functionality.

Next switch to **SMNoteListViewController.m** and synthesize the properties:

```
@synthesize detailViewController = _detailViewController;
@synthesize document;
```

Also modify viewDidLoad to add a button to add a new note to the toolbar, and register callbacks for when notes get reloaded or the application becomes active:

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

    UIBarButtonItem *addNoteItem = [[UIBarButtonItem alloc]
        initWithTitle:@"Add"
        style:UIBarButtonItemStylePlain
        target:self
        action:@selector(addNote:)];

    self.navigationItem.rightBarButtonItem = addNoteItem;

    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(loadNotes)
        name:UIApplicationDidBecomeActiveNotification
        object:nil];

    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(notesLoaded:)
        name:@"com.studiomagnolia.notesLoaded"
        object:nil];
}

}
```

Next add the implementation of addNote:, which is pretty trivial:

```
- (void)addNote:(id)sender {

    SMNote *doc = [[SMNote alloc] init];
    doc.noteContent = @"Test";
    [document addNote:doc];
}
```



This simply creates a new SMNote with some default content and adds it to the SMNotesDocument.

The selector triggered when notes are reloaded is simple as well:

```
- (void) notesLoaded:(NSNotification *) notification {  
  
    document = notification.object;  
    [self.tableView reloadData];  
  
}
```

This notification passes the document as a parameter, so we store it in our instance variable and reload the table view to display the new data.

The loadNotes method, triggered when the application becomes active, is defined as follows.

```
- (void)loadNotes {  
  
    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:NSNotificationCenterQueryDidFinishGatheringNotification  
            object:query];  
  
        [query startQuery];  
  
    } else {  
        NSLog(@"No iCloud access");  
    }  
}
```

You should be pretty familiar with this code. We check if iCloud is available and we run a query to look for a file in the cloud. Before we 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:NSMetadataItemURLKey];  
  
        SMNotesDocument *doc =  
            [[SMNotesDocument alloc] initWithFileURL:url];  
        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];
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 Entitlements checkbox to enable iCloud. The default entries will be fine. You should be able to use the same provisioning profile we created in the last chapter, since your project name (hence bundle ID) should be the same.

Compile 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 we run the application. The notes array is obviously empty.

Now we have to just to integrate the mechanism to render the view correctly according to the iCloud document. Add the following code to **SMNoteListViewController.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";
}
```

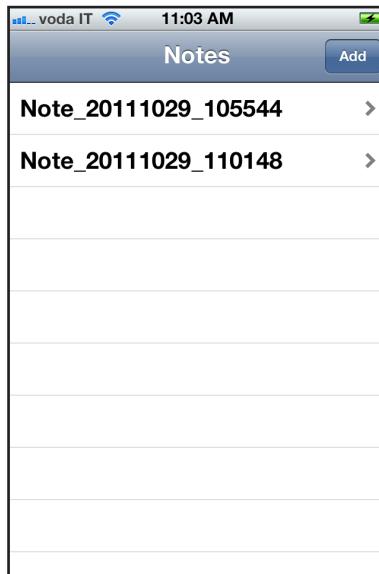


```
SMNote *n = [self.document entryAtIndex:indexPath.row];

UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:
        UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone) {
        cell.accessoryType =
        UITableViewCellAccessoryDisclosureIndicator;
    }
}

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 we have created multiple notes they are all wrapped in a single file.



We 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 we are left with the last step: to show the content of a single note and save it when it is edited. First import the header for the detail view at the top of **SMNoteListViewController.m**:

```
#import "SMNoteViewController.h"
```

Then add the code to present the detail view when a note is tapped:



```

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    SMNote *n = [self.document entryAtIndex:indexPath.row];

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone) {
        if (!self.detailViewController) {
            self.detailViewController = [[SMNoteViewController alloc]
                initWithNibName:@"SMNoteViewController_iPhone"
                bundle:nil];
        }
    } else {

        if (!self.detailViewController) {
            self.detailViewController = [[SMNoteViewController alloc]
                initWithNibName:@"SMNoteViewController_iPad"
                bundle:nil];
        }
    }

    self.detailViewController.currentNote = n;
    [self.navigationController pushViewController:
        self.detailViewController animated:YES];
}

```

When the user taps a note on the table we push a new view controller, with the XIB depending on the device we're running on.

We should remember to update **SMNoteViewController.h** with a new property called `currentNote`:

```

// Before the @interface
@class SMNote;

// In the property section
@property (strong, nonatomic) SMNote *currentNote;

```

And import the header and synthesize the property in **SMNoteViewController.m**:

```

// At top of file
#import "SMNote.h"

// After @implementation
@synthesize isChanged;

```



```
@synthesize noteContentView;
@synthesize currentNote;
```

This will be a reference to the single note opened in the view.

Next modify viewDidLoad to setup the delegate for the text view and initialize isChanged to NO:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.noteContentView.delegate = self;
    isChanged = NO;
}
```

The only method triggered by the text view that we override is textViewDidChange: which updates the boolean value accordingly:

```
- (void)textViewDidChange:(UITextView *)textView
{
    isChanged = YES;
}
```

When the view is going to appear we set the content of the text view:

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    if (self.currentNote) {
        self.noteContentView.text = self.currentNote.noteContent;
    }
}
```

When the user closes the view we update the content of the note and post a "note changed" notification:

```
- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
    if (isChanged) {
        self.currentNote.noteContent = self.noteContentView.text;
        [[NSNotificationCenter defaultCenter]
            postNotificationName:@"com.studiomagnolia.noteChanged"
            object:nil];
    }
}
```

This notification will be caught by the UIDocument subclass which will trigger the save procedure.



We are done again! We 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.

That concludes this topic, so it's time for another break if you need one! Next up, we'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 our 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 is just 64KB!

One way that we could use the key-value in our app is to keep track of the current note we'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'd like to pick it up on the iPad later but you don't remember the note title or id.

If we kept track of the current note we're editing in the key-value store, the app could look this up when you open the app and start the user where he left off. This is exactly the feature that we are going to build next!

In this case we will store a simple and small piece of information: the id of the last edited note. In general the iCloud key-value store is meant for this kind of usage.

The class which 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 our 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:store];
```

The pattern to work with key-value stores is pretty simple:

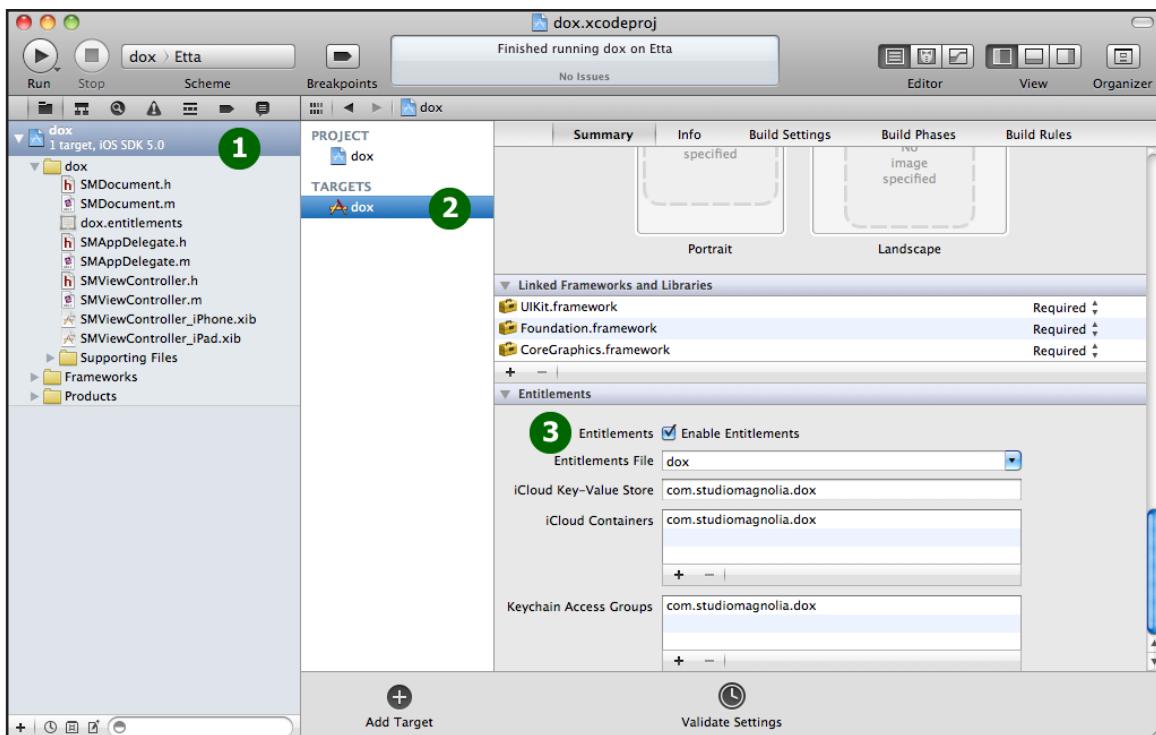
- Register for updates from the iCloud key-value store
- React accordingly in the callback

So let's update our app to work with the key-value store. Here's the changes we'll make:

- Store a key-value entry when a single note is displayed and the user closes an application
- Listen for key-value changes in `SNNoteListViewController`
- When notified, push a new view controller showing the note stored as the last edited

Before writing any code we 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 **SMNoteViewController.m**. Add the following code to the bottom of `viewWillAppear:`:

```
[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(saveNoteAsCurrent)
    name:UIApplicationDidEnterBackgroundNotification
    object:nil];
```

Here we start listening to see if the application enters the background when the user is editing a note. Next add this code to the bottom of `viewWillDisappear` to remove the observer:

```
[NSNotificationCenter defaultCenter]
    removeObserver:self
    name:UIApplicationDidEnterBackgroundNotification
    object:nil];
```

Next implement the `saveNoteAsCurrent` callback:

```
- (void) saveNoteAsCurrent {
    [[NSUbiquitousKeyValueStore defaultStore]
        setObject:self.currentNote.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 we will use the very same key to look up the value later.

Now let's modify **SMNoteListViewController.m**. Since we are going to update the view upon a key-value notification it is better to listen only after we have retrieved the freshest list of notes. So add the following to the bottom of notesLoaded:

```
NSUbiquitousKeyValueStore* store =
    [NSUbiquitousKeyValueStore defaultStore];
[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(updateCurrentNodeIfNeeded:)
    name:NSUbiquitousKeyValueStoreDidChangeExternallyNotification
    object:store];
[store synchronize];
```

When we receive an update notification, we'll behave as if the user had tapped the cell corresponding to the note with that identifier. We will retrieve the note using the key and we push an instance of SMNoteViewController, as shown below:

```
- (void) updateCurrentNodeIfNeeded:(NSNotification *) notification {

    NSString *currentNoteId =
        [[NSUbiquitousKeyValueStore defaultStore] stringForKey:
            @"com.studiomagnolia.currentNote"];
    SMNote *n = [document noteById:currentNoteId];

    if ([[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone) {
        if (!self.detailViewController) {
            self.detailViewController = [[SMNoteViewController alloc]
                initWithNibName:@"SMNoteViewController_iPhone"
                bundle:nil];
        }
    } else {
        if (!self.detailViewController) {
            self.detailViewController = [[SMNoteViewController alloc]
                initWithNibName:@"SMNoteViewController_iPad"
                bundle:nil];
        }
    }

    self.detailViewController.currentNote = n;
    [self.navigationController pushViewController:self.
        detailViewController animated:YES];
}
```



We 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!

How To Delete a File

We have already 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, we can use the standard method `removeItemAtURL:error:`:

```
NSError *err;
[[NSFileManager defaultManager] removeItemAtURL:[doc 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 our application.



We will start from a basic version of the project similar to the one we worked on earlier. To simplify things we can get rid on the loading of the single document, the refresh and the switch button. So our starting point is an iCloud enabled project which stores notes in separate files.

For your convinience I created this for you. Check in the chapter resources and open the **dox4-Starter** project. You'll have to change your Bundle identifier in the info.plist, your Entitlements in your Target Summary tab, and your code signing certificate to match the info you set up earlier.

Go ahead 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.

Next open **SMListViewController.m**, and add the following code to the end of viewDidLoad to add a button to the navigation bar to switch the table to editing mode:

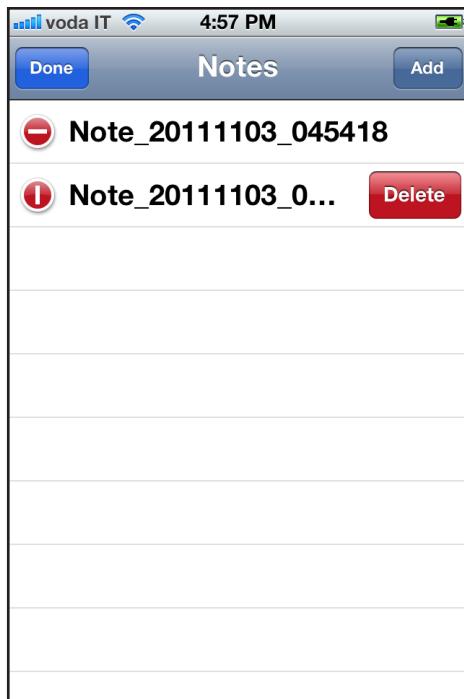
```
self.tableView.allowsSelectionDuringEditing = YES;
self.navigationItem.leftBarButtonItem = [self editButtonItem];
```

Next we'll override a method from the UITableViewDelegate protocol, tableView:editingStyleForRowAtIndexPath:, which allows specifying how a cell has to appear in editing mode. Implement it to return a built-in constant UITableViewCellStyleDelete:

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

Compile and run the application now, and you'll already see some progress. On the left you'll see an 'edit' button. If you tap it, you can try to delete rows by using the red button.





Of course, if you tap the delete button, nothing happens but we're getting close!

We just have to implement one more method: `tableView:commitEditingStyle: forRowAtIndexPath:`. Here we define what happens to the cell that the user has chosen to delete.

The selected note has to be deleted from three places: the table view, the array which populates the table and the file system. Here is the implementation:

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

    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 we 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 let's 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 we 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 we handle that in real-time? Yes indeed - and that's the feature that we 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 we 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 which is 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 we 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 SMLViewController to be ready to react to a deletion. The first step we want to achieve is to reload the list of notes in the table in case of change.

Open up **SMLViewController.m** and add the following to the bottom of viewDidLoad to listen for an document change notification:

```
[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(noteHasChanged:)
    name:UIDocumentStateChangedNotification
    object:nil];
```

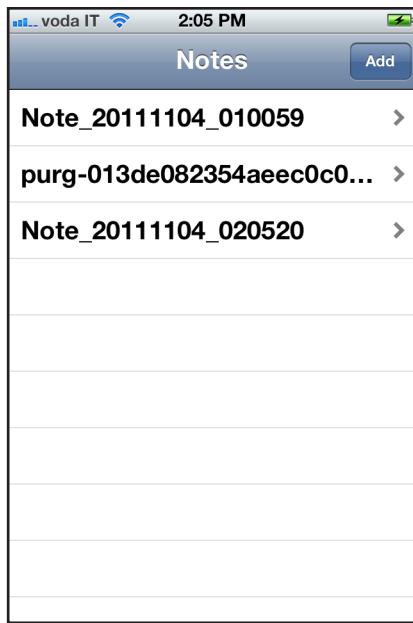
It is important to notice that this notification is posted *whenever* there is a change, including when you open a file or when you add a new note.

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 we can run the application on the two devices, verify the same list of documents appear, and delete a document on one of the devices. After a few second we should see the following:





The note has been deleted from the cloud and we have a sort of temporary version locally, without the previous id and file url. If we tap the note we 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-E25E258309F8-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, we 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 we can create a new document with the content attached to the 'limbo note'.

To achieve this we have to refactor **SMViewController.h** a bit. We have to add a new property (`isReinstated`) and a `UIBarButtonItem` to trigger the action. Here is the new header file.

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

@interface SMViewController : UIViewController <UITextViewDelegate>

@property (strong) SMNote *currentNote;
@property (strong) IBOutlet UITextView *noteContentView;
@property BOOL isChanged;
@property BOOL isReinstated;
```



```
@property (strong, nonatomic) UIBarButtonItem *reinstateNoteButton;  
@end
```

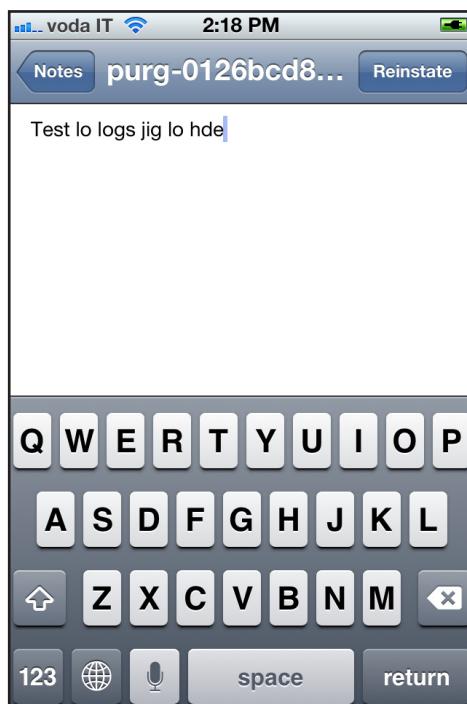
Next switch to **SMViewController.m** and synthesize the properties:

```
@synthesize isReinstated;  
@synthesize reinstateNoteButton;
```

Also add this code to the bottom of viewWillAppear to show a button in the navigation bar when the note is in a UIDocumentStateSavingError state:

```
isReinstated = NO;  
isChanged = NO;  
  
if (self.currentNote.documentState == UIDocumentStateSavingError) {  
    self.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 we haven't implemented the callback yet, so let's do that next.

The callback method is pretty similar to the one used to create a new note. In fact, we have to instantiate a new note with a new url and save it on disk. If the save operation is successful we can hide the 'Reinstate' button and set the property `isReinstated` to YES.

```
- (void) reinstateNote {

    // Generate new filename for note based on date
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"yyyyMMdd_hhmss"];

    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.noteContentView.text;
    self.reinstateNoteButton.enabled = NO;

    [n saveToURL:[n fileURL] forSaveOperation:UIDocumentSaveForCreating
        completionHandler:^(BOOL success) {
        if (success) {
            self.currentNote = n;
            isReinstated = YES;
            self.navigationItem.rightBarButtonItem = nil;
        } else {
            self.reinstateNoteButton.enabled = YES;
        }
    }];
}

}
```

The last step is to trigger the reload of data from iCloud when we switch back to the table view. This is needed only when there is a reinstated note. So add the following to the bottom of `viewWillDisappear`:

```
if (isReinstated) {
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"com.studiomagnolia.noteReinstated"
        object:self];
```



```
}
```

If the note has been reinstated we post a notification. We have now to catch it in **SMLViewController.m**. Add the following to the bottom of viewDidLoad:

```
[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(loadNotes)
    name:@"com.studiomagnolia.noteReinstated"
    object:nil];
```

This way, whenever a note is reinstated, the application loads a fresher list of notes from iCloud.

Since we have two view controllers, we have to manage the "note has been deleted" scenario also in SMViewController. Add the following to the bottom of viewDidLoad in **SMViewController.m**:

```
[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(noteHasChanged:)
    name:UIDocumentStateChangedNotification
    object:nil];
```

The selector is pretty similar to the previous one: we change the title and we show the "Reinstate" button.

```
- (void)noteHasChanged:(id)sender {

    if (self.currentNote.documentState == UIDocumentStateSavingError) {
        self.title = @"Limbo note";
        self.reinstateNoteButton = [[UIBarButtonItem alloc]
            initWithTitle:@"Reinstate"
            style:UIBarButtonItemStylePlain
            target:self
            action:@selector(reinstateNote)];

        self.navigationItem.rightBarButtonItem = reinstateNoteButton;
    }
}
```

Compile 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!

Now that we know how to create and delete notes, let's see how to handle changes.



Handling Changes

So far we took care of situations where files are created or deleted. But we also need to handle the case where the content of a file changes. In particular, what happens if we are editing a note on two devices at about the same time?

In our current implementation, nothing happens because although we observe for document changes, we filter for only the `UIDocumentStateSavingError` case.

In the scenario of concurrent editing there is no save error, for 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 our current implementation we are already listening for changes to the `documentState` property. Add the following code to the bottom of the `noteHasChanged` method in **SMViewController.m** as follows:

```
if (self.currentNote.documentState == UIDocumentStateEditingDisabled) {  
    self.navigationItem.leftBarButtonItem.enabled = NO;  
    NSLog(@"document state is UIDocumentStateEditingDisabled");  
}  
  
if (self.currentNote.documentState == UIDocumentStateNormal) {  
    self.navigationItem.leftBarButtonItem.enabled = YES;  
    NSLog(@"old content is %@", self.noteContentView.text);  
    NSLog(@"new content is %@", self.currentNote.noteContent);  
}
```



Now run the app on two devices, and make sure the second is connected to XCode so we can see the output on the console. Then run the following test:

- On the second device, tap a note as if you are going to edit it.
- On the first device, open the same note, edit the note, 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 we 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 we 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), we 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 our text view. This is the key point to implement your policy. Update the code so that the lastest version wins, by updating the `UIDocumentStateNormal` case to the following:

```
if (self.currentNote.documentState == UIDocumentStateNormal) {  
  
    self.navigationItem.leftBarButtonItem.enabled = YES;  
    self.noteContentView.text = self.currentNote.noteContent;  
  
}
```

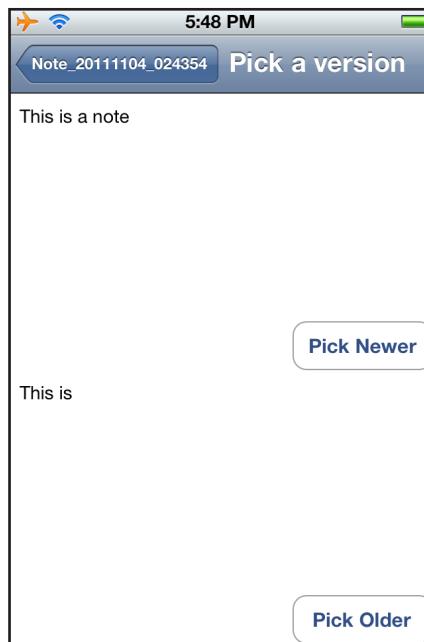
Compile and run to test it out - now when you edit a note on one device and have it open on a second, the second will automatically update to the latest content.

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 which 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 add the button when needed:

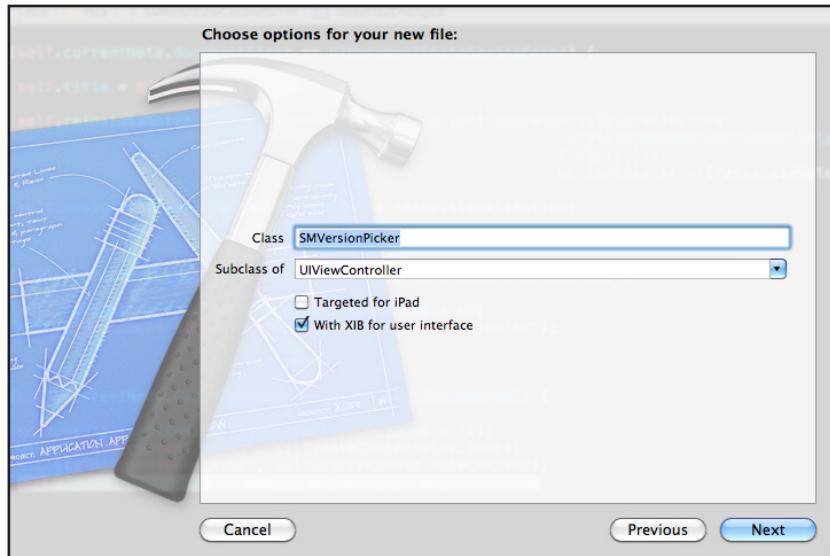
```
if (self.currentNote.documentState == UIDocumentStateNormal) {  
  
    self.navigationItem.leftBarButtonItem.enabled = YES;  
    NSLog(@"old content is %@", self.noteContentView.text);  
    NSLog(@"new content is %@", self.currentNote.noteContent);  
  
    if (![self.noteContentView.text  
        isEqualToString:self.currentNote.noteContent]) {  
  
        UIBarButtonItem * resolveButton = [[UIBarButtonItem alloc]  
            initWithTitle:@"Resolve"  
            style:UIBarButtonItemStylePlain  
            target:self  
            action:@selector(resolveNote)];  
  
        self.navigationItem.rightBarButtonItem = resolveButton;  
  
    }  
}
```

The action associated to the button will create a new view controller to let the user



to resolve the versions and push it onto the navigation stack. But before we can write this, we need to create the new view controller.

So create a new file with the **iOS\Cocoa Touch\UIViewController subclass** template, name it **SMVersionPicker**, make it a subclass of **UIViewController**, and check **With XIB for user interface**, as shown below:



To account for the navigation bar you might want it to be visible in design mode. Open the XIB, select the view and select "Navigation Bar" as "Top Bar."

Next open **SMVersionPicker.h** and replace the contents with the following:

```
#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;

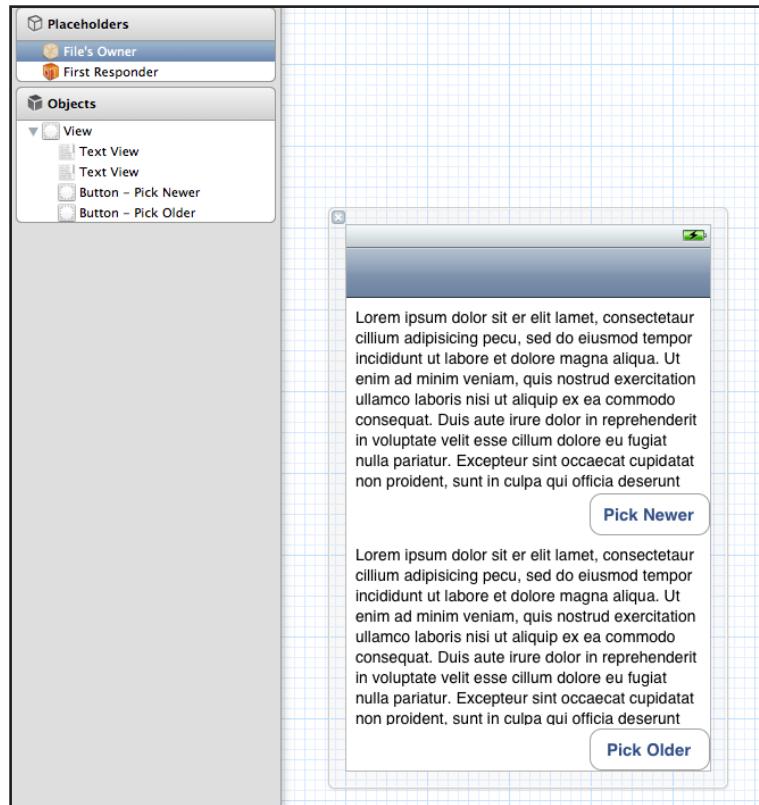
@end
```

The controller includes two text views to show both versions of a note, which are stored also in two string properties, plus a reference to the current note being



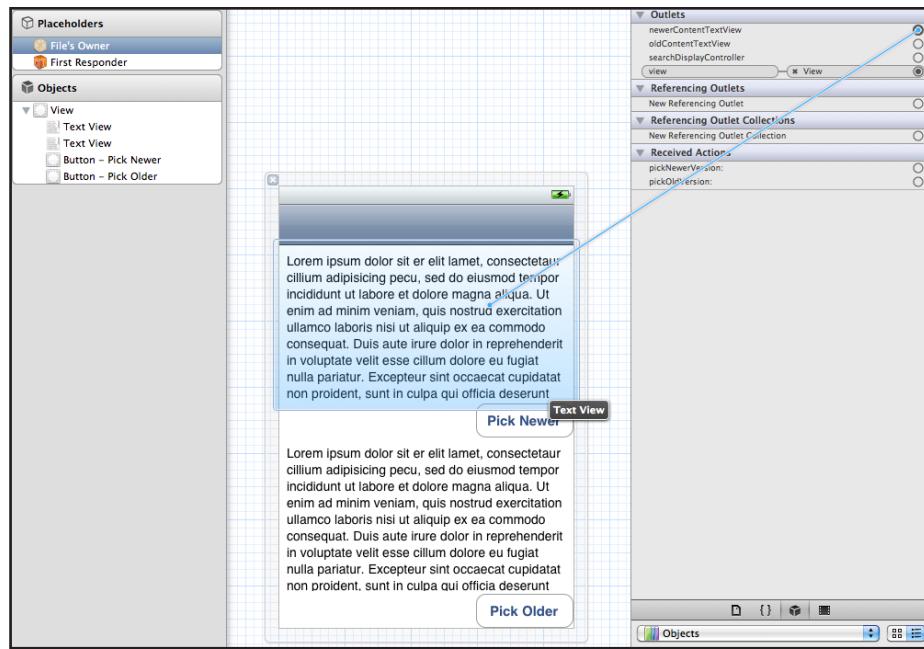
resolved. Two IBActions will be associated to the buttons that the user will tap to make his choice.

Rename **SMVersionPicker.xib** to **SMVersionPicker_iPhone.xib**, open it and place two UITextView and two UIButton so it looks like the following:

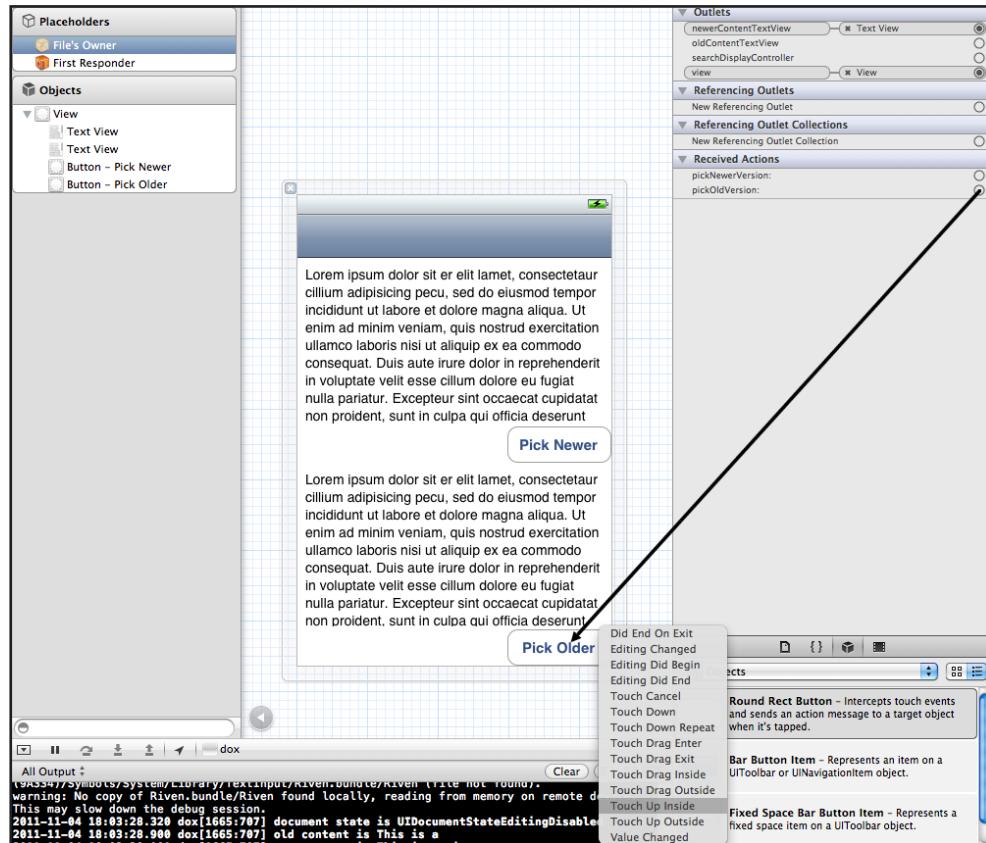


Also be sure to link both text views to their respective outlets:



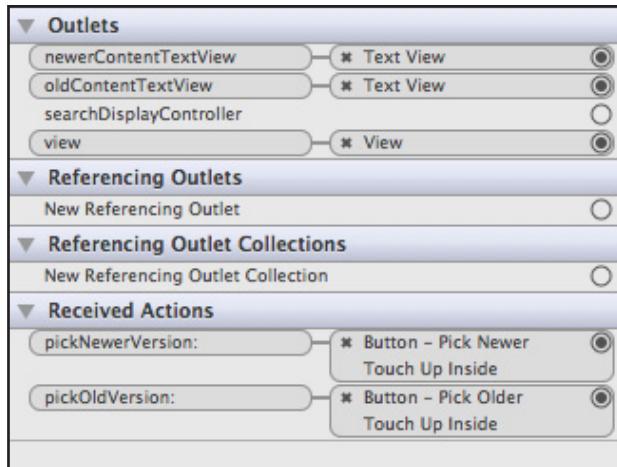


Then we also hook up the touch up inside event of each button with the IBActions declared in the header file.



At the end of the process the outlets should be set as follows:





Create a new xib called **SMVersionPicker_iPad** and repeat the same steps described above. Don't forget to set the File's Owner to SMVersionPicker, add a root view, and connect it to the File's Owner's view outlet.

Now that the user interface is complete, let's add the implementation. Open up **SMVersionPicker.m** and replace it with the following:

```
#import "SMVersionPicker.h"
#import "SMNote.h"

@implementation SMVersionPicker

@synthesize oldContentTextView, newerContentTextView;
@synthesize oldNoteContentVersion, newerNoteContentVersion;
@synthesize currentNote;

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.title = @"Pick a version";
}

@end
```

This synthesizes all the properties and assigns a default title when the view is loaded.

Next add the implementation of `viewWillAppear`:

```
- (void) viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    self.oldContentTextView.text = oldNoteContentVersion;
```



```
    self.newerContentTextView.text = newerNoteContentVersion;  
}
```

This makes it so when the view first appears, it populates the text views with the content stored in the properties.

Next add the callbacks for the button taps:

```
- (IBAction)pickNewerVersion:(id)sender {  
  
    self.currentNote.noteContent = self.newerContentTextView.text;  
    [self.currentNote updateChangeCount:UIDocumentChangeDone];  
    [self.navigationController popViewControllerAnimated:YES];  
  
}  
  
- (IBAction)pickOldVersion:(id)sender {  
  
    self.currentNote.noteContent = self.oldContentTextView.text;  
    [self.currentNote updateChangeCount:UIDocumentChangeDone];  
    [self.navigationController popViewControllerAnimated:YES];  
  
}
```

When a button is tapped the content of the note is updated and the `updateChangeCount:` message is sent. After that, we can close the view and pop back to the previous view on the stack.

Now that the new view controller is complete, all we have to do is add the code to display it in **SMViewController.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.currentNote.noteContent;
picker.oldNoteContentVersion = self.noteContentView.text;
picker.currentNote = self.currentNote;

[self.navigationController pushViewController:picker
    animated:YES];

}
```

Here we are, finally! Build and run, edit a document while you're viewing it on another, and you'll see our application is now able to detect changes to document and prompt the user with a view which helps choosing the version to keep on iCloud.

Handling Conflicts

When we described the list of available states we also mentioned `UIDocumentStateInConflict`. This state is assigned to documents which 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 our project so we can get an indication of when we have a conflict.

Inside `SMLViewController.m`, refactor `noteHasChanged` as follows:

```
- (void)noteHasChanged:(id)sender {
    [self.tableView reloadData];
}
```

Then add the following code to the bottom of `tableView:cellForRowAtIndexPath` (right before the "return cell;" line) to color a cell red when it is in conflict:

```
if ([n documentState] == UIDocumentStateInConflict) {
    cell.textLabel.textColor = [UIColor redColor];
} else {
```

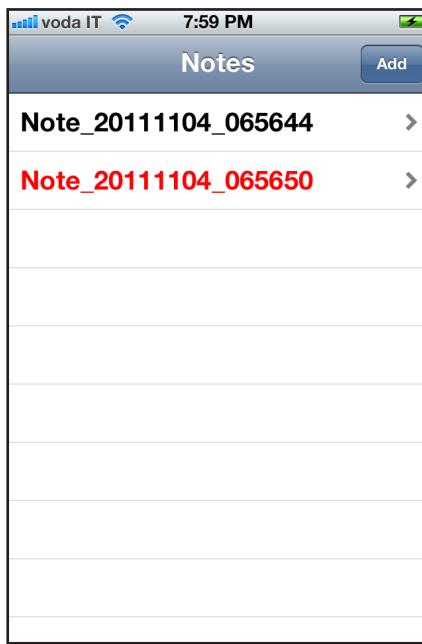


```
    cell.textLabel.textColor = [UIColor blackColor];
}
```

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 back 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 a 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:

```

if ([n documentState] == UIDocumentStateInConflict) {
    cell.textLabel.textColor = [UIColor redColor];

    NSArray *conflicts =
    [NSFileVersion unresolvedConflictVersionsOfItemAtURL:n.fileURL];

    for (NSFileVersion *version in conflicts) {

        NSLog(@"=====");
        NSLog(@"name = %@", version.localizedName);
        NSLog(@"date = %@", version.modificationDate);
        NSLog(@"device = %@", version.localizedDescriptionOfSavingComputer);
        NSLog(@"url = %@", version.URL);
        NSLog(@"=====");

    }

} else {
    cell.textLabel.textColor = [UIColor blackColor];
}

```

Run your project again, and you should see something like the following in the console:

```

2011-11-04 20:22:29.235 dox[332:707] =====
2011-11-04 20:22:29.784 dox[332:707] name = Note_20111104_065650
2011-11-04 20:22:31.089 dox[332:707] date = 2011-11-04 17:57:50 +0000
2011-11-04 20:22:33.295 dox[332:707] device = jeff
2011-11-04 20:22:35.647 dox[332:707] url = file:///localhost/private/var/mobile/Library/Application%20Support/
Ubiquity/genstore/peer-98B230AE-135F-D571-D757-E25E258309F8/98B230AE-135F-D571-D757-
E25E258309F8-0x00000000000014f/Note_20111104_065650.f09b74a9-75c7-49b0-8a13-86e9f007358a
2011-11-04 20:24:24.577 dox[332:707] =====

```

Notice that the URL of the document in conflict is a very long, unique and dynamically generated 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. To handle this case we have to refactor the `SMVersionPicker` a bit. In this case we assume there will be just two competing versions of a document but you can easily adapt it to allow more complex cases.

First of all we should re-factor the names of properties, for in this case it not appropriate to say 'old' and 'new' version of file. It is rather more correct to talk about 'this device' version and 'other device' version.



Right click on each property and method from **SMVersionPicker.h** in the list below and use the built-in refactor wizard (Refactor\Rename) to rename the properties/methods so he references are updated to use the new names as well:

- **oldContentTextView**: rename to `thisDeviceContentTextView`
- **newerContentTextView**: rename to `otherDeviceContentTextView`
- **oldNoteContentVersion**: rename to `thisDeviceContentVersion`
- **newerNoteContentVersion**: rename to `otherDeviceContentVersion`
- **pickNewerVersion**: rename to `pickOtherDeviceVersion`
- **pickOldVersion**: rename to `pickThisDeviceVersion`

Also add a new method declaration to the header, as shown below:

```
- (void) cleanConflicts;
```

Before we forget, open **SMVersionPicker_iPhone.xib** and **SMVersionPicker_iPad.xib**, make sure all the outlets are hooked up correctly (with the new names), and rename the buttons (the top should be "Pick Other" and the bottom should be "Pick This").

Next switch to **SMVersionPicker.m** and replace `viewWillAppear` and the button tap methods with the following:

```
- (void) viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    self.thisDeviceContentTextView.text = thisDeviceContentVersion;
    self.otherDeviceContentTextView.text = otherDeviceContentVersion;
}

- (IBAction)pickOtherDeviceVersion:(id)sender {
    self.currentNote.noteContent = self.otherDeviceContentTextView.text;
    [self cleanConflicts];
    [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.navigationController popViewControllerAnimated:YES];  
  
    [[NSNotificationCenter defaultCenter]  
    postNotificationName:@"com.studiomagnolia.conflictResolved"  
    object:self];  
  
}
```

In both button tap callbacks we store the value selected, call `cleanConflicts`, pop the view controller, and we post a notification.

Next add the implementation of `cleanConflicts` as follows:

```
- (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 we have met these conditions the conflict is fully resolved and a reload of the note list will show the normal black labeled titles!

One final step - we have to present the version picker if the document is in conflict. To do so, import the file at the top of **SMLListViewController.m**:



```
#import "SMVersionPicker.h"
```

Then modify `tableView:didSelectRowAtIndexPath:` as follows:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    SMNote * note = [notes objectAtIndex:indexPath.row];

    if (note.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 = note;

        NSArray *conflicts =
            [NSFileVersion unresolvedConflictVersionsOfItemAtURL:note.fileURL];

        for (NSFileVersion *version in conflicts) {

            SMNote *otherDeviceNote = [[SMNote alloc]
                                       initWithFileURL:version.URL];

            [otherDeviceNote openWithCompletionHandler:^(BOOL success) {

                if (success) {
                    picker.thisDeviceContentVersion = note.noteContent;
                    picker.otherDeviceContentVersion =
                        otherDeviceNote.noteContent;
                    [self.navigationController pushViewController:
                     picker animated:YES];
                }
            }];
        }
    }
    return;
}
```



```

if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    self.detailViewController = [[SMViewController alloc]
        initWithNibName:@"SMViewController_iPad" bundle:nil];
} else {
    self.detailViewController = [[SMViewController alloc]
        initWithNibName:@"SMViewController_iPhone" bundle:nil];
}
self.detailViewController.currentNote = note;
[self.navigationController pushViewController:
    self.detailViewController animated:YES];

}

```

The last step is to listen for the "com.studiomagnolia.conflictResolved" notification in SMLViewController to correctly update the list of notes and revert to black those previously red. Add the following to the viewDidLoad method.

```

[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(conflictResolved)
    name:@"com.studiomagnolia.conflictResolved"
    object:nil];

```

The associated selector 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:

- Create a conflicted file if you don't have one already (following the instructions earlier)
- On d1 tap the red note and pick one version
- Notice on d1 the reloaded list of notes with no conflict
- Wait to see the update also on d2

Our knowledge of iCloud is growing more and more. We are now able to create, edit, delete documents and even resolve version conflicts!

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 CoreData.



Using The Undo Manager

In all the examples so far, to save a document we 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, we'll cover is a third way to trigger this mechanism: using the undo manager.

`NSUndoManager` is a class available since iOS 3 and it makes it easy to keep track of (and undo) 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:

- **target:** The object to which the selector belongs.
- **selector:** The method called to revert the modification.
- **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 that so, if you want, you can reinstate it by means of a redo. More on this in a few minutes.

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.



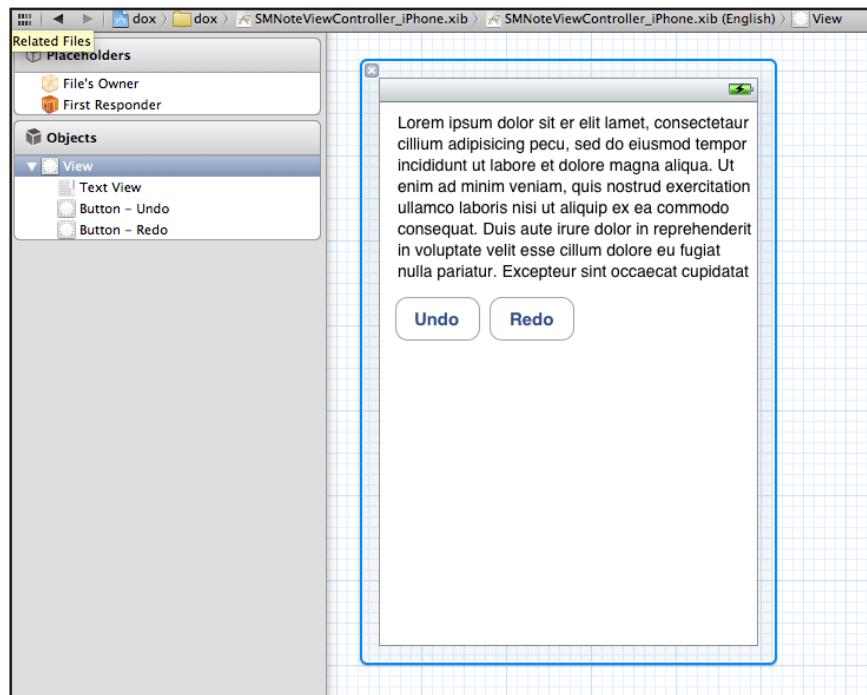
Let's see how we can integrate an undo manager in our application. For this example, we're going to go back to our **dox4-Starter** project to start at a clean slate. You'll find this in the resources for this chapter - just make a copy and we'll begin from there. Don't forget to update your bundle ID, entitlements, and code signing profile.

Open up **SMViewController.h** and add two new properties and methods, as shown below:

```
@property (strong, nonatomic) IBOutlet UIButton *undoButton;
@property (strong, nonatomic) IBOutlet UIButton *redoButton;

- (IBAction)performUndo:(id)sender;
- (IBAction)performRedo:(id)sender;
```

Then open **SMViewController_iPhone.xib**, squeeze a bit the text view and add two UIButtons for undo and redo:



Note you should put the buttons high enough so that they aren't covered up by the keyboard when it appears.

Also connect the buttons to their outlets, and connect the Touch Up Inside action for each buttons to their respective methods. Finally, repeat this entire process for **SMViewController_iPad.xib**.

Now that our user interface is hooked up with the view controller we need to refactor its logic.



First synthesize the new properties at the top of **SMViewController.m**:

```
@synthesize undoButton;
@synthesize redoButton;
```

As we said above there is no need to call `updateChangeCount:` or manually keep track of whether the note has changed, modify `textViewDidChange` and `viewWillDisappear` as follows:

```
- (void)textViewDidChange:(UITextView *)textView {
    // Do nothing
}

- (void)viewWillDisappear:(BOOL)animated
{
    [self saveNoteWithContent:self.noteContentView.text];
    [super viewWillDisappear:animated];
}
```

Then add the implementation for `saveNoteWithContent:` above `viewWillDisappear`:

```
- (void) saveNoteWithContent:(NSString *)newContent {

    NSString *currentText = self.currentNote.noteContent;

    if (newContent != currentText) {

        [self.currentNote.undoManager registerUndoWithTarget:self
            selector:@selector(saveNoteWithContent:)
            object:currentText];

        self.currentNote.noteContent = newContent;
        self.noteContentView.text = self.currentNote.noteContent;

    }

    self.undoButton.enabled = [self.currentNote.undoManager canUndo];
    self.redoButton.enabled = [self.currentNote.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 Note class (inherited from `UIDocument`). After that it updates the states of buttons according to the stack of changes.

This way, whenever an 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. The actions associated to the buttons are very simple: we call undo or redo if the manager is entitled to perform that action:



```
- (IBAction)performUndo:(id)sender {  
    if ([self.currentNote.undoManager canUndo]) {  
        [self.currentNote.undoManager undo];  
    }  
  
}  
  
- (IBAction)performRedo:(id)sender {  
    if ([self.currentNote.undoManager canredo]) {  
        [self.currentNote.undoManager redo];  
    }  
  
}
```

The final touch is to update the state of buttons before the view appears. Add these two lines to the bottom of `viewWillAppear:`:

```
self.undoButton.enabled = [self.currentNote.undoManager canUndo];  
self.redoButton.enabled = [self.currentNote.undoManager canredo];
```

Here is a new tool that we can exploit to build user friendly applications, which enable the user to review changes to the content of notes.

Compile and run on two different devices, and run through the following test case:

1. Open a note on device 1 and edit the text. Go back to the list of notes.
2. Wait 15 seconds, and open the same note on device 2. Verify you see the edited text. This proves that the changes were propagated by the undo manager itself, without explicitly saving or calling `updateChangeCount`.
3. Open the note on device 1, and tap undo. This removes the change you moved last time. You can also redo the change!

Note that the undo/redo stack is according to a single document instance. Once you reopen the same note undo and redo buttons are enabled according to the stack populated during previous interactions.

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`:

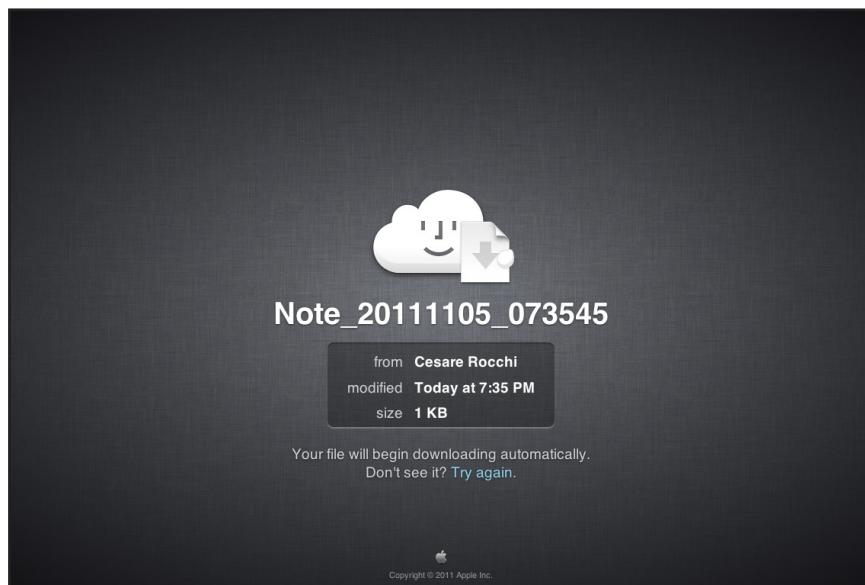


```
- (NSURL *)URLForPublishingUbiquitousItemAtURL:(NSURL *)url  
expirationDate:(NSDate **)outDate error:(NSError **)error;
```

This method returns a url like the following:

```
https://www.icloud.com/documents/dl/  
?p=2&t=BAJJZKAKzzXMg7o4tC8B0DEI0zk0d6UN2q0A
```

which is temporary 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 we can integrate this functionality in our application. We will start again from the **dox4-Starter** project, so make a copy and update your bundle ID, entitlements, and code signing profile as usual.

In this section we will add an export button when a note is opened. Tapping the button will send an email containing the URL of the exported note.

Open **SMViewController.h** and make the following changes:

```
// Add to top of file  
#import <MessageUI/MessageUI.h>  
  
// Modify interface to implement MFMailComposeViewControllerDelegate  
@interface SMViewController : UIViewController <UITextViewDelegate,  
MFMailComposeViewControllerDelegate>  
  
// Predeclare new method  
- (NSURL *) generateExportURL;
```



Since we're using the MessageUI framework we have to import it into our 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 **SMViewController.m** and add the following code to the bottom of viewDidLoad to add the export button to the navigation bar:

```
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 It will expire in one hour.", url]
        isHTML:NO];
    [self presentModalViewController:mailComposer animated:YES];
}
```

This creates a view controller similar to the one used by the mail application in iOS. We set the subject and the body of the email. The url of the note is generated by the following function:

```
- (NSURL *) generateExportURL {

    NSTimeInterval oneHourInterval = 3600.0;
    NSDate *expirationInOneHourSinceNow =
        [NSDate dateWithTimeInterval:oneHourInterval
            sinceDate:[NSDate date]];
    NSError *err;

    NSURL *url = [[NSFileManager defaultManager]
        URLFor PublishingUbiquitousItemAtURL:[self.currentNote fileURL]
        expirationDate:&expirationInOneHourSinceNow]
```



```
        error:&err];
    if (err)
        return nil;
    else
        return url;

}
```

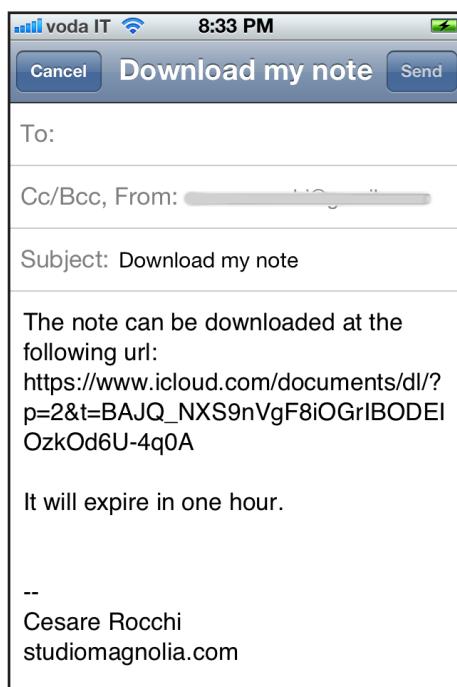
Here we create an interval of one hour to calculate the expiration date of the download link. Then we pass it as a parameter to the URLForPublishingUbiquitousItemAtURL:expirationDate:error method, which instructs iCloud to make the file downloadable for a given time. If you are not interested in an expiration 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 {

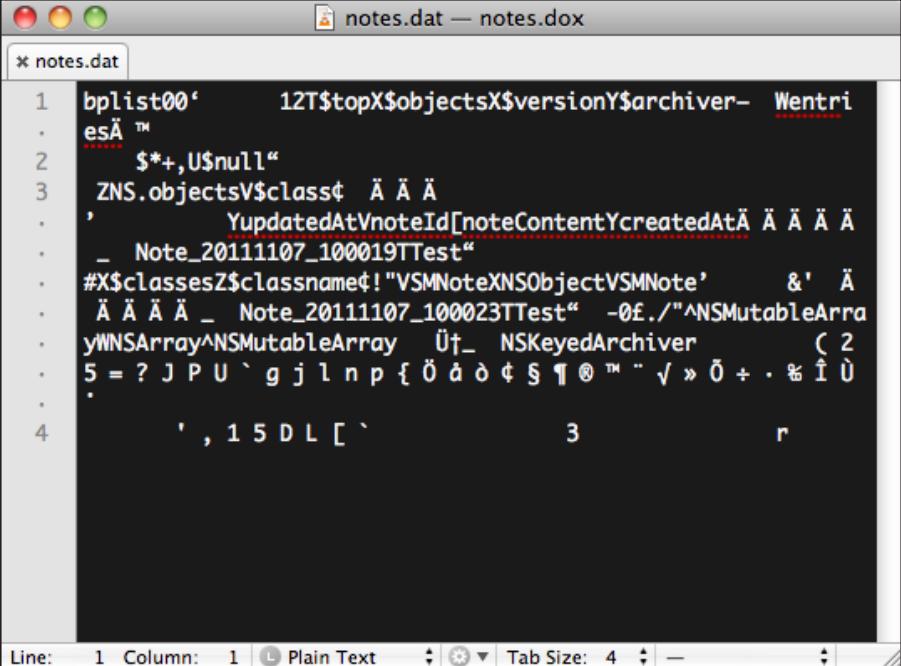
    if (result == MFMailComposeResultSent) {
        [self dismissModalViewControllerAnimated:YES];
    }
}
```

Compile 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?

You can also export data stored with `NSFileWrapper`. In this case you would place the export button and functionalities in the `SMNoteListViewController`. The code will be exactly the same. When you unzip the downloaded file it will produce a folder as in the following screenshot:



```
notes.dat
1 bplist00'      12T$topX$objectsX$versionY$archiver- Wentri
. esÃ™
2 $*+,U$null"
3 ZNS.objectsV$class# Ä Ä Ä
. ,      YupdatedAtVnoteId[noteContentYcreatedAtÄ Ä Ä Ä Ä
. _ Note_20111107_100019TTest"
. #X$classesZ$classname¢! "VSMNoteXNSObjectVSMNote'      &' Ä
. Ä Ä Ä Ä _ Note_20111107_100023TTest" -0f./"^NSMutableArray
. yWNSArray^NSMutableArray Ü†_ NSKeyedArchiver    ( 2
. 5 = ? J P U ` g j l n p { Ö å ð ö § ¶ ® ™ " √ » Ö + . % î Ù
. .
4     ' , 1 5 D L [ `           3           r

Line: 1 Column: 1 Plain Text Tab Size: 4
```

That was the exact same structure that we created in the `contentsForType:error:` method. Data is encoded in binary format, in this case representing the array of notes saved on iCloud.

Congrats, we have now covered almost everything you might want to do with loading and saving data with the `UIDocument` class. Next up we'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 which 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 iOS3 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 our objects we will write actual Objective-c code. For example to create a new employee and assign him to a department we 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 our domain. If Core Data were an ORM it would just store information in SQLite form. It allows instead to dump 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 we will see in details their respective roles.



Model

We 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 our 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 our 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)

Besides relationships there is a similar concept, fetched property. 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 among which:

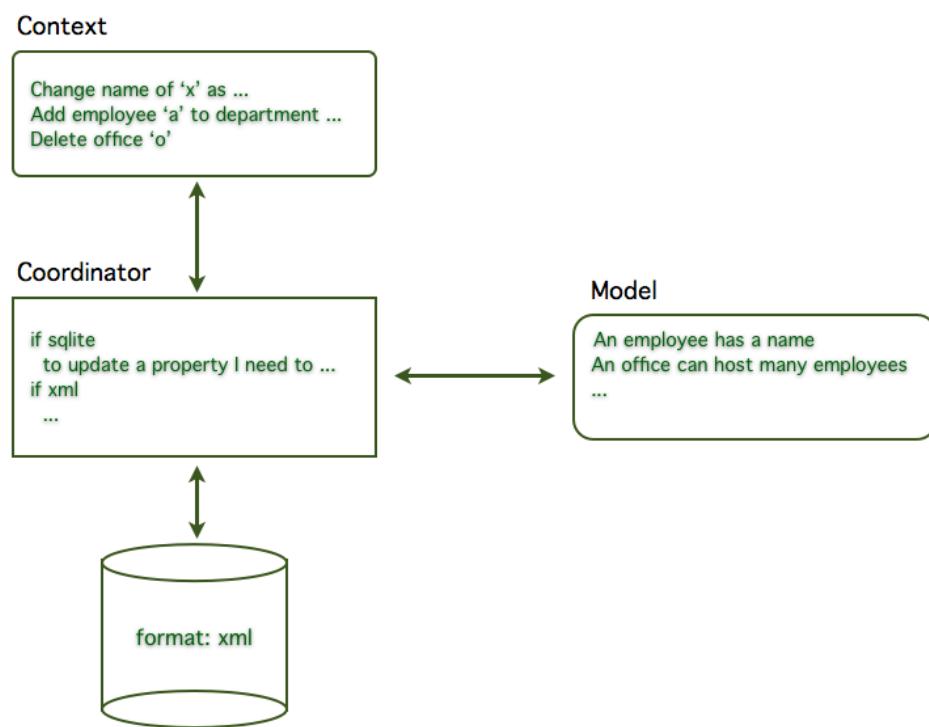
- validating values (for example a filler for the name property has to be a string)
- managing undo and redo



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 we 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. On a side note we 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 we 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 we are going to extend a bit our domain. Each note will have one or more tags attached. This way we will have two entities in our domain, note and tag, related by a many-to-many relationship, meaning that each note can have multiple tags and a 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. If so, you can use **dox8-CoreData** as a starting point.

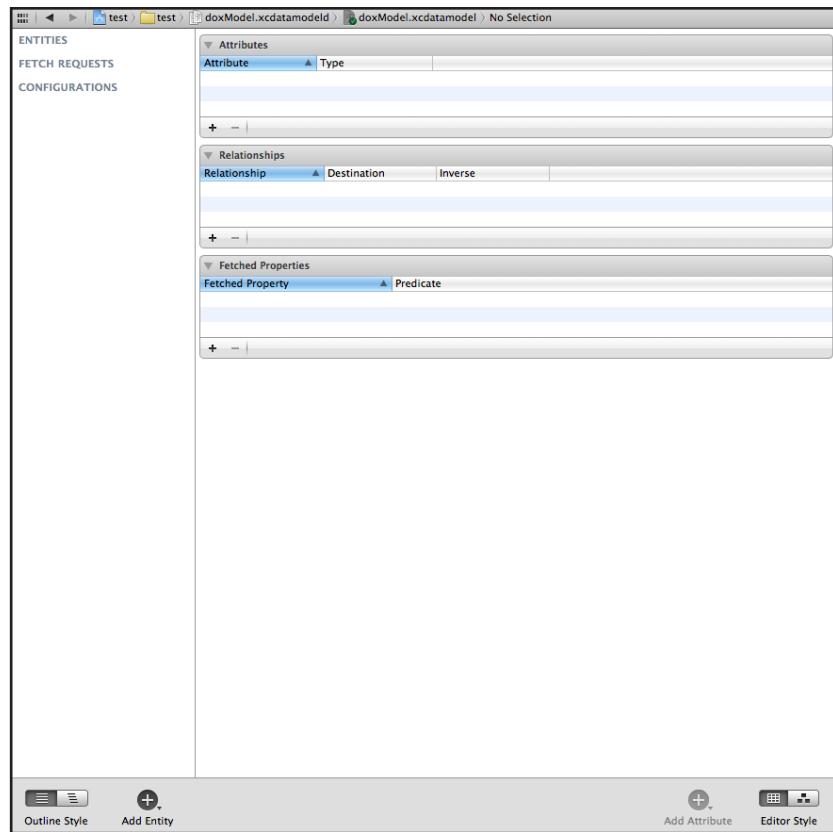
Creating A Model For Notes And Tags

One last time, we will start from the **dox4-Starter** project, so create a copy and edit your bundle ID, entitlements, and code signing profile as usual.

First of all we can delete our SMNote class, because we'll be generating a new version with the Core Data modeling tool in Xcode soon.

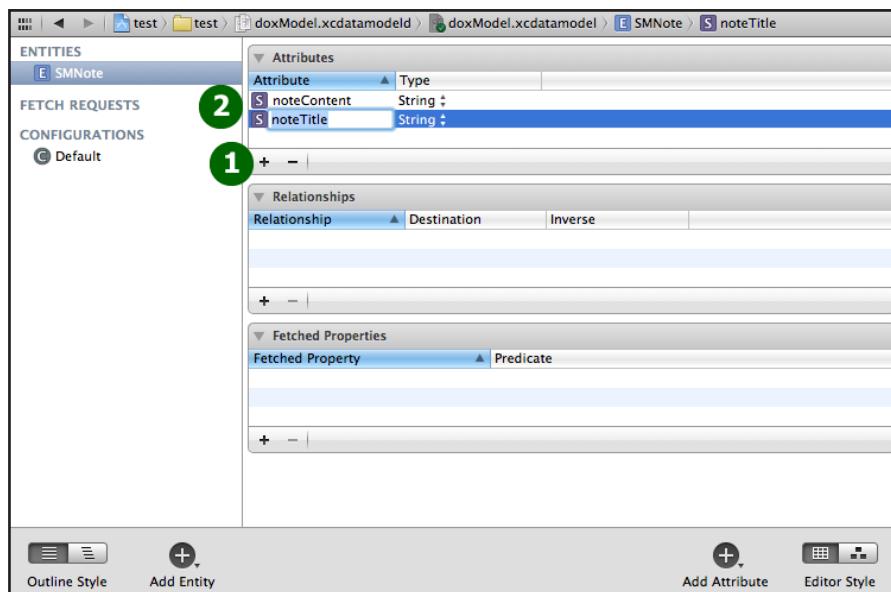
Next, create a new file with the **iOS\Core Data\Data Model** template, and name it **doxModel.xcdatamodeld**. Select the file, and you'll see the following view:





This is where we can define the entities we'll use in our project and their relationships. At the bottom there is a button **Add Entity**. Click it and call the new entity **SMNote**.

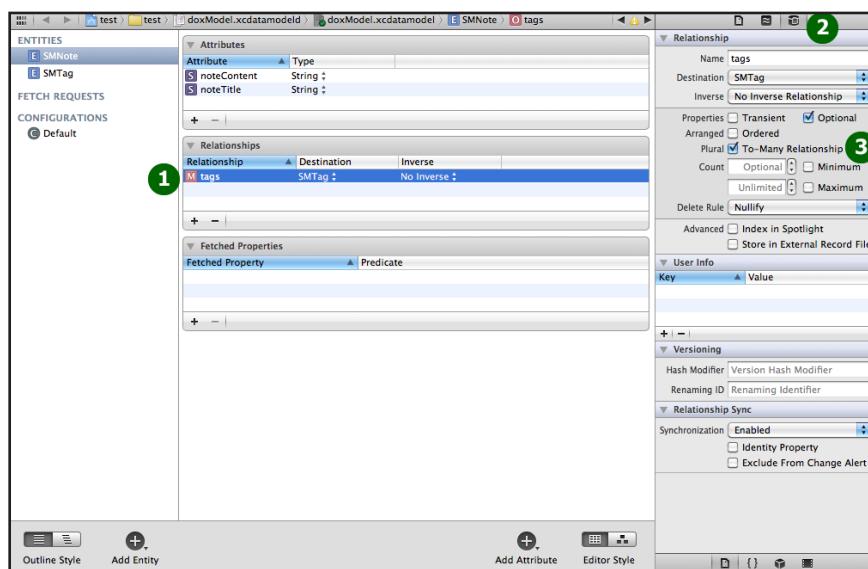
Next add two properties to the class. Click on the '+' sign in the attributes section and add two entries named noteContent and noteTitle, and set them as strings.



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 we have created. As we said 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 we should specify that the relation is of type **To-Many**. With the relation selected we open the panel on the right and we select the corresponding option.

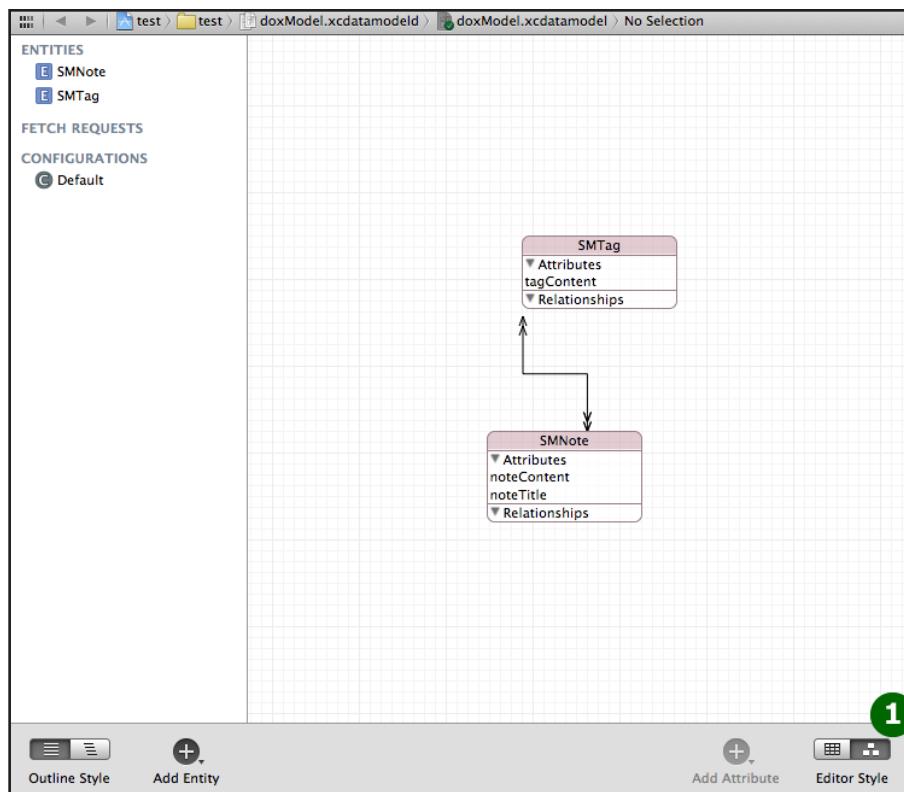


This is just one part of relation, from a note to a tag. We should also model its counterpart.

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 we switch to the **Editor Style** to **Graph** (click the button in the bottom right of the Core Data editor), we'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 we can generate classes for the entities we 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's editor menu select "**Create NSManagedObject Subclass**", and click Create. XCode will generate header and implementation files for both of our entities - `SMTag.h/m` and `SMNote.h/m`.

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 we 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 our 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 we mentioned above there are three elements which interplay in the Core Data-based application: model, context and coordinator. Before adding these elements to the project we have to import the Core Data framework. 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 we 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 we can perform the actions needed to refresh data. The three classes that we need are: NSManagedObjectModel, NSPersistentStoreCoordinator and NSManagedObjectContext.

Go go ahead and update **SMAppDelegate.h** as follows:

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

@interface SMAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) UINavigationController
*navigationController;

@property (readonly, strong, nonatomic) NSManagedObjectContext
*managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel
*managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;

@end
```



We have also added two helper methods to retrieve the documents directory and to save changes.

Next switch to **SAppDelegate.m** and synthesize the new properties:

```
@synthesize managedObjectContext = __managedObjectContext;
@synthesize managedObjectModel = __managedObjectModel;
@synthesize persistentStoreCoordinator = __
    persistentStoreCoordinator;
```

Let's start by defining the model. The task of this method is to load the schema that we have defined graphically to provide all the rules and constraints needed to verify the consistency. The model needs a url, that we build from the main bundle file of the application.

```
- (NSManagedObjectModel *)managedObjectModel
{
    if (__managedObjectModel != nil)
    {
        return __managedObjectModel;
    }

    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"doxModel"
                                                 withExtension:@"momd"];

    __managedObjectModel = [[NSManagedObjectModel alloc]
                           initWithContentsOfURL:modelURL];
    return __managedObjectModel;
}
```

You might wonder why we 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 we have to load from the bundle of the application.

Now we 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 our case we are going to use a SQLite store type so we need to provide a name for the file where the database will be stored, in our case 'dox.sqlite'. The coordinator is initialized with the model created above. Once we have an instance of the coordinator we can add the persistent store by specifying the type and the store url. We could also provide a configuration and some options. To keep things simple, for the moment we will pass nil as a parameter. We will come back to options when we will configure Core Data for iCloud.

The method `persistentStoreCoordinator` calls the helper method `applicationDocumentsDirectory`, which is defined as follows:

```

- (NSURL *)applicationDocumentsDirectory
{
    return [[[NSFileManager defaultManager] URLsForDirectory:
        NSDocumentDirectory inDomains:NSUserDomainMask] lastObject];
}

```

Now we are left with 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.persistentStoreCoordinator =
    }
}

```



```

        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 we have defined previously. Let's see how we can retrieve and store notes in our new data structure.

Adding And Retrieving Notes

As mentioned above, the interaction with the data store is managed by the context. So we 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 **SMLViewController.h** with the following:

```
#import <UIKit/UIKit.h>
#import "SMNote.h"
#import "SMTag.h"
#import "SMViewController.h"

@interface SMLViewController : UITableViewController
<NSFetchedResultsControllerDelegate>

@property (strong, nonatomic) SMViewController *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.

Next switch to **SMLViewController.m** and replace the contents with the following (yes we're deleting everything to start from scratch):

```
#import "SMLViewController.h"
#import "SMViewController.h"

@implementation SMLViewController

@synthesize detailViewController = _detailViewController;
@synthesize managedObjectContext = __managedObjectContext;
@synthesize fetchedResultsController = __fetchedResultsController;

@end
```

There are a few steps to use a fetched results controller:

1. Create a fetch request.
2. Create an entity description for the entity to be fetched (in our 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 for the above steps - go ahead and add it to the implementation:

```
- (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 we did not provide any section path (see the Core Data documentation for more details) and we specified a name for the cache to quicken the retrieval time. Finally we have called `performFetch:` to load data when the instance variable is created. You might have noticed that we have specified 'self' as the delegate of the controller.

Next we are going to override the fetched result controller delegate's `controllerDidChangeContent:` method, which is called when the result controller has processed some change to data. Override it to reload data in the table view:

```

- (void)controllerDidChangeContent:
{
    (NSFetchedResultsController *)controller {
        NSLog(@"something has changed");
        [self.tableView reloadData];
    }
}

```

Then implement `viewDidLoad` to add a button to add a new note as usual:

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    UIBarButtonItem *addNoteItem = [[UIBarButtonItem alloc]
        initWithTitle:@"Add"
        style:UIBarButtonItemStylePlain
        target:self
        action:@selector(addNote:)];
    self.navigationItem.rightBarButtonItem = addNoteItem;
}

```

Unlike previous examples, notes populating the table view are not stored in an array inside the class, but rather the Core Data store itself. So when we add a new note we instantiate it in the context, we set its properties and we save the context:

```

- (void)addNote:(id)sender {
    SMNote *newNote = [NSEntityDescription
        insertNewObjectForEntityForName:@"SMNote"
        inManagedObjectContext:self.managedObjectContext];

    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"yyyyMMdd_hhmss"];
}

```



```
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]);  
    abort();  
}  
}
```

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 we have an instance we 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, we 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 <NSFetchedResultsControllerSectionInfo> sectionInfo =
        [[self.fetchedResultsController sections] objectAtIndex:section];
    return [sectionInfo numberOfRowsInSection];
}

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

    NSManagedObject *managedObject = [self.fetchedResultsController
        objectAtIndex:indexPath];

    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 = [managedObject valueForKey:@"noteTitle"];
return cell;
}
```

This way data persisted on disk will be displayed in the table view as in previous examples. One final step: Open **SMAppDelegate.m** and add this line to application:didFinishLaunchingWithOptions, right before creating the navigation controller:

```
masterViewController.managedObjectContext =
    self.managedObjectContext;
```

At this point we can compile to check if all the methods in **SMViewController.m** work correctly.

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 we quit and restart the application, the data is properly reloaded from the Core Data SQLite database. Nice!

If we want to show a single note we have to push an instance of SMViewController. So open **SMListViewController.m** and add the implementation for tableView:didSelectRowAtIndexPath: method as follows:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
```



```
(NSIndexPath *)indexPath  
{  
    SMNote *n = (SMNote *)[self.fetchedResultsController  
        objectAtIndex:indexPath];  
  
    if ([[UIDevice currentDevice] userInterfaceIdiom] ==  
        UIUserInterfaceIdiomPhone) {  
        self.detailViewController = [[SMViewController alloc]  
            initWithNibName:@"SMViewController_iPhone" bundle:nil];  
    } else {  
        self.detailViewController = [[SMViewController alloc]  
            initWithNibName:@"SMViewController_iPad" bundle:nil];  
    }  
  
    self.detailViewController.currentNote = n;  
    [self.navigationController pushViewController:self.detailViewController  
        animated:YES];  
}
```

Now let's move on to the visualization of a single note and its tags.

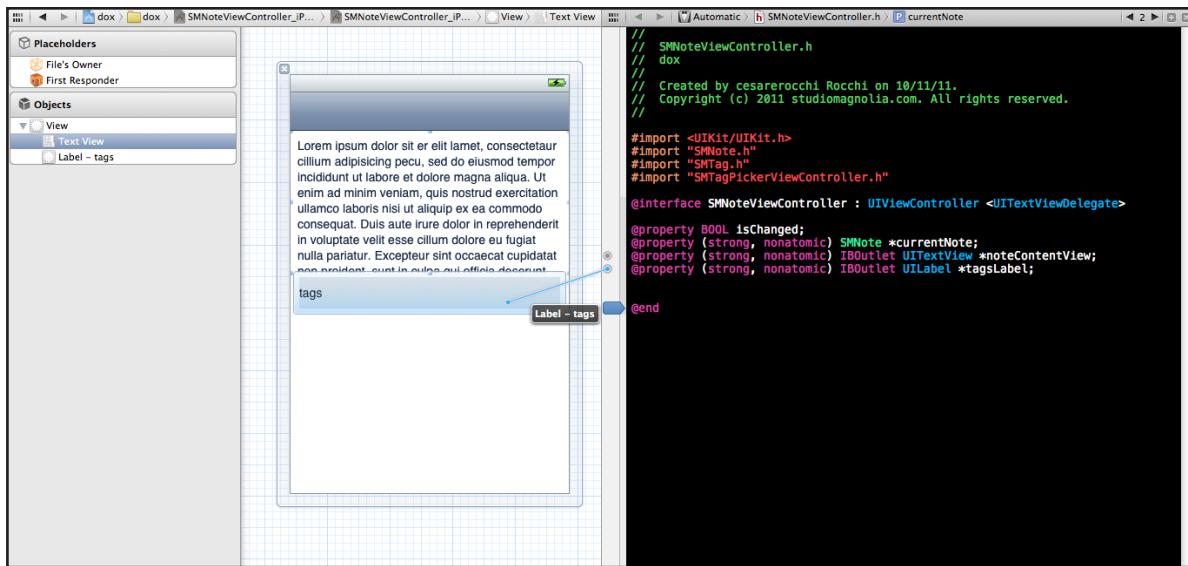
Single Note and Tags

Now it is time to rework the user interface of SMViewController. First add a new outlet for the label which will show tags in **SMViewController.h**:

```
@property (strong, nonatomic) IBOutlet UILabel *tagsLabel;
```

We have to squeeze the text view to accommodate a new label to show tags related to a note. Open **SMViewController_iPhone.xib**, resize the text view, and add a new label. Then connect the label to a new outlet declared in the header file.





Repeat the same steps for **SMViewController_iPad.xib**.

Then open **SMViewController.m** and replace it with the following:

```

#import "SMViewController.h"
#import "SMTag.h"
#import "SMTagPickerController.h"

@implementation SMViewController

@synthesize currentNote, noteContentView, isChanged, tagsLabel;

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    if (self.currentNote) {
        self.noteContentView.text = self.currentNote.noteContent;
        self.title = self.currentNote.noteTitle;

        NSArray *tagsArray = [self.currentNote.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.noteContentView becomeFirstResponder];
    }
}

@end

```



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 we have transformed the set in an array and concatenated its elements with commas.

A note gets saved when the SMViewController disappears. Unlike previous example, here we call the save: method of the context to store changes. The implementation of viewWillDisappear is the following:

```
- (void) viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];

    self.currentNote.noteContent = self.noteContentView.text;

    NSError *error = nil;
    if (![self.currentNote.managedObjectContext save:&error]) {
        NSLog(@"Core data error %@", error, [error userInfo]);
        abort();
    }
}
```

A new note comes with no tags, so we have to create a way to assign one or more tags to a note. In the next step we will build a tag picker controller. Since we are editing the single note controller let's add the code to show the picker. The action is triggered when the user taps the tags label. We associate this action once the view is loaded.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.noteContentView.delegate = self;
    isChanged = NO;

    tagsLabel.userInteractionEnabled = YES;
    UITapGestureRecognizer *tapGesture = [[UITapGestureRecognizer alloc]
        initWithTarget:self
        action:@selector(tagsTapped)];
    [tagsLabel addGestureRecognizer:tapGesture];
}
```

The action tagsTapped creates an instance of a new controller (which we are going



to create in a bit), assign it the current note and push it onto the navigation stack. As usual we 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.currentNote;
    [self.navigationController pushViewController:tagPicker
                                         animated:YES];
}
```

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\UIViewController subclass** 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 usual.

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 our domain.

Switch to **SMTagPickerController.m** and synthesize these new properties:

```
@synthesize currentNote;
@synthesize pickedTags;
@synthesize fetchedResultsController = __fetchedResultsController;
```

The new fetched results controller 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;
    }
    NSFetchRequest *fetchRequest = [[NSFetchRequest 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 we 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 we use `self.currentNote.managedObjectContext`.

To keep things simple, we want to avoid creating another view controller to enter new tags. So we will populate our application with a static list of tags if none are present. We will perform this task when the tag picker has loaded.

First we perform a query and if that is empty we create a few tags and save them in the context. Once saved we reload tags from the persistent store.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    pickedTags = [[NSMutableSet alloc] init];
    self.title = @"Tag your note";

    // Retrieve tags
    NSError *error;
    if (![_fetchedResultsController performFetch:&error]) {
        NSLog(@"Unresolved error %@", error, [error userInfo]);
        abort();
    }

    // in case there are no tags,
    // presumably it's the first time we run the application
    if (_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 (![_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 currentNote.tags) {
    [pickedTags addObject:tag];
}
}
}

```

This code is executed when the tag picker is loaded but new tags are added just once, the first time the application is run. We 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 = pickedTags;

    NSError *error = nil;
    if (![[self.currentNote.managedObjectContext save:&error]]) {
        NSLog(@"Core data error %@", error, [error userInfo]);
        abort();
    }
}
}

```

Now we have all the infrastructure needed to create the tag picker. We 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 = @"TagCell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    cell.accessoryType = UITableViewCellAccessoryNone;

    SMTag *tag = (SMTag *)[self.fetchedResultsController
        objectAtIndexPath:indexPath];
    if ([pickedTags containsObject:tag]) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }

    cell.textLabel.text = tag.tagContent;
    return cell;
}

```

Finally when we tap a cell we will add or remove the tag from the array of picked tags.

```

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    SMTag *tag = (SMTag *)[self.fetchedResultsController
        objectAtIndexPath:indexPath];
    UITableViewCell * cell = [self.tableView
        cellForRowAtIndexPath:indexPath];
    [cell setSelected:NO animated:YES];

    if ([pickedTags containsObject:tag]) {
        [pickedTags removeObject:tag];
        cell.accessoryType = UITableViewCellAccessoryNone;
    } else {
        [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.





We have built a good old Core Data application! 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 iOS5 Core Data has been extended to support iCloud. One key addition is related to the event that happens when you push new changes to the store. Each variation 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 we 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 **dox8-CoreData** 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, we are in an asynchronous situation, since data from the cloud might take a while to be downloaded. That's why we have to use a secondary thread to instantiate the store coordinator.

Let's proceed step by step. In the previous example, in the SMAppDelegate, we have provided no options to the addPersistentStoreWithType:configuration:URL:options:error: method. In this case we have to create an array of options, in which we specify the features to be enabled in the persistent store. Three are the options we are interested in:

- **NSPersistentStoreUbiquitousContentNameKey**: A unique name which 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 we can provide the name we like, e.g. @"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, we'll give you 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. We 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 we 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 **SMAAppDelegate.m** with the following:

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (_persistentStoreCoordinator != nil) {
        return _persistentStoreCoordinator;
    }

    __persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel: [self managedObjectModel]];

    NSPersistentStoreCoordinator* psc = __persistentStoreCoordinator;
    NSString *storePath = [[self applicationDocumentsDirectory]
        stringByAppendingPathComponent:@"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), ^{
    NSURL *storeUrl = [NSURL fileURLWithPath:storePath];

    // 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]);
    abort();
}

[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;
}

```

We have slightly revised the applicationDocumentsDirectory to return a string. Here is the implementation. Remember to update also the declaration in the header file.

```

- (NSString *)applicationDocumentsDirectory {
    return [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask, YES) lastObject];
}

```

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 we send messages to a context which implements a queue like this we 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 we can use:

```

[moc performBlockAndWait:^{
    [moc setPersistentStoreCoordinator: coordinator];
    // other configurations
}];

```

As we mentioned 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 we define the context. In this case the notification has a very long name: NSPersistentStoreDidImportUbiquitousContentChangesNotification. We can set up an observer like this:

```

[[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(mergeChangesFrom_iCloud:)
    name:NPSPersistentStoreDidImportUbiquitousContentChangesNotification
    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 we use the `performBlock:` API.

```
- (void)mergeChangesFrom_iCloud:(NSNotification *)notification {
    NSManagedObjectContext* moc = [self managedObjectContext];
    [moc performBlock:^{
        [self mergeiCloudChanges:notification
                           forContext:moc];
    }];
}
```

We should remember to add this method signature to the header file.

```
- (void)mergeiCloudChanges:(NSNotification*)note
                     forContext:(NSManagedObjectContext*)moc;
```

The actual method which performs the merging is defined as follows.

```
- (void)mergeiCloudChanges:(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 which have been notified as changed, inserted or deleted. If you need to have a higher control on the merging, we will discuss that below.

Once this code is executed all the views which 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 we have setup a notification when we add the persistent store to the coordinator. It is important to catch it and refresh the user interface.

In **SMLViewController.m**, once the view has loaded, we listen for such a notification. Add this in `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]);
        abort();
    } 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, we 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 app. You should follow the usual protocol: install on two devices and see changes propagated from one to another.

Merging Conflicts in Core Data Apps

In the last application we have seen that merging can be done by means of a simple call to the `mergeChangesFromContextDidSaveNotification:` method. For sake of completeness we should mention that, if you like to dig deeper, there are many more aspects that you might want to consider.

In this section we won't guide you through the process of building an application, for conflict resolution is very specific to the purpose of the application. We 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. We 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 which 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 our 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 we add a note, if we print out the notification object we should see in the console a message like the following.

```
- (void)mergeiCloudChanges:(NSNotification*)notification
    forContext:(NSManagedObjectContext*)moc {
    NSLog(@"%@", notification);
}

// You'd see this in the console:

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 we pass to the mergeChangesFromContextDidSaveNotification: method. So a schema of a custom policy for conflict resolution might be like this.

```
- (void)mergeiCloudChanges:(NSNotification*)note
```



```
forContext:(NSManagedObjectContext*)moc {  
  
    NSMutableDictionary *mergingPolicyResult = [NSMutableDictionary  
dictionary];  
  
    // 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 which wraps 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 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 which 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;
}
```

For more information, see: https://developer.apple.com/library/ios/#qa/qa1719_index.html

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 your are building an 'ecosystem', e.g. an iOS and MacOS application, it is much 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! :]



Beginning OpenGL ES 2.0 with GLKit

by Ray Wenderlich

iOS5 comes with a new set of APIs that makes developing with OpenGL much easier than it used to be.

The new set of APIs is collectively known as GLKit. It 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. **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.
3. **GLMath**. Prior to iOS5, pretty much every game needed their own math library with common vector and matrix manipulation routines. Now with GLMath, most of the common math routines are there for you!
4. **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!

The goal of this tutorial is to get you quickly up-to-speed with the basics of using OpenGL with GLKit, from the ground up, assuming you have no experience whatsoever. We 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 in the past, or if you're a complete beginner!



What is OpenGL?

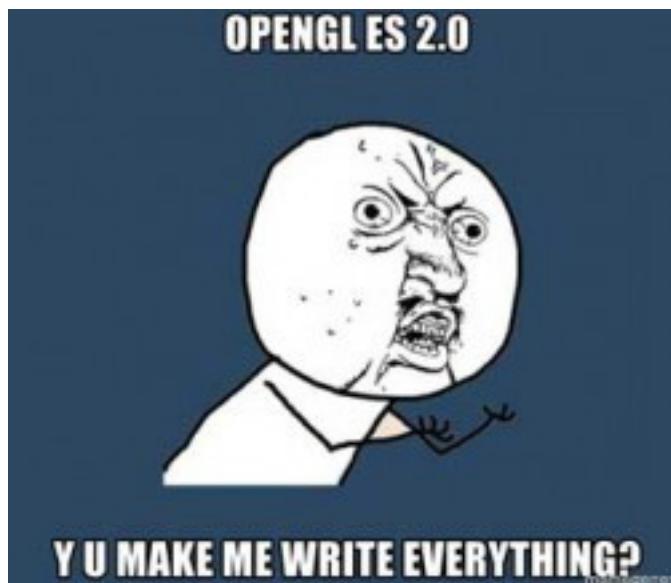
Before we get started, let's first talk about what OpenGL is for those of you who are unfamiliar with it.

OpenGL 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 special effects.

If you are familiar with the Cocos2D game framework, this is built on top of OpenGL. Programmers sometimes prefer working with Cocos2D instead of OpenGL directly, because it is easier to learn and often faster to write code with. However, OpenGL gives you the most power, so it's good to know. Plus, GLKit has made working with OpenGL directly much easier than it used to be.

OpenGL is a C-based API. If you do not know C (and only know Objective-C), some of the code in this chapter might look a bit unfamiliar. If this is the case, I recommend getting a handy C reference book to help you get used to the syntax.

OpenGL ES 1.0 vs OpenGL ES 2.0

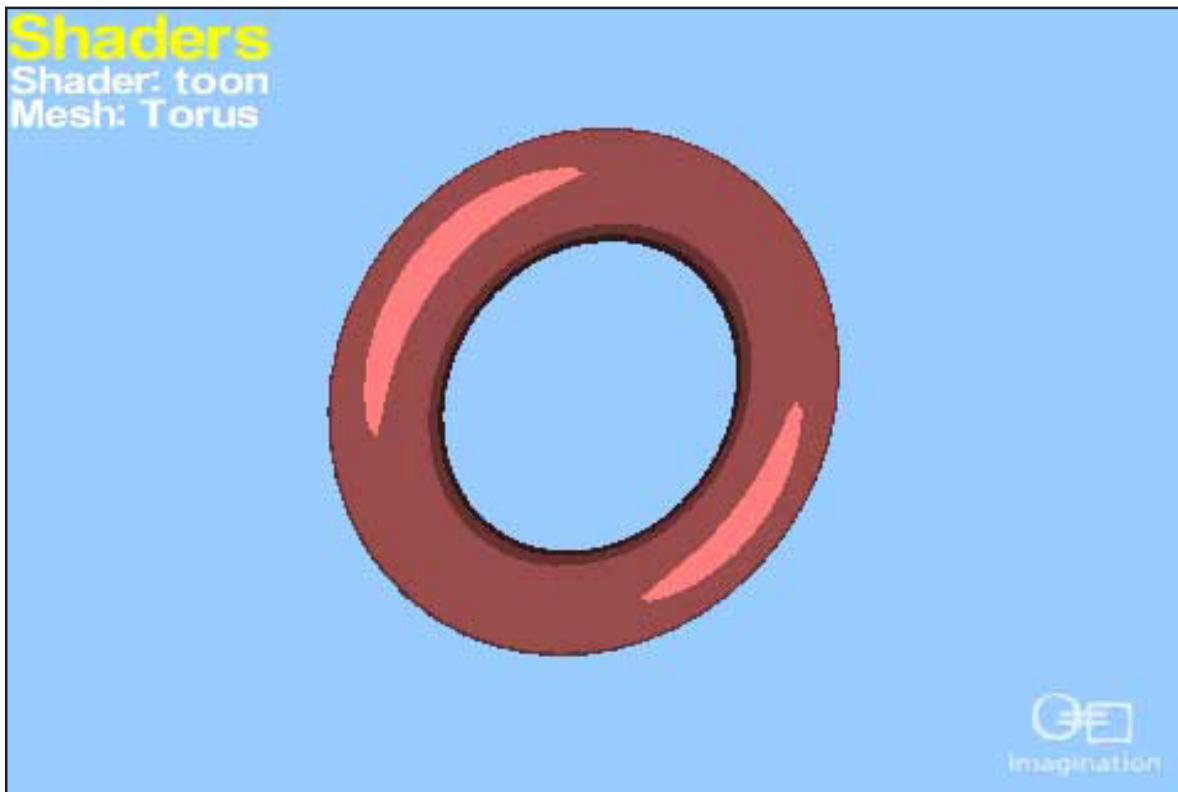


Note that this tutorial will be focusing on OpenGL ES 2.0 (not OpenGL ES 1.0).

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, vertexes, colors, cameras, 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 use 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 even 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 percentage of people with these devices is in the majority now, so it's 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 GLKEffects and GLKMath APIs lets you easily do a lot of the stuff that was built into OpenGL ES 1.0.

I'd say if you're new to OpenGL 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 tutorial should help get understand the basics! :]

Getting Started

Create a new project in Xcode and select the **iOS\Application\Empty Application Template**. We're selecting the Empty Application template (not the OpenGL Game template!) so you can put everything together from scratch and get a better idea how everything fits together.

Set the Product Name to **HelloGLKit**, make sure the Device Family is set to iPhone, make sure "Use Automatic Reference Counting" is checked but the other options are not, and click Next. Choose a folder to save your project in, and click Create.

If you run your app, you should see a plain blank Window:



The project contains almost no code at this point, but let's take a quick 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 switch over to the only class in the template, **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! About as "from scratch" as you can get.

Introducing GLKView

To get started with OpenGL ES 2.0, the first thing we need to do is add a subview to the window that does its drawing with OpenGL. If you've done OpenGL ES 2.0 programming before, you know that there was a ton of boilerplate code to get this working - things like creating a render buffer and frame buffer, etc.

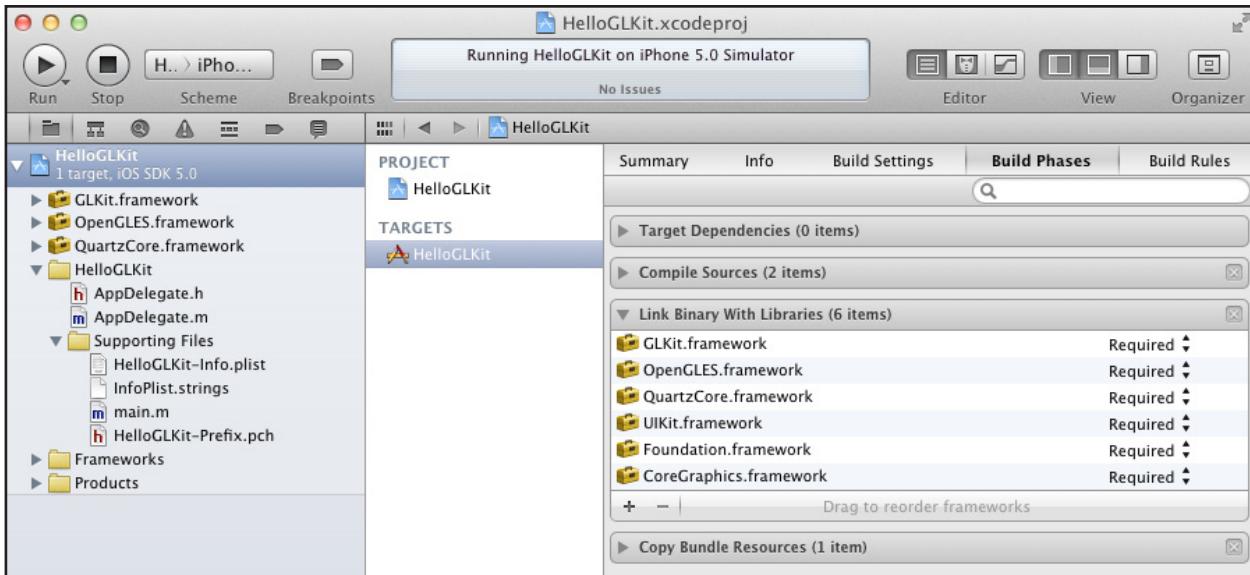
But now it's nice and easy with a new GLKit class called **GLKView**! Whenever you want to use OpenGL rendering inside a view, you simply add a **GLKView** (which is a normal subclass of **UIView**) and configure a few properties on it.

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. In this method you can put in your OpenGL 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 **Build Phases**, 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 to add a 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:kEAGLRenderingAPIOpenGL2]; // 1
    GLKView *view = [[GLKView alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]]; // 2
    view.context = context; // 3
    view.delegate = self; // 4
    [self.window addSubview:view]; // 5

    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 a OpenGL context.** To do anything with OpenGL, 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 Core Graphics context to do anything with Core Graphics.

When you create a context, you specify what version of the API you want to use. Here, you specify that you want to use OpenGL ES 2.0. If it is not available (such as if the program were run on an iPhone 3G), the app would terminate.

2. **Create a GLKView.** This creates a new instance of a [GLKView](#), and makes it as large as the entire window.
3. **Set the GLKView's context.** When you create a GLKView, you need to tell it the OpenGL context to use, so we specify the one we already created.
4. **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 class you specify here. We will implement this inside the App Delegate shortly to contain some basic OpenGL commands to paint the screen red.
5. **Add the GLKView as a subview.** This line adds the GLKView as a subview of the main window.

Since we marked the App Delegate as GLKView's delegate, we need to mark it as implementing the GLKViewDelegate protocol. So switch to **AppDelegate.h** and modify the @interface line as follows:

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

One step left! Switch back to **AppDelegate.m**, add the following code right before the @end:

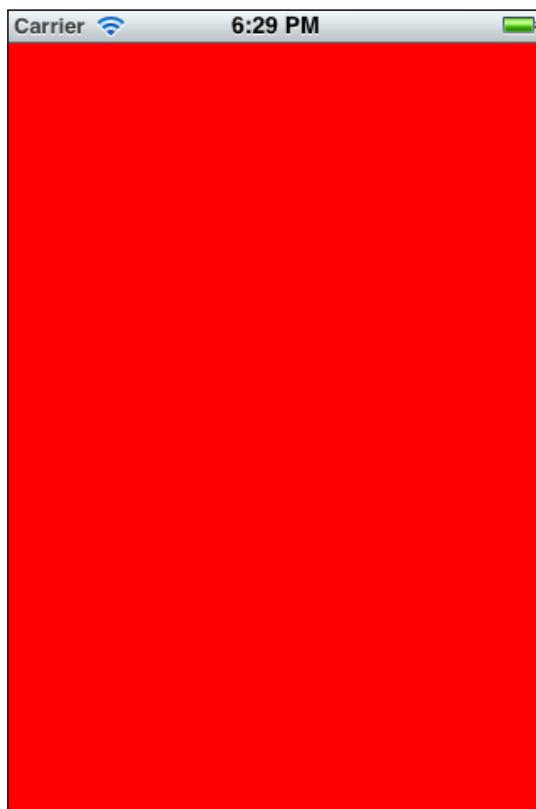
```
#pragma mark - GLKViewDelegate  
  
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {  
  
    glClearColor(1.0, 0.0, 0.0, 1.0);  
    glClear(GL_COLOR_BUFFER_BIT);  
  
}
```

The first line calls [glClearColor](#) to specify the RGB and alpha (transparency) values to use when clearing the screen. We set it to red here.



The second line calls `glClear` to actually perform the clearing. Remember that there can be different types of buffers, such as the render/color buffer we're displaying, and others we're not using yet such as depth or stencil buffers. Here we use the `GL_COLOR_BUFFER_BIT` to specify what exactly to clear: in this case, the current render/color buffer.

That's it! Compile and run, and with just 7 lines of code we have OpenGL rendering to the screen!



Those of you who are completely new to OpenGL might not be very impressed, but those of you who have done this before will be very happy with the convenience here.

GLKView Properties and Methods

We only set a few properties of `GLKView` here (`context` and `delegate`), but I wanted to mention the other properties and methods on `GLKView` that might be useful to you in the future.

This is an optional reference section and is for informational purposes only. If you want to keep coding away, feel free to skip to the next section! :]



context and delegate

We already covered these in the previous section, so I won't repeat here.

drawableColorFormat

Your OpenGL context has a buffer it uses to store the colors that will be displayed to 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

Your OpenGL 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 keeps a buffer kind of like a pixel 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 for 3D games), you should choose `GLKViewDrawableDepthFormat16` or `GLKViewDrawableDepthFormat24`.

The tradeoff here is with `GLKViewDrawableDepthFormat16` your app will use less resources, but you might have rendering issues when objects are very close to each other.

drawableStencilFormat

Another optional buffer your OpenGL 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 to be cast on the floor.



The default value for this property is `GLKViewDrawableStencilFormatNone`, which means there is no stencil buffer, but you can enable it by setting it to the only alternative - `GLKViewDrawableStencilFormat8`.

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 edges of geometry.

Be careful about setting this 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`.

drawableHeight/drawableWidth

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

This is a handy way to get a `UIImage` of the view's current context.

bindDrawable

OpenGL 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

`GLKView` and OpenGL 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 and display

I don't want to spoil the surprise - we'll explain these in the next section! :]

Updating the GLKView

Let's try to update our GLKView periodically, like we would in a game. How about we 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 variables as follows:

```
@implementation AppDelegate {
    float _curRed;
    BOOL _increasing;
}
```

And initialize these in application:didFinishLaunchingWithOptions:

```
_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 drawInRect is called, it updates the _curRed value a little bit based on whether it's increasing or decreasing. Note that this code isn't perfect, because it doesn't take into effect how long it takes between calls to drawInRect. This means that the animation might be faster or slower based on how quickly drawInRect is called. We'll discuss a way to fix this later in the tutorial.

Compile and run and... wait a minute, nothing's happening!

By default, the GLKView only updates itself on an as-needed basis - i.e. when views are first shown, the size changes, or the like. However for game programming, you often need to redraw every frame!

We can disable this default behavior of GLKView by setting enableSetNeedsDisplay to false. Then, we can control when the redrawing occurs by calling the display method on GLKView whenever we want to update the screen.

Ideally we would like to synchronize the time we render with OpenGL to the rate at which the screen refreshes.

Luckily, Apple provides an easy way for us to do this with CADisplayLink! It's really easy to use so let's just dive in. First add this import to the top of AppDelegate.m:

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

Then add these lines to application:didFinishLaunchingWithOptions:

```
view.enableSetNeedsDisplay = NO;
CADisplayLink* displayLink = [CADisplayLink
    displayLinkWithTarget:self selector:@selector(render:)];
[displayLink addToRunLoop:[NSRunLoop currentRunLoop]
    forMode:NSTimerDeliveryMode];
```

Then add a new render function as follows:

```
- (void)render:(CADisplayLink*)displayLink {
    GLKView * view = [self.window.subviews objectAtIndex:0];
    [view display];
}
```

Compile and run, and you should now see a cool pulsating "red alert" effect!

Introducing GLViewController

You know that code we just wrote in that last section? Well you can just forget about it, because there's a much easier way to do so by using GLViewController :]



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 writing that code, plus adds some extra neat features that you would have had to code yourself.

So 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]];
    view.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 it is easy to see what's no longer needed. There are also four new lines (marked with comments):

1. **Create a GLKViewController.** This creates a new instance of a [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 we set it to the one we already created.
3. **Set the GLKViewController's delegate.** We set the current class (AppDelegate) as the delegate of the GLKViewController. This means that the GLKViewController will notify us each frame so we can run game logic, or when the game pauses (a nice built-in feature of GLKViewController we'll demonstrate later).
4. **Set the preferred FPS.** The 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 long 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 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 framesPerSecond property.

5. **Set the rootViewController.** We want this view controller to be the first thing that shows up, so we add it as the rootViewController of the window. Note that we no longer need to add the view as a subview of the window manually, because it's the root view of the GLKViewController.

Notice that we no longer need the code to run the render loop and tell the GLView to refresh each frame - GLKViewController does that for us in the background! So go ahead and comment out the render method as well.

Also remember that we set the GLKViewController's delegate to the current class (AppDelegate), so let's mark it as implementing GLKViewControllerDelegate. Switch to **AppDelegate.h** and replace the @implementation with the following line:

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

The final step is to update the glkView:drawInRect method, so switch back to **AppDelegate.m** and add the implementation for GLKViewController's glkViewControllerUpdate callback:

```
#pragma mark - GLKViewControllerDelegate  
  
- (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 we moved the code to change the current color from the draw method (where it didn't really belong) to the update method (intended for game/app logic).

Also notice that we 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.

This is another of those convenient things GLKViewController does for you! We didn't have to write special code to store the time since the last update - it did it for us! There are some other time-based properties, but we'll discuss those later.

GLKViewController and Storyboards

So far, we've manually created the GLKViewController and GLKView because it was a simple way to introduce you to how they work. But you probably wouldn't want to do it 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 hierarchy!

So let's do a little refactoring to accomplish that. First, let's create a subclass of GLKViewController that we can use to contain our app's logic. So create a new file with the **iOS\Cocoa Touch\UIViewController subclass** template, name the class **HelloGLKitViewController**, as a subclass of **GLKViewController** (you can type this in even though it's not in the dropdown). Make sure Targeted for iPad and With XIB for user interface are both unselected, and create the file.



Open up HelloGLKitViewController.h, and import the GLKit header at the top of the file:

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

Then switch to **HelloGLKitViewController.m**, and add a private category on the class to contain the instance variables we need, and a new property to store the context:

```
@interface HelloGLKitViewController () {
    float _curRed;
    BOOL _increasing;

}
@property (strong, nonatomic) EAGLContext *context;

@end

@implementation HelloGLKitViewController
@synthesize context = _context;
```

Then implement viewDidLoad and viewDidUnload as the following:

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

    self.context = [[EAGLContext alloc]
        initWithAPI:kEAGLRenderingAPIOpenGLGLES2];

    if (!self.context) {
        NSLog(@"Failed to create ES context");
    }

    GLKView *view = (GLKView *)self.view;
    view.context = self.context;

}

- (void)viewDidUnload
{
    [super viewDidUnload];

    if ([EAGLContext currentContext] == self.context) {
        [EAGLContext setCurrentContext:nil];
    }
    self.context = nil;
}
```



In viewDidLoad, we create an OpenGL ES 2.0 context (same as we did last time in the App Delegate) and squirrel it away. Our root view is a GLKView (we know this because we will set it up this way in the Storyboard editor), so we cast it as one. We then set its context to the OpenGL context we just created.

Note that we don't have to set the view controller as the view's delegate - GLKView-Controller does this automatically behind the scenes.

In viewDidUnload, we just do the opposite to clean up. We have to make sure there's no references left to our context, so we check to see if the current context is our context, and set it to nil if so. We also clear out our reference to it.

At the bottom of the file, add the implementations of 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

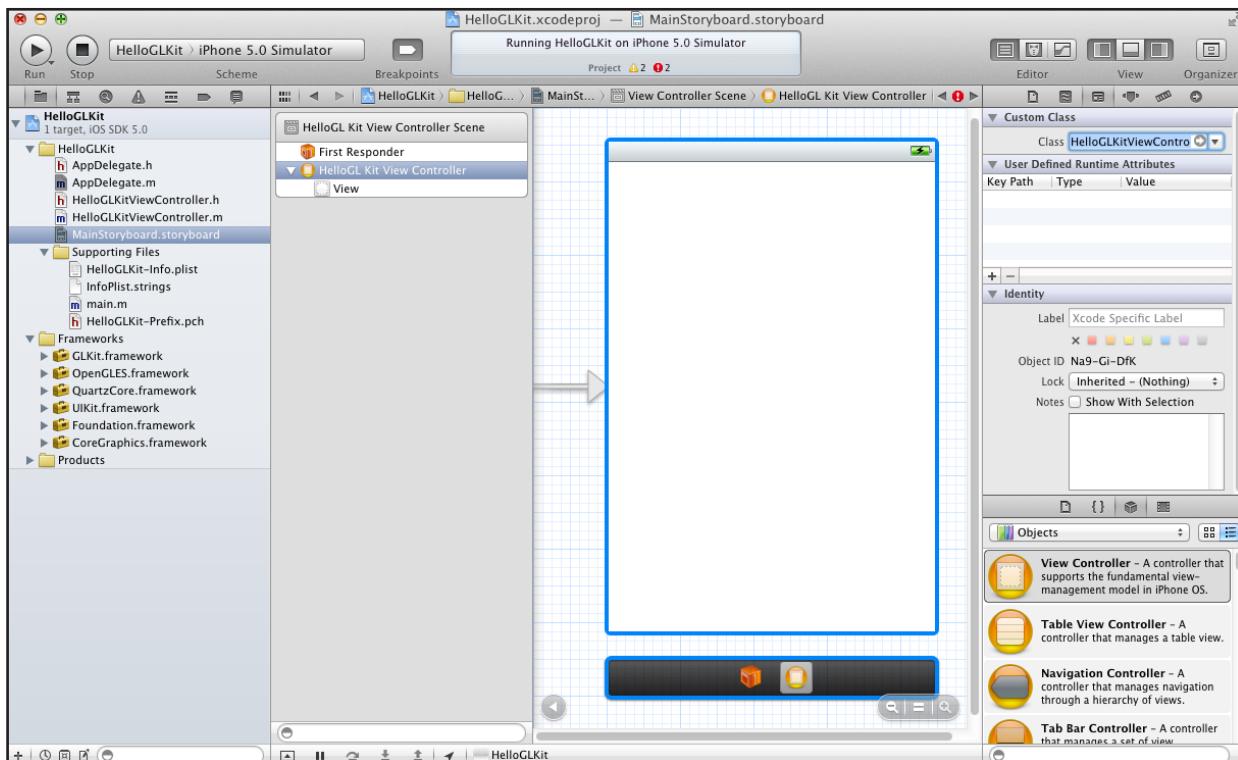
- (void)update {
    if (_increasing) {
        _curRed += 1.0 * self.timeSinceLastUpdate;
    } else {
        _curRed -= 1.0 * self.timeSinceLastUpdate;
    }
    if (_curRed >= 1.0) {
        _curRed = 1.0;
        _increasing = NO;
    }
    if (_curRed <= 0.0) {
        _curRed = 0.0;
        _increasing = YES;
    }
}
```

Note that the update method is called plain "update", because now that we're inside the GLKViewController we can just override this method instead of having to set a delegate. Also the timeSinceLastUpdate is accessed via "self", not a passed in view controller.



With this 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 View Controller into the grid area. Select the View Controller, and in the Identity Inspector set the class to **HelloGLKitViewController**:



Also, select the View inside the View Controller, and in the Identity Inspector set the Class to **GLKView**.

To make this Storyboard run on startup, open **HelloGLKit-Info.plist**, control-click in the blank area, and select Add Row. From the dropdown select Main storyboard file base name, and enter **MainStoryboard**.

That pretty much completes everything we need, but we still have some old code in **AppDelegate** that we need to clean up. Start by deleting the `_curRed` and `_increasing` instance variables from **AppDelegate.m**. Also delete the `glkView:drawInRect` and `glkViewControllerUpdate` methods.

And delete pretty much everything from `application:didFinishLaunchingWithOptions` so it looks like this:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
```

```
    return YES;  
}
```

And modify the AppDelegate @interface to strip out the two GLKit delegates since we aren't using those anymore:

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

That's it! Compile 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 (except it has a lot of other code in there you can just delete if you don't need it). Feel free to choose that in the future when you're creating a new OpenGL project to save a little time - but now you know how it works from the ground up!

GLKViewController and Pausing

Now that we're all nicely set up in a custom GLKViewController subclass, let's play around with one of the neat features of GLKViewController - pausing!

To see how it works, just add the following to the bottom of **HelloGLKitViewController.m**:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {  
    self.paused = !self.paused;  
}
```

Compile and run the app, and now whenever you tap the animation stops! 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 pauseOnWillResignActive property that is by default set to YES. This means when the user hits the home button or receives an interruption such as a phone call, your game will be automatically paused! Similarly, it has a resumeOnDidBecomeActive property that is by default set to YES, which means when the user comes back to your app, it will automatically unpause. Handy, that!

We've covered almost every property of GLKViewController by now, except for the extra time info properties I discussed 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 last time your game was unpause.

Let's add some code to try these out. Add the following code to the top of touchesBegan:

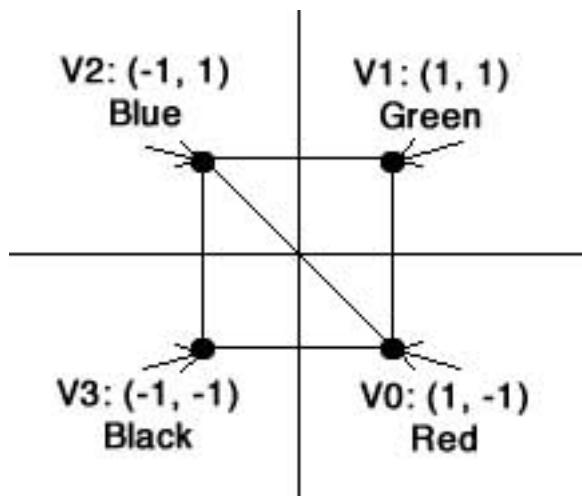
```
NSLog(@"%@", @"timeSinceLastUpdate: %f", self.timeSinceLastUpdate);
NSLog(@"%@", @"timeSinceLastDraw: %f", self.timeSinceLastDraw);
NSLog(@"%@", @"timeSinceFirstResume: %f", self.timeSinceFirstResume);
NSLog(@"%@", @"timeSinceLastResume: %f", self.timeSinceLastResume);
```

Play around with it so you're familiar with how they work. Then it's time for more fun stuff - rendering a square to the screen!

Creating Vertex Data for a Simple Square

Let's start things nice and simple by rendering a square to the screen. To do this we need to know four points for each of the corners. We often call a point in OpenGL space a vertex, and a set of points vertices.

The square will be set up like the following:



When you render geometry with OpenGL, it can't render squares-it can only render triangles. However we 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. Open up **HelloGLKitViewController.m** and create a plain old C-structure and a few arrays to keep track of our square 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
};
```

So basically we create:

- a structure to keep track of all our 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 like 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 in the diagram above.
- an array that gives a list of triangles to create, by specifying the 3 vertices that make up each triangle

We now have all the information we need, we 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 we have in the Vertices array), and one to keep track of the indices that make up triangles (like we have in the Indices array).

So first add the following instance variables to your private HelloGLKitViewController category:

```
GLuint _vertexBuffer;  
GLuint _indexBuffer;
```

Then add a method above viewDidLoad to create these:

```
- (void)setupGL {  
  
    [EAGLContext setCurrentContext:self.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 it does is set the current OpenGL 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 we need a vertex buffer to store all of the vertices for the square so OpenGL can draw our square for us.

Note that if you are unfamiliar with C, this syntax might look different to you, but this is calling a function called `glGenBuffers` with the first parameter set to 1, and the second parameter set to a pointer to the vertex buffer variable. It 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. We need to do this because we're about to tell OpenGL to do something with the vertex buffer we created.

Finally, we call `glBufferData` to send the data over to OpenGL-land. Although we have defined our vertices earlier in our own array, this memory is on the CPU, not in the graphics card. So we have to call this to move the data to the graphics card.

Also add a new method to delete the vertex and index buffers:

```
- (void)tearDownGL {  
    [EAGLContext setCurrentContext:self.context];  
  
    glDeleteBuffers(1, &_vertexBuffer);  
    glDeleteBuffers(1, &_indexBuffer);  
  
}
```

And before we forget, add this to the bottom of `viewDidLoad`:

```
[self setupGL];
```

And this inside `viewDidUnload`, right after the call to `[super viewDidUnload]`:

```
[self tearDownGL];
```

Almost done - we 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 them 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 the 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 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.** Usually when you make your own shaders, they take parameters called attributes and you write code to get their IDs. For GLKBaseEffect's built in shaders, these are already predefined as constants such as GLKVertexAttribPosition or GLKVertexAttribColor. So you need to enable any parameters that you want to pass in to the shaders, and give them pointers to data.
5. **Draw your geometry.** Once you have everything set up, you can use normal OpenGL 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 if you use them, you don't necessarily have to write any shaders at all! Of course you're still welcome to if you'd like - and you can mix and match and render some things with GLKBaseEffect, and some with your own shaders. If you look at the OpenGL template project, you'll see an example of exactly that!

In this book, we'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-by-one in code.



1. Create a GLKBaseEffect

The first step is to simply create a GLKBaseEffect. Up in your private HelloGLKit-ViewController category, add a property for a GLKBaseEffect:

```
@property (strong, nonatomic) GLKBaseEffect *effect;
```

And synthesize it after the @implementation below:

```
@synthesize effect = _effect;
```

Then in setupGL, initialize it right after calling [EAGLContext setCurrentContext:...]:

```
self.effect = [[GLKBaseEffect alloc] init];
```

And set it to nil at the bottom of tearDownGL:

```
self.effect = nil;
```

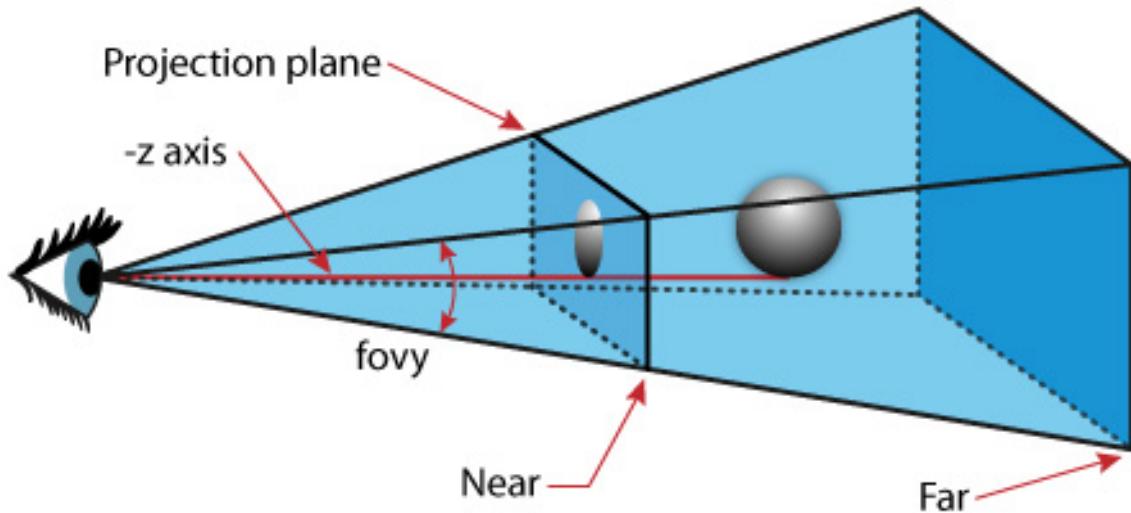
Now that we've created the effect, let's use it in conjunction with our vertex and index buffers to render the square. The first step is to set our effect's projection matrix!

2. Set GLKBaseEffect properties

The first property we 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 we'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 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 (i.e. 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 than 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);
self.effect.transform.projectionMatrix = projectionMatrix;
```

In the first line, we get the aspect ratio of the GLKView.

In the second line, we use a built in helper function in the GLKit math library to easily create a perspective matrix for us - all we have to do is pass in the parameters

discussed above. We set the near plane to 4 units away from the eye, and the far plane to 10 units away.

In the third line, we just set the projection matrix on the effect's transform property!

We 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 once 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);
self.effect.transform.modelviewMatrix = modelViewMatrix;
```

If you remember back to where we set up the vertices for the square, remember that the z-coordinate for each vertex was 0. If we 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 we need to do is to move this backwards. So in the first line, we use the GLKMatrix4MakeTranslation function to create a matrix for us that translates 6 units backwards.

Next, we want to make the square rotate for fun. So we increment an instance variable that keeps track of the current rotation (which we'll add in a second), and use the GLKMatrix4Rotate method to modify the current transformation by rotating it as well. It takes radians, so we use the GLKMathDegreesToRadians method to that conversion. Yes, this math library has just about every matrix and vector math routine you'll need!

Finally, we set the model view matrix on the effect's transform property.

Before we forget, add the rotation instance variable to your HelloGLKitViewController's private category:

```
float _rotation;
```

We'll play around with more GLKBaseEffect properties later, since there's a lot of cool stuff and we've barely scratched the surface here. But let's continue on for now, so we can finally get something rendering!



3. Call `prepareToDraw` on the `GLKBaseEffect`

For this step, add the following line to the bottom of `glkView:drawInRect:`:

```
[self.effect prepareToDraw];
```

w00t! Just remember that you need to call this after any time you change properties on a `GLKBaseEffect`, before you draw with it.

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've programmed with OpenGL ES 2.0 before this will look familiar to you, but if not let me explain.

Every time before you draw, you have to tell OpenGL which vertex buffer objects you should use. So here we bind the vertex and index buffers we created earlier. Strictly, we didn'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 you use many different vertex buffer objects.

Next, we have to enable the pre-defined vertex attributes we want the `GLKBaseEffect` to use. We use the `glEnableVertexAttribArray` to enable two attributes here - one for the vertex position, and one for the vertex color. GLKit has predefined constants we need to use for these - `GLKVertexAttribPosition` and `GLKVertexAttribColor`.

Next, we 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. We just use the pre-defined 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 we can simply pass in sizeof(Vertex) here to get the compiler to compute it for us.
- The final parameter is the offset within the structure to find this data. We can use the handy offsetof operator to find the offset of a particular field within a structure.

So now that we're passing on 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:

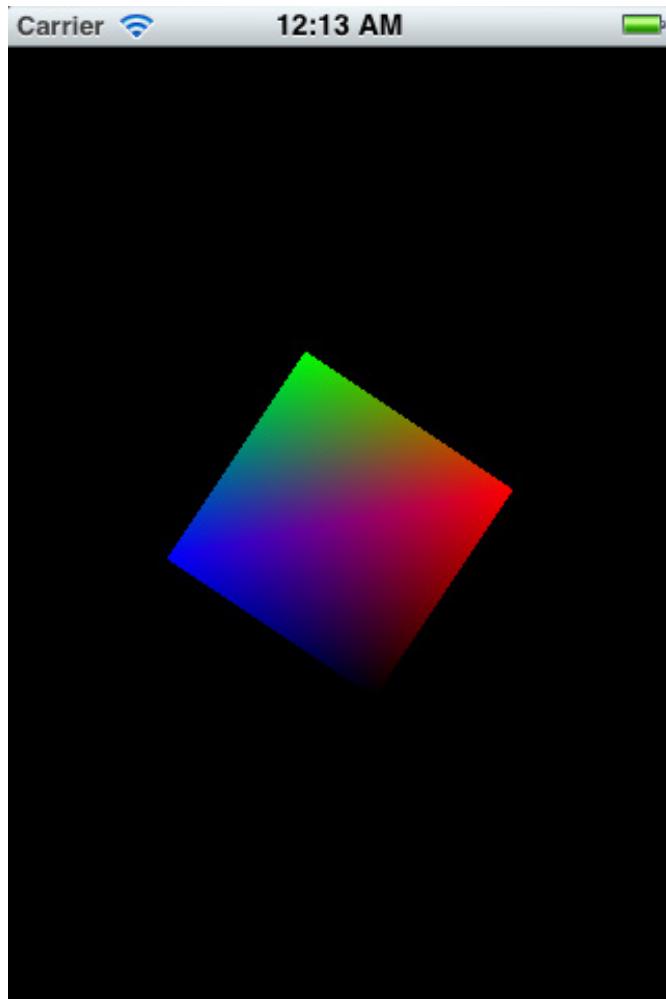
```
glDrawElements(GL_TRIANGLES, sizeof(Indices)/sizeof(Indices[0]),
               GL_UNSIGNED_BYTE, 0);
```

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. 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. We use a C trick to compute the number of elements in an array here by dividing the sizeof(Indices) (which gives us the size of the array in bytes) by sizeof(Indices[0]) (which gives us the size of the first element in the array).
- The third parameter is the data type of each individual index in the Indices array. We're using an unsigned byte for that so we specify that here.
- From the documentation, it appears that the final parameter should be a pointer to the indices. But since we're using vertex buffer objects it's a special case - it will use the indices array we 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 with 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 and index buffers and vertex attributes.

However, there's more cool stuff in store for you with GLKit. Keep reading for the next chapter, where we'll move to full 3D, and demonstrate some of the cool effects you can get with `GLKitBaseEffect`, such as lighting, fog, and more!



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 GLKEffects.

In this chapter, we're going to take things to full 3D and convert our 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

Before we begin, I wanted to introduce you to an optimization we can make to our code to make it perform better.

In the last chapter, we rendered our square with vertex buffer objects and index buffer objects, and bound to these objects each time we wanted to draw. This works, but isn't ideal.

If we were drawing a lot of different geometry to the scene, our code would get quite tedious. Every time we want to draw something, we'd have to bind the correct vertex and index buffers, enable the attributes we want for the shader, and specify where their data is located.

Continually making all these calls each time is also slow. To speed things up, Apple recommends using 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, let's try them out. First add a new instance variable to your HelloGLKitViewController private category:

```
GLuint _vertexArray;
```

Then modify your setupGL method as follows:

```
- (void)setupGL {  
  
    [EAGLContext setCurrentContext:self.context];  
  
    self.effect = [[GLKBaseEffect alloc] init];  
  
    // 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 draw)  
    glEnableVertexAttribArray(GLKVertexAttribPosition);  
    glVertexAttribPointer(GLKVertexAttribPosition, 3, GL_FLOAT,  
                         GL_FALSE, sizeof(Vertex), offsetof(Vertex, Position));  
    glEnableVertexAttribArray(GLKVertexAttribColor);  
    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,  
                         GL_FALSE, sizeof(Vertex), offsetof(Vertex, Color));  
  
    // New line  
    glBindVertexArrayOES(0);  
  
}
```

The first group of new lines creates a new vertex array object and binds to it. The rest of the setup calls we make will be stored in the vertex array object.

After setting up the vertex and index buffer, we add the lines of code to set up the vertex attributes that were previously in draw. This configuration is also stored in the vertex array object.



Finally, we bind the current vertex array object to 0 since we'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);  
  
    [self.effect prepareToDraw];  
  
    glBindVertexArrayOES(_vertexArray);  
    glDrawElements(GL_TRIANGLES, sizeof(Indices)/sizeof(Indices[0]),  
        GL_UNSIGNED_BYTE, 0);  
}
```

Note how much simpler it is! It's faster too, so you should get into the habit of using these. If you compile and run, it should work as normal.

Gratuitous Textures

There's one big aspect of GLKit that we haven't played around with - texture loading!

This was one of the new features I was most excited about, because the code to do this used to be so horrendous, especially when you want 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 setupGL, right after creating your self.effect:

```
NSDictionary * options = [NSDictionary dictionaryWithObjectsAndKeys:  
    [NSNumber numberWithBool:YES],  
    GLKTextureLoaderOriginBottomLeft,  
    nil];  
  
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]);
}
self.effect.texture2d0.name = info.name;
self.effect.texture2d0.enabled = true;

```

This code loads the tile_floor.png image from the texture and sends it to OpenGL as a texture that we can use to render with. All we had to do was use the textureWithContentsOfFile method on the GLKTextureLoader singleton and pass in a path, and we get back a class with information about the texture, including the OpenGL name we can use for rendering.

Note we pass a dictionary of options into the GLKTextureLoader. By default when you load a texture the origin is the upper left, but that's annoying because in OpenGL the origin is the lower left. So here I set the flag to make the texture coordinates match up to OpenGL coordinates.

Also notice that after we load the texture, we set a few more properties on the effect to set up the texture for use. We 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. So let's modify our Vertex structure and array to include texture coordinates:

```

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}}
};

```

And 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 we've covered already in this book. 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 of the Introducing GLKBaseEffect section in the previous chapter.

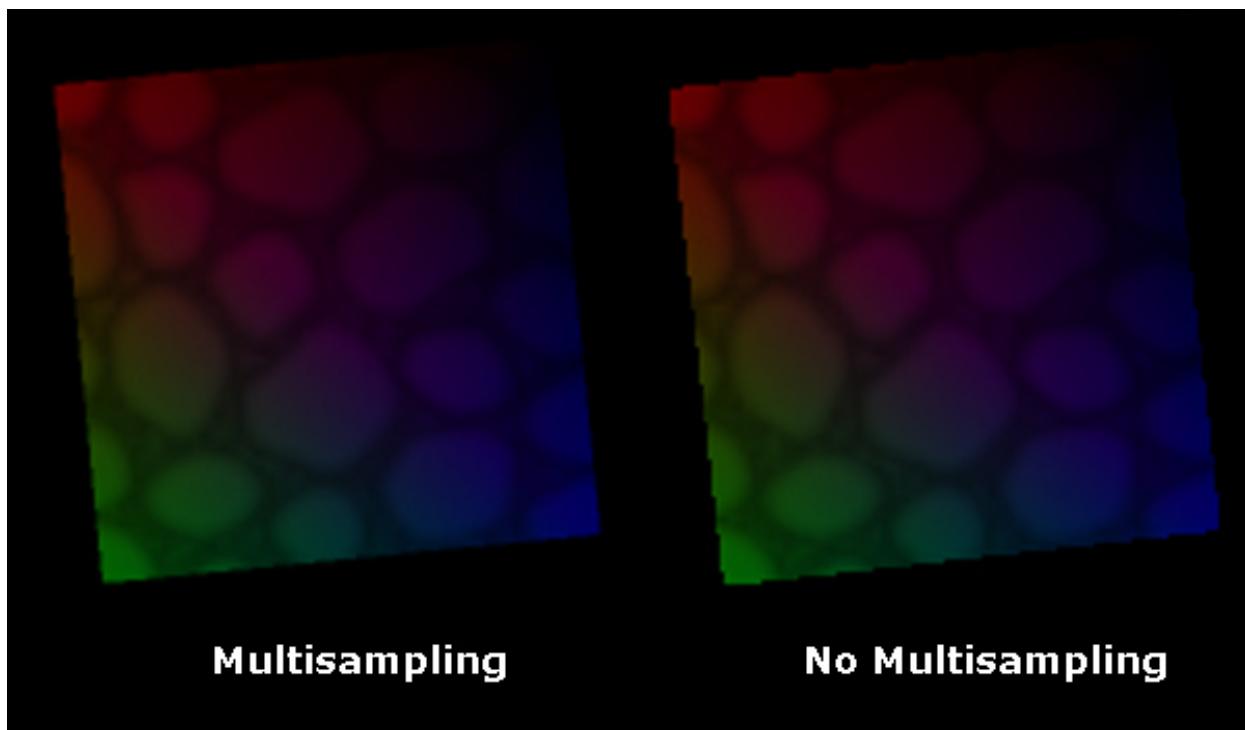
Gratuituous Multisampling

Just wanted to show you another reason GLKit is awesome. Run the app and watch the square rotate, and see if you notice jaggedy lines.

Then add the following line of code into viewDidLoad, right before the call to [self setupGL]:

```
view.drawableMultisample = GLKViewDrawableMultisample4X;
```

This single line enables multisampling (as you already know if you read the optional GLKit properties section earlier in this tutorial!) This renders your pixels as finer levels of granularity and merges the results to get an anti-aliased look, as you can see below:



Run the app again and see if you notice the difference. Not bad for one line, 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 we'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!

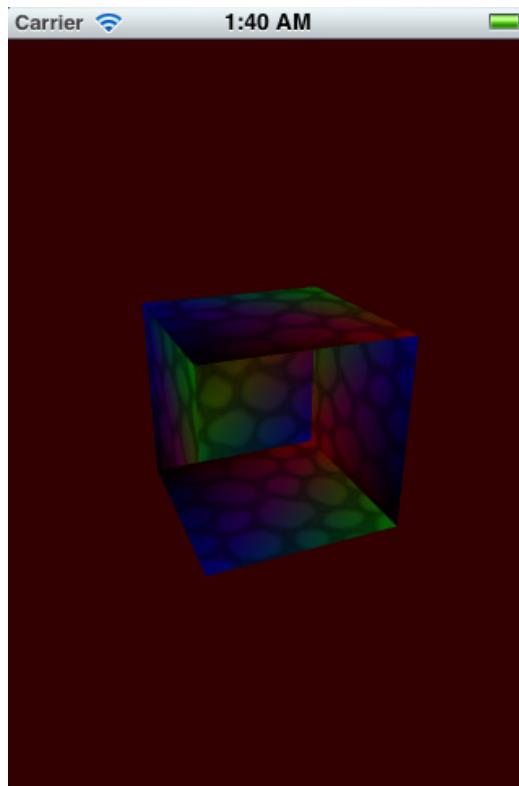
However if you do this note one tricky bit - you need to make sure you specify the indices in counter-clockwise order (from the perspective of looking at the box from the outside), for reasons I'll get into shortly.

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);
self.effect.transform.modelviewMatrix = modelViewMatrix;
```

Compile 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 currently we 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 inside right side face on top of where the front face should be, or similarly the back face on top of where the front face should be.

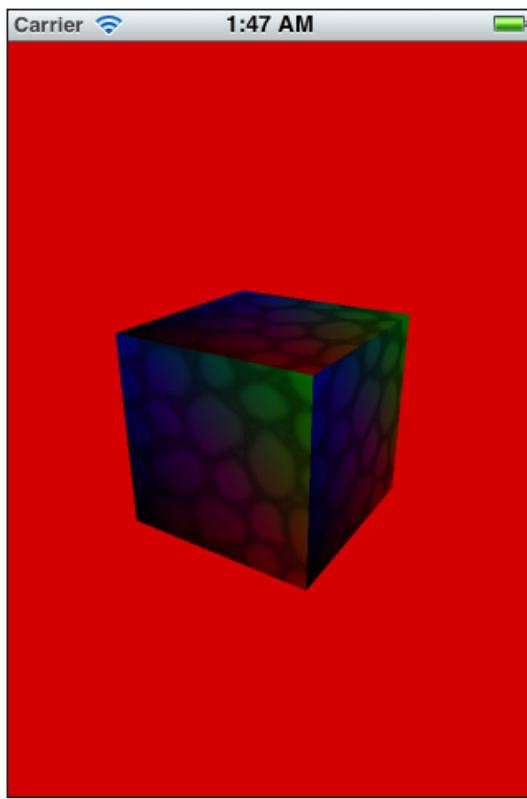
Luckily that is an easy fix. Since we were careful to define the vertices in counter-clockwise order, we can enable an OpenGL flag called `GL_CULL_FACE`. This makes OpenGL just skip drawing any frame that is backwards facing (i.e. not visible, since we have a cube!)

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 GLView depth buffer. However, using backface culling like this is much more optimal since it doesn't have to draw the back facing triangles at all. This can be a big performance improvement for more complicated games. Depth testing is necessary when you have geometry that may cover other objects.

Compile and run - and it works, our rotating textured 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 we have set just 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`, and are instances of `GLKEffectPropertyLight`. The `GLKEffectPropertyLight` class in turn has many properties you can set to configure how the light works.

Let's get a simple light working, then we'll dive deeper into all the various ways you can configure it.

Inside `setupGL`, add the following lines of code after enabling the texture on the effect:

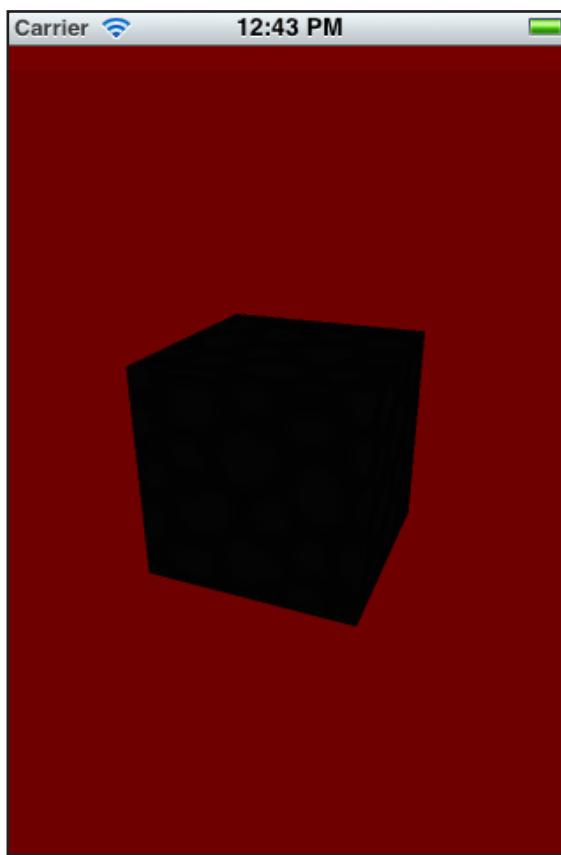


```
self.effect.light0.enabled = GL_TRUE;  
self.effect.light0.diffuseColor = GLKVector4Make(1, 1, 1, 1.0);  
self.effect.light0.position = GLKVector4Make(1, 1, 0, 1);
```

Here we set three properties on the light:

- **enabled**: Turns the light on (default is off).
- **diffuseColor**: We'll talk more about what this means later, but for now just think of this as the color of the light.
- **position**: The last component of this vector indicates whether the light is positional (1) or directional (0). We'll talk more about the differences between this later, but for now this means that we've set the position of the light at (1, 1, 0) in 3D space.

w00t we're done, right? Compile and run and you'll see this:



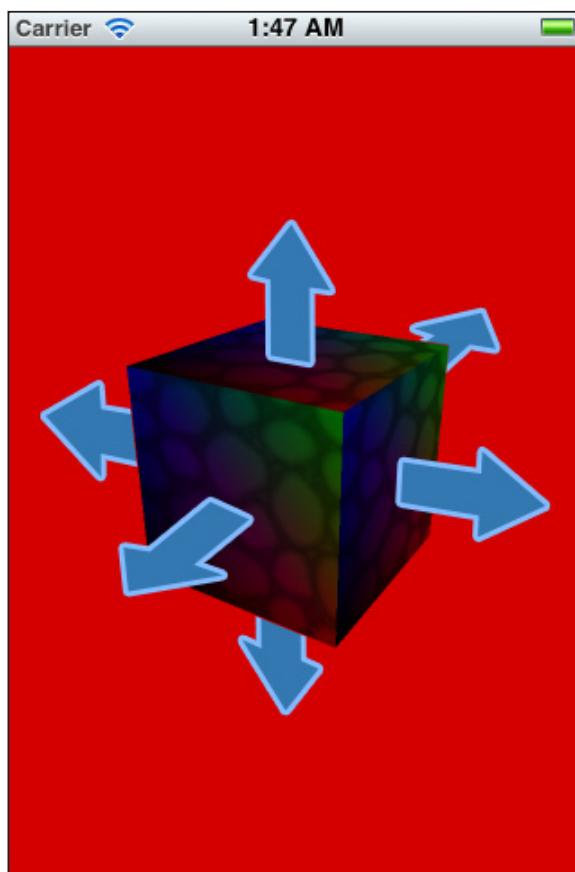
Sadly, it appears that we have no light at all. What gives?

Well this introduces our next topic - 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 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 our 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 our simple cube, the vertex normals will be the same as the surface normal for each vertex on the face.

To add vertex normals for each vertex, we need to update our 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 go through the normals for each surface, and make sure you understand why the normal is set the way it is.

Now that we have this new information in our Vertex structure, we 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));

```

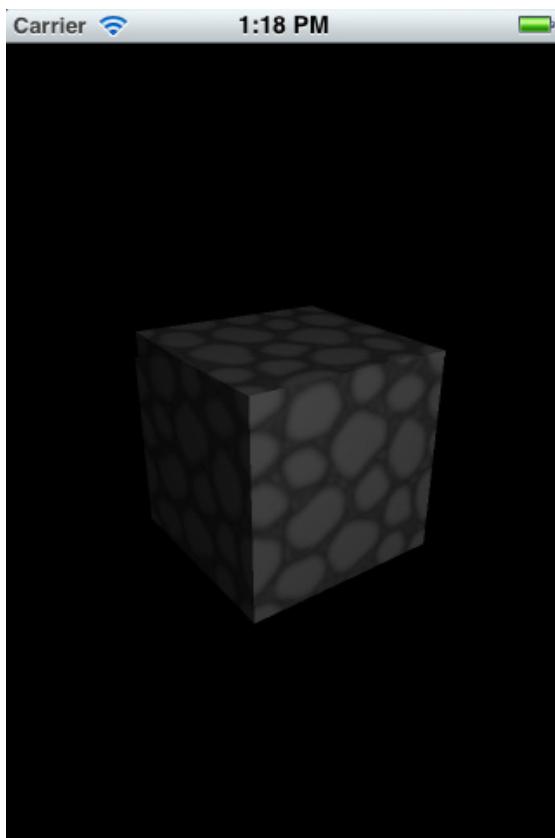
This should be familiar to you from the last tutorial. Try to go through each pa-



parameter in the above methods and make sure you understand what they do. If you don't recall, refer back to the last tutorial for a full explanation.

One more thing. As we're playing with lighting the pulsating effect is going to be confusing. So comment out the code in update that modifies the _curRed value to disable that.

Compile and run your code - and guess what, you've got lighting - all without writing a single line of shader code!



Note: You may notice that our cube appears gray (the color of the tile_floor texture) rather than our neat rainbow-colored cube as before. It turns out that when you enable lighting, by default the GLKBaseEffect shader will ignore any color values you pass in, and instead use a property on the light called the surface material, which we'll discuss later. If this is not what you want (and you want the color values you pass in used instead), you can set the self.effect.colorMaterialEnabled property to YES (and 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 particular 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 makes objects feel 3D.
- **Ambient light:** This is 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 so 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 a shiny spot on a marble).

When you set up a light, you can specify the colors for the diffuse, ambient, and specular types independently. To make things even more complicated, you can also set the diffuse, ambient, and specular types on the "material" of the geometry itself. The colors of the light and the 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:

```
self.effect.light0.enabled = GL_TRUE;
self.effect.light0.diffuseColor = GLKVector4Make(0, 1, 1, 1);
self.effect.light0.ambientColor = GLKVector4Make(0, 0, 0, 1);
self.effect.light0.specularColor = GLKVector4Make(0, 0, 0, 1);

self.effect.lightModelAmbientColor = GLKVector4Make(0, 0, 0, 1);
self.effect.material.specularColor = GLKVector4Make(1, 1, 1, 1);

self.effect.light0.position = GLKVector4Make(0, 1.5, -6, 1);
self.effect.lightingType = GLKLightingTypePerPixel;
```

In the first section we set up the three types of colors on the light. We set the diffuse color to a teal color, and set the ambient and specular colors to black (which is the equivalent of "no light").

We then set the lightModelAmbientColor, which defines the global ambient light in the scene. For demonstration purposes, we want to make sure that no light is affecting our cube except what we specifically set up on light 0, so we turn this off by setting it to black/"no light". The default values is {0.2, 0.2, 0.2, 1.0} by the way.

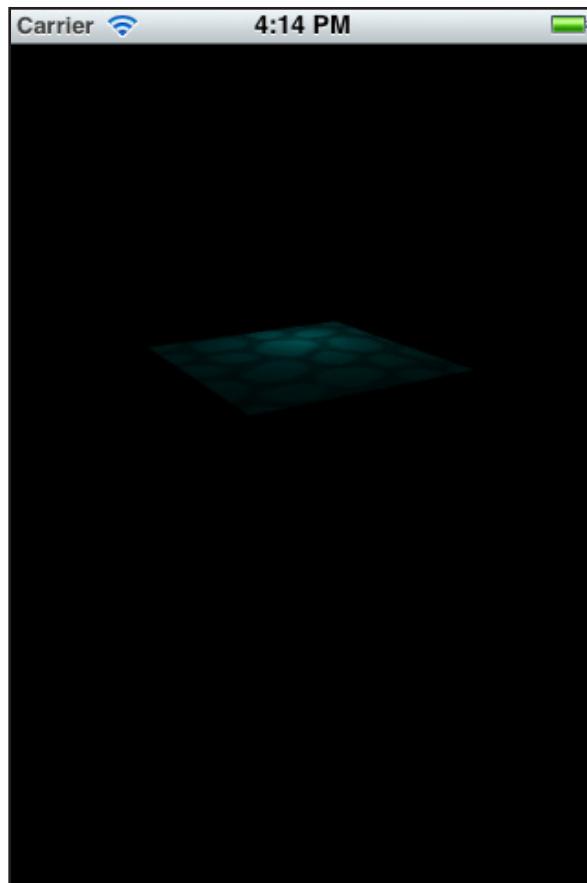
We then set the specular color of the material to white/"full light". It is important to set this, because the default value is $\{0, 0, 0, 1\}$, which means that there would be no specular highlight at all, even if we set the light's specularColor.

We leave the material's diffuse and ambient colors at their default values ($\{0.8, 0.8, 0.8\}$ and $\{0.2, 0.2, 0.2\}$ respectively).

We move the position of the light to be right above where the box is (remember the box is 1 unit tall, and we translate the box backwards 6 units so it's visible).

Finally, we 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 trade-off is increased processing time.

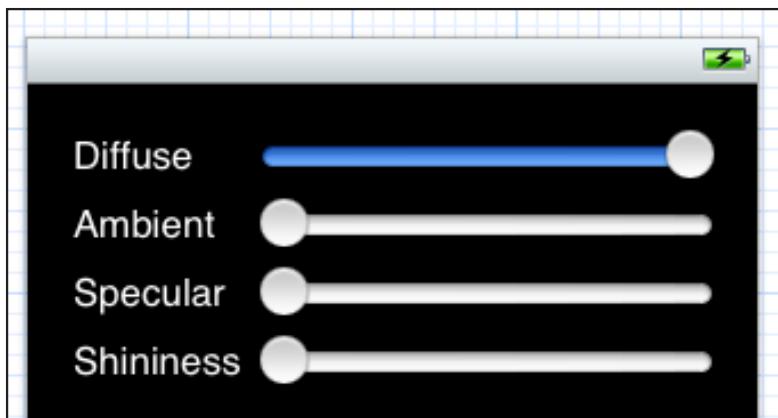
Compile and run your code, and now you should see a cool teal light shining down on top of the box:



Since we 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 we can get a better feel of how they work. Rather than having to continuously tweak these and recompile, let's add some sliders to our view controller so we 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's 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.h is visible, and control-drag from the diffuse slider down to below the @interface. 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 switch to HelloGLKitViewController.m and implement these methods as the following:

```
- (IBAction)diffuseChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    self.effect.light0.diffuseColor =
        GLKVector4Make(0, slider.value, slider.value, 1);
}
```

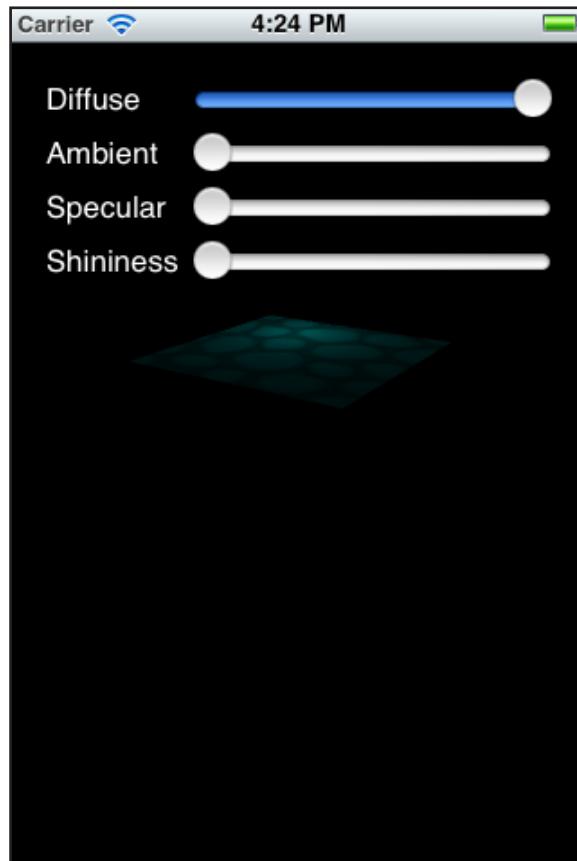
```
- (IBAction)ambientChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    self.effect.light0.ambientColor =
        GLKVector4Make(slider.value, slider.value, slider.value, 1);
}

- (IBAction)specularChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    self.effect.light0.specularColor =
        GLKVector4Make(slider.value, slider.value, slider.value, 1);
}

- (IBAction)shininessChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    self.effect.material.shininess = slider.value;
}
```

The callbacks simply update the various properties on the material. We've seen the first three before but we haven't seen shininess yet. The higher the shininess value, the more "focused and reflective" the specular component appears.

To see what I mean, let's play around! Compile 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

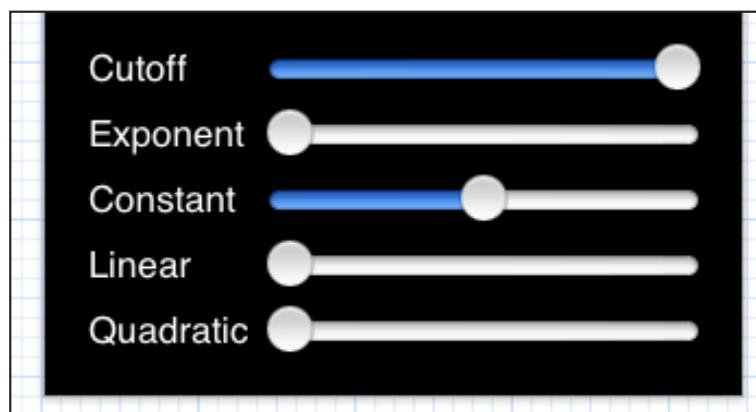
You can easily configure a light to work as a spotlight by setting the following properties:

- **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. We'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.

Another set of properties that are useful to set on a light 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. We'll just add some sliders for these so you can play around with them and get a feel for how they work - but if you want to see the equation that is used behind the scenes, see the `GLKEffect-PropertyLight Class Reference`.



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 as you did before. Here's 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)constantValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    self.effect.light0.constantAttenuation = slider.value;
}

- (IBAction)linearValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
    self.effect.light0.linearAttenuation = slider.value;
}

- (IBAction)quadraticValueChanged:(id)sender {
    UISlider * slider = (UISlider *)sender;
```

```

        self.effect.light0.quadraticAttenuation = slider.value;
    }

    - (IBAction)cutoffValueChanged:(id)sender {
        UISlider * slider = (UISlider *)sender;
        self.effect.light0.spotCutoff = slider.value;
    }

    - (IBAction)exponentValueChanged:(id)sender {
        UISlider * slider = (UISlider *)sender;
        self.effect.light0.spotExponent = slider.value;
    }
}

```

Here we simply set each of these values so we can experiment with them. That's it - compile 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 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

One more thing to show you with lights, then we're done covering those for now.

Let's add one more light to our scene, that revolves around our cube, just for fun!

First add a new instance variable for the light rotation to the HelloGLKitViewController private category:

```
float _lightRotation;
```

Then initialize a new light inside setupGL with the following properties:

```

self.effect.light1.enabled = GL_TRUE;
self.effect.light1.diffuseColor = GLKVector4Make(1.0, 1.0, 0.8, 1.0);
self.effect.light1.position = GLKVector4Make(0, 0, 1.5, 1);

```

Note that we make the light yellowish, and we set the base position to be 1.5 along the z-axis.

Then add the following code inside update, right after setting the projection matrix:

```
GLKMatrix4 lightModelViewMatrix =
```

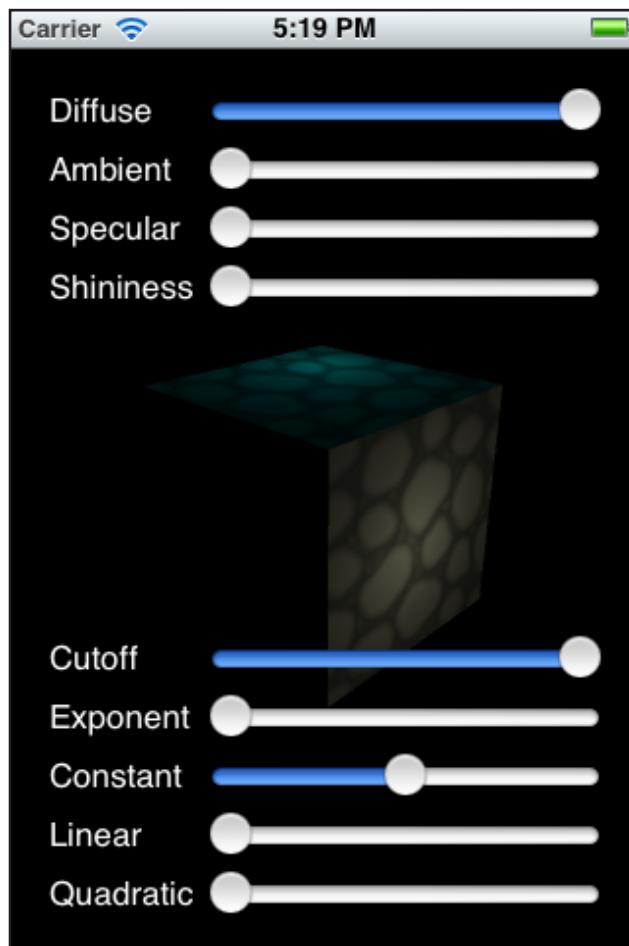


```
GLKMatrix4MakeTranslation(0.0f, 0.0f, -6.0f);
_lightRotation += -90 * self.timeSinceLastUpdate;
lightModelViewMatrix = GLKMatrix4Rotate(lightModelViewMatrix,
    GLKMathDegreesToRadians(25), 1, 0, 0);
lightModelViewMatrix = GLKMatrix4Rotate(lightModelViewMatrix,
    GLKMathDegreesToRadians(_lightRotation), 0, 1, 0);
self.effect.transform.modelviewMatrix = lightModelViewMatrix;
self.effect.light1.position = GLKVector4Make(0, 0, 1.5, 1);
```

The first thing we do is to construct a model view matrix to transform the light's position each frame. We start by moving the light backwards so it stays near the box. Then we make it rotate each frame - but the opposite direction the cube is rotating.

We 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.

Compile and run, and you should see a sweet rotating light around the cube!



Congratulations, you now have hands-on experience with every property 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 a surface that is toward the light (with respect to the surface normal). If you want light to be applied to both sides, you can set this to TRUE. But note the trade-off is decreased performance.

Multi-Texturing

The multi-texturing support in GLKBaseEffect is quite limited, but let's take a look to see what it can do.

In the resources for this chapter, drag the item_powerup_fish.png into your project. Then add the following code into setupGL (after loading the first texture):

```
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]);
}
self.effect.texture2d1.name = info.name;
self.effect.texture2d1.enabled = true;
self.effect.texture2d1.envMode = GLKTextureEnvModeDecal;
```

Here we're loading a texture with GLKTextureLoader just like we did before, except this time we are setting it into the texture2d1 property (the only other texture slot available in GLKBaseEffect).

The other difference is we 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, also add these lines of code right before the call to glBindArrayOES(0):

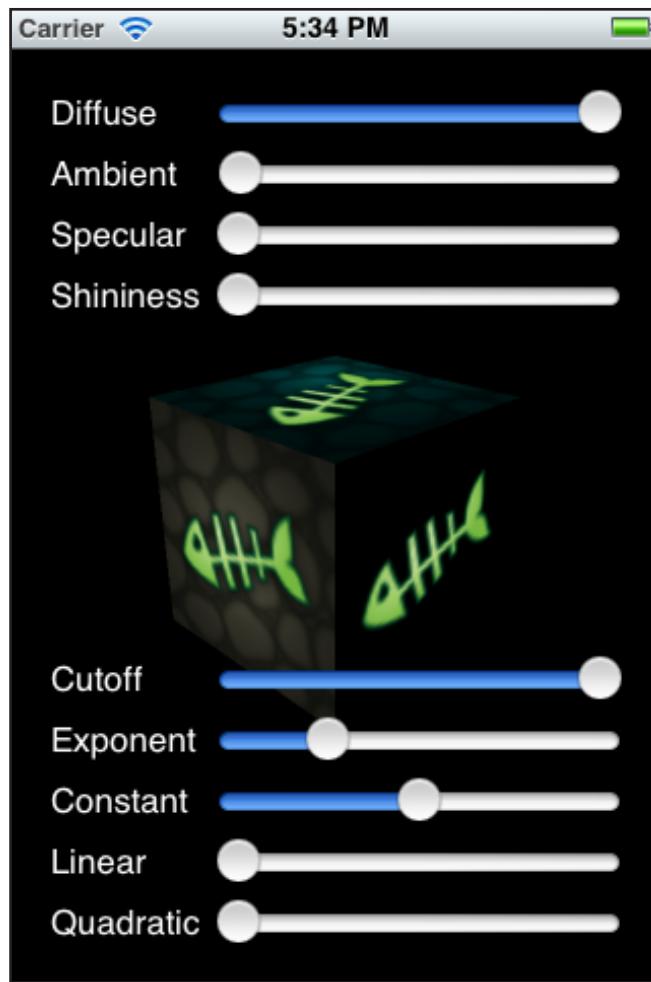
```
glEnableVertexAttribArray(GLKVertexAttribTexCoord1);
glVertexAttribPointer(GLKVertexAttribTexCoord1, 2, GL_FLOAT,
    GL_FALSE, sizeof(Vertex),
    (const GLvoid *) offsetof(Vertex, TexCoord));
```

Here we pass in the texture coordinates for the second texture. Note I used the



same texture coordinates as the first texture to make things easy, but you could pass in different coordinates if you needed to.

Compile and run, and you'll see glowy fish on top of your textures:



So one thing to note 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 faraway objects more smooth (instead of having them randomly disappear!) It can also be fun just for a neat fog effect :]

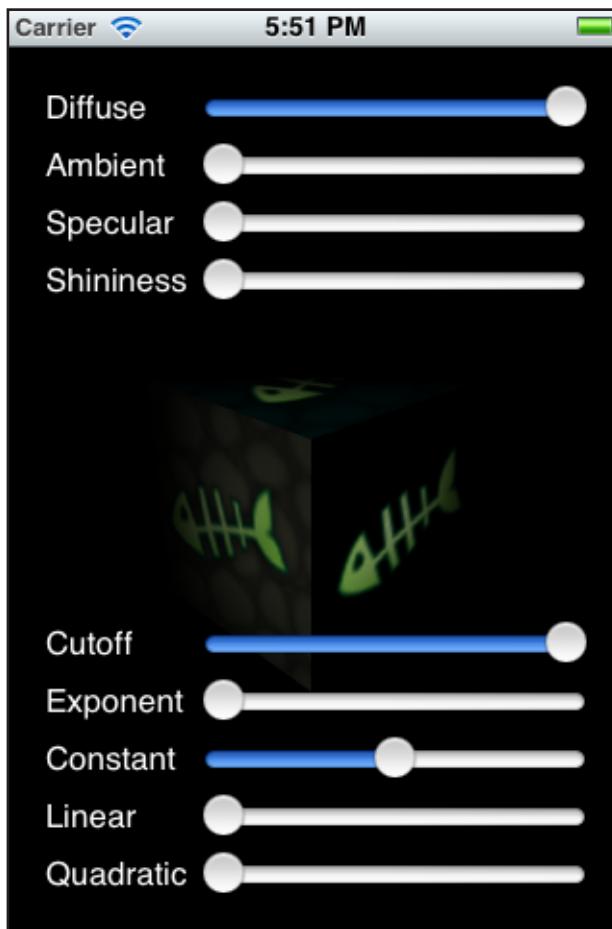


Using it is simple. Add the following code somewhere in `setupGL`:

```
self.effect.fog.color = GLKVector4Make(0, 0, 0, 1.0);
self.effect.fog.enabled = YES;
self.effect.fog.end = 5.5;
self.effect.fog.start = 5.0;
self.effect.fog.mode = GLKFogModeLinear;
```

Here we make the color of the fog black (since our background is black in this app). We set the fog as enabled mark where the fog of wall begins and ends (in amount of units from the eye) and make the mode of fog linear.

That's it - compile and run, and you should see a cool fog effect!



Note that the fog doesn't occur on a per pixel basis, it occurs on a per vertex basis. So you might see some strange "popping" as the cube rotates in and out of the fog. For most cases this won't really matter though, because usually you have your fog cloud far enough away that you wouldn't notice something like this.

Guess what - that pretty much covers everything you can do with `GLKBaseEffect`! There are two more properties that let you set the vertex colors to a constant color

rather than passing in the color vertex attribute like we've been doing (`useConstantColor` and `constantColor`), but other than that you're set!

Where To Go From Here?

Sadly this is a book on iOS5, 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 book I know of at the moment about OpenGL ES 2.0 is iPhone 3D Programming by Philip Rideout. Also, Erik Buck and Jeff LaMarche are working some new books about OpenGL ES 2.0 that look quite promising, so keep an eye out for those for your future studies.

Finally, I have some tutorials about using OpenGL ES 2.0 on raywenderlich.com. They do not cover GLKit, but they do show you the basics of writing your own vertex and fragment shaders, a technique you will find useful when you need to do something that GLKitBaseEffect does not support.

Best of luck with your future adventures with OpenGL ES 2.0 and GLKit, I hope these chapters have helped get you started on your journey!



10 Beginning UIKit Customization

by Steve Baranski

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, those dark days are over! iOS 5 has included many new APIs you can use to easily customize the appearance of various UIKit controls.

To illustrate some of these new APIs, in this tutorial series we'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 we can focus on the meat of this tutorial 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 XIBs. You'll see that the primary view presents a list of our 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 we 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, starting with **DetailView.xib**. 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**

In this chapter, we'll completely customize the detail screen to give it some style, using the new APIs available in iOS 5. In the next chapter, we'll customize the plain vanilla table view and port it to the iPad as well!

So with an idea of what's in store, let's convert this app from "zero" to "hero".

Adding a Background Image

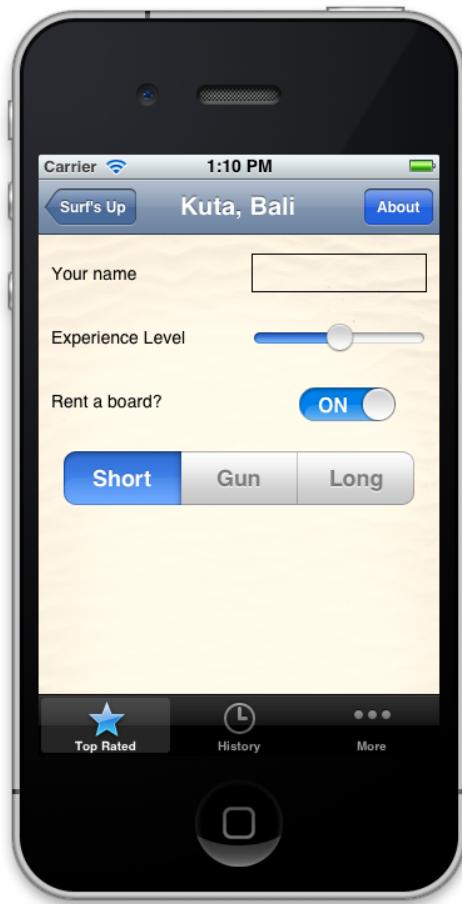
If you open up the Images folder in your project, you'll see that we already have some images we can use to customize the UI included in the project - we just need to modify the code to make use of them. Inside the images folder is a file called **bg_sand.png**. We're going to start our UI customization 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] setBackgroundColor:[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 can feel the sand between our toes already!

Customizing UINavigationBar

If you look inside the images folder, you'll see two images that we want to use to customize the navigation bar: **surf_gradient_textured_32.png** and **surf_gradient_textured_44.png**.

We 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 we can use to set a custom background image like this.
- **UIImage** has a new **resizableImageWithCapInsets** method we 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.

We could go into the detail view and use these new APIs to set the navigation bar's background image directly. But then we'd have to go and do the same thing inside our list view, and any other views we might have in our 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, we'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.



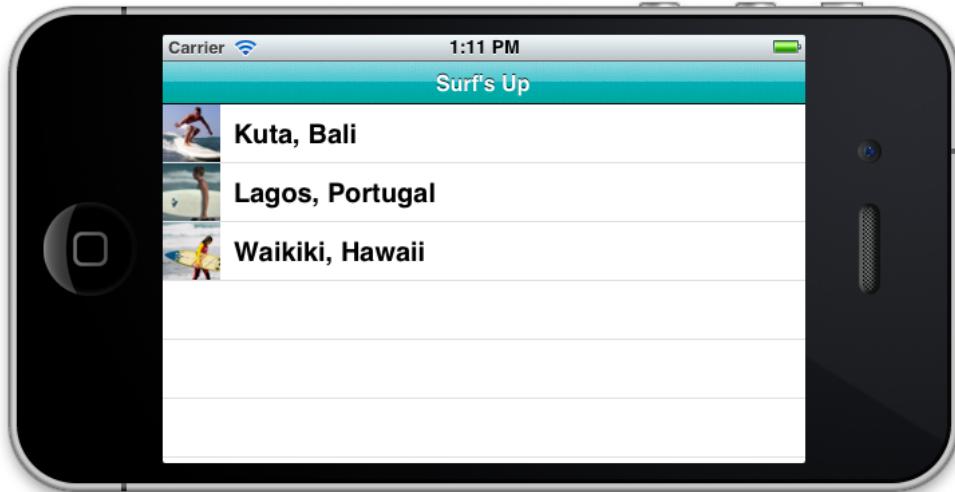
For the cap insets, you basically specify the fixed region of a given image in top, left, bottom, right. What's left is stretched over the remainder of the region to which the image is applied. In this particular image we want the whole thing stretched, so we pass 0 for all of the fixed caps. The next 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 our detail view. To do so, we 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!



Customizing UIBarButtonItem

Open up the Images directory and look at **button_textured_24.png** and **button_textured_30.png**. We want to use these to customize the look and feel of the buttons that appear in the UINavigationBar.

Notice that we'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, we don't want the 5 leftmost pixels to stretch, nor the 5 rightmost pixels, so we'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! We'll use the appearance proxy to customize all the UIBarButtonItem at once, like we 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. We create the stretchable images for the buttons and set them as the background for both display in both portrait & landscape orientation. We then format the text to match the typewriter-style font you saw at the outset of the chapter.

The "back" bar button item needs special customization, because it should look different - like it's pointing backwards. Take a look at the images we're going to use to



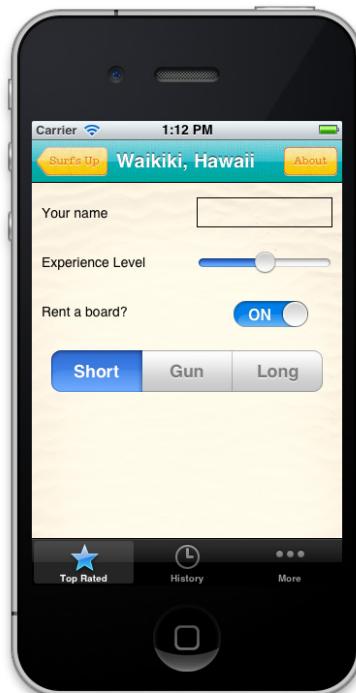
see what I mean: **Images\button_back_textured_24.png** and **Images\button_back_textured_30.png**.

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 we 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 we use here.

Compile and run, and you should now see some cool customized **UIBarButtonItem**s in your **UINavigationBar**!



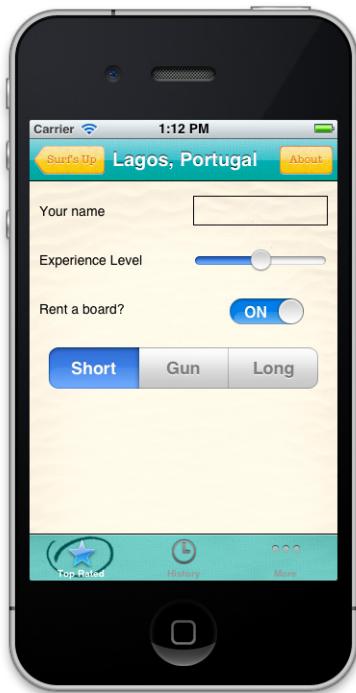
Customizing UITabBar

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 we'll use for these.

Although our mockups only depict one UITabBar, these will in all likelihood have the same appearance if others appear, so we'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

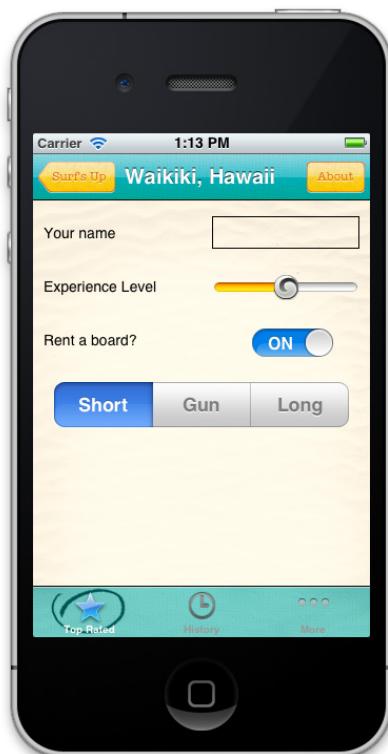
Open up **Images\slider_minimum.png**, **Images\slider_maximum.png**, and **Images\thumb.png** to see the images that we're going to use to customize the **UISlider**. iOS 5 makes it ridiculously easy to customize the **UISlider** by just setting the "maximumTrackImage", "minimumTrackImage", and "thumbImage" 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 we'll customize our segmented control. This component is a little bit more complicated, as we 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 we'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"];

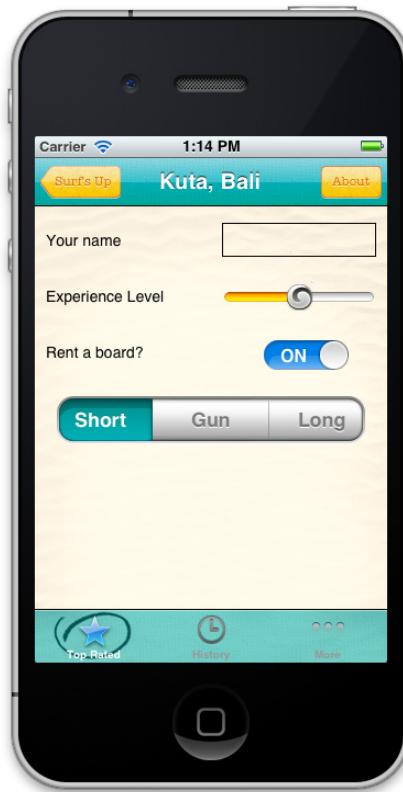
[[UISegmentedControl appearance]
    setBackgroundImage:segmentUnselected
    forState:UIControlStateNormal
    barMetrics:UIBarMetricsDefault];
[[UISegmentedControl appearance]
    setBackgroundImage:segmentSelected
    forState:UIControlStateSelected
    barMetrics:UIBarMetricsDefault];

[[UISegmentedControl appearance]
    setDividerImage:segmentUnselectedUnselected
    forLeftSegmentState:UIControlStateNormal
    rightSegmentState:UIControlStateNormal
    barMetrics:UIBarMetricsDefault];
[[UISegmentedControl appearance]
    setDividerImage:segmentSelectedUnselected
    forLeftSegmentState:UIControlStateSelected
    rightSegmentState:UIControlStateNormal
    barMetrics:UIBarMetricsDefault];
[[UISegmentedControl appearance]
    setDividerImage:segUnselectedSelected
    forLeftSegmentState:UIControlStateNormal
    rightSegmentState:UIControlStateSelected]
```



```
barMetrics:UIBarMetricsDefault];
```

Compile and run, and now our UISegmentedControl has a completely different look!



Customizing UISwitch

At the time of writing this chapter, there is no easy way to customize the artwork on a **UISwitch**. However (like many other controls) it is extremely easy to change its color via the **tintColor** property.

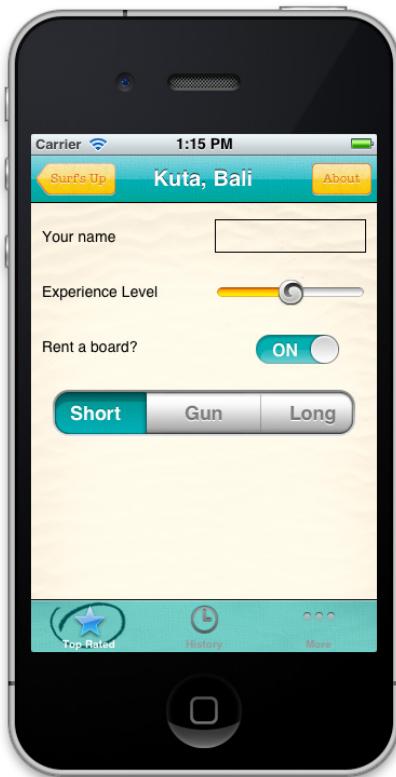
To ensure that we touch on a few different approaches to customization, we'll customize the `onTintColor` parameter in code. You'll note that in **DetailViewController.h**, we already have an `IBOutlet rentSwitch` wired to our switch in **DetailView.xib**.

So, add the following code to set the tint color in the **viewDidLoad** method of **DetailViewController.m**:

```
[rentSwitch setOnTintColor:[UIColor colorWithRed:0 green:175.0/255.0  
blue:176.0/255.0 alpha:1.0]];
```

Compile and run, and check out your newly colored switch!





Things are looking pretty good, but we still have a couple of items outstanding. We need to update the labels and set the background of our custom UITextField.

Customizing UILabel

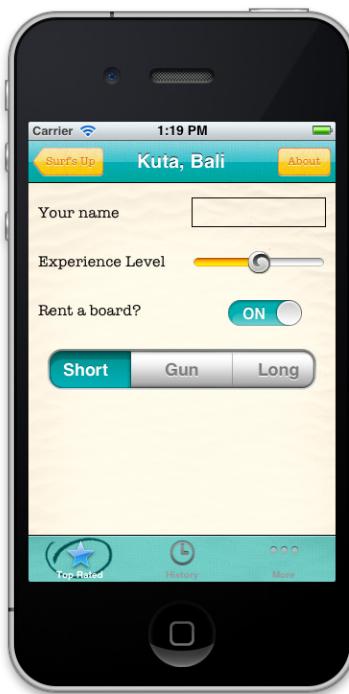
The labels are one part of the detail view we won't customize via the appearance proxy, so open **DetailView.xib** so that we can edit them in Interface Builder.

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 two remaining labels: "Experience Level" and "Rent a board?". Compile and run, and now your labels have a neat typewriter feel!





Customizing UITextField

Our UITextField has already been set to use `UITextBorderStyleLine`. Since we're still in Interface Builder, let's set the font to American Typewriter, Size 12, Regular.

Now if you look at the Identity Inspector for the text field, you'll see that the Custom Class has been set to `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 we have a type called exactly that.

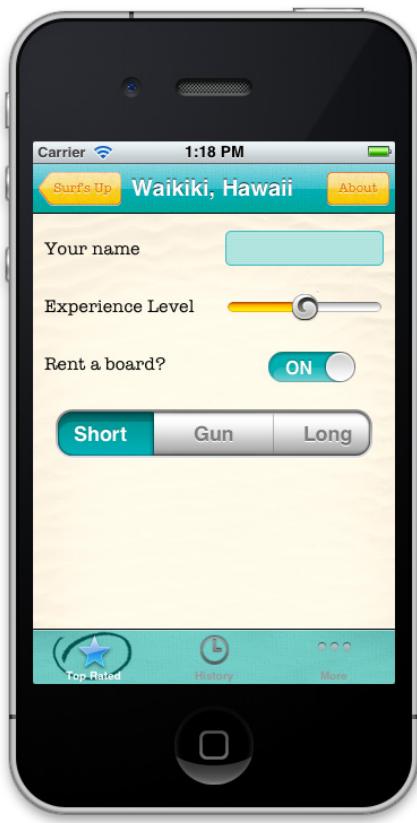
Right now the `drawRect:` method of our `UITextField` delegates to the superclass implementation. But in order to paint the teal background, we are going to override `drawRect:` as another customization technique. Replace the call to `super` with the following code:

```
- (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 we create yet another stretchable image with appropriate insets, and draw



it in the rectangle defined by the bounds of this view (i.e., our UITextField). Let's Build & Run:



Congratulations – the detail view is complete!

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 we'll cover how to customize the table view, port the app to the iPad, and much more!

Intermediate UIKit Customization

by Steve Baranski

In the previous chapter, we took a "plain vanilla" app with the default UIKit controls, and customized the look-and-feel to add some style and flair. We customized the background image, UINavigationBar, UIBarButtonItem, UITabBar, UISlider, UISegmentedControl, UISwitch, UILabel, and UITextField.



In this chapter, we're going to continue where we left off and customize 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 4 and iOS 5 that you may not be familiar with.

Later on in the chapter, we're also going to port this app to the iPad, and customize how to customize a popover controller.



Without further ado, let's surf the iOS customization wave and dive into finishing this app!

Customizing the UINavigationBar Title

Start by opening the Surf's Up project from where we left it off in the previous chapter.

In the previous chapter, we already customized parts of the navigation bar via the appearance proxy, but we still need to add our title graphic. Let's do that first, then we'll proceed to customize the UITableView.

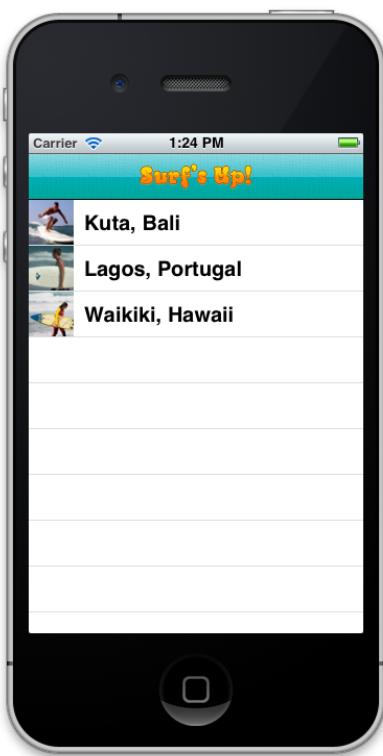
Our title graphic can actually be placed in a UIImageView and set as the `titleView` of our primary view controller's navigation item. This sounds more complicated than it actually is, so let's dive right into code to see what it looks like.

Open up **SurfsUpAppDelegate_iPhone.m**, and find the **application:didFinishLaunchingWithOptions:** method. You'll see that it includes code to instantiate SurfsUpViewController and designate it as our root view controller. We give it a title, which is currently displayed in the center navigation bar, and in the back button in our detail view. Right after the line of code that sets the title, add the following line of code:

```
[[vc navigationItem] setTitleView:  
 [[UIImageView alloc] initWithImage:  
 [UIImage imageNamed:@"title.png"]]];
```

Repeat this for **SurfsUpAppDelegate_iPad.m** (but use `masterVC` instead of `vc`). Compile and run, and check out how it looks:

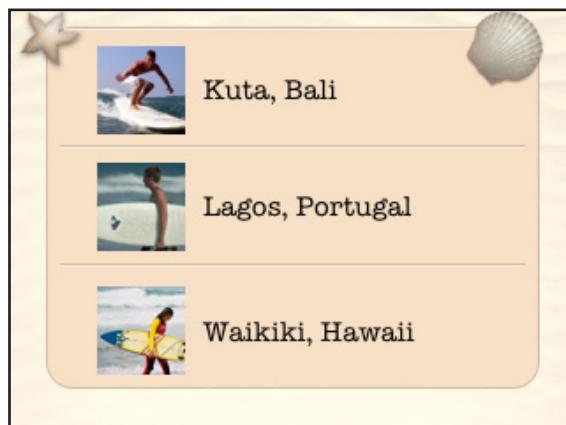




Excellent – I like where this is headed.

Customizing UITableView: Overview

To finish the customization of our primary view, we must now turn our attention to `SurfsUpViewController`, a subclass of `UITableViewController`. Even though we're going to use the "plain" style for the `UITableView`, the artwork we'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 our 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

We can take advantage of this shared scenario; we'll create one subclass of `UITableViewCell` in code and then associate four distinct NIBs in Interface Builder with that class. Finally, we'll employ automatic cell loading – a new feature in iOS 5 – to streamline the use of these custom cells in our table view controller.

Note: We are not be using Storyboards in this app - just normal `UIViewController`s. We thought it was better to do things this way so you see how things work under the hood. However, the material you learn here will still apply if you want to use Storyboards in your app - it's just that Storyboards will make some of the work even easier!

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** – we're going to add two properties that each of our NIBs will rely upon. Specify the following properties in our header:

```
@property (nonatomic, retain) IBOutlet UIImageView *tripPhoto;  
@property (nonatomic, retain) IBOutlet UILabel *tripName;
```

And synthesize these in **CustomCell.m**:



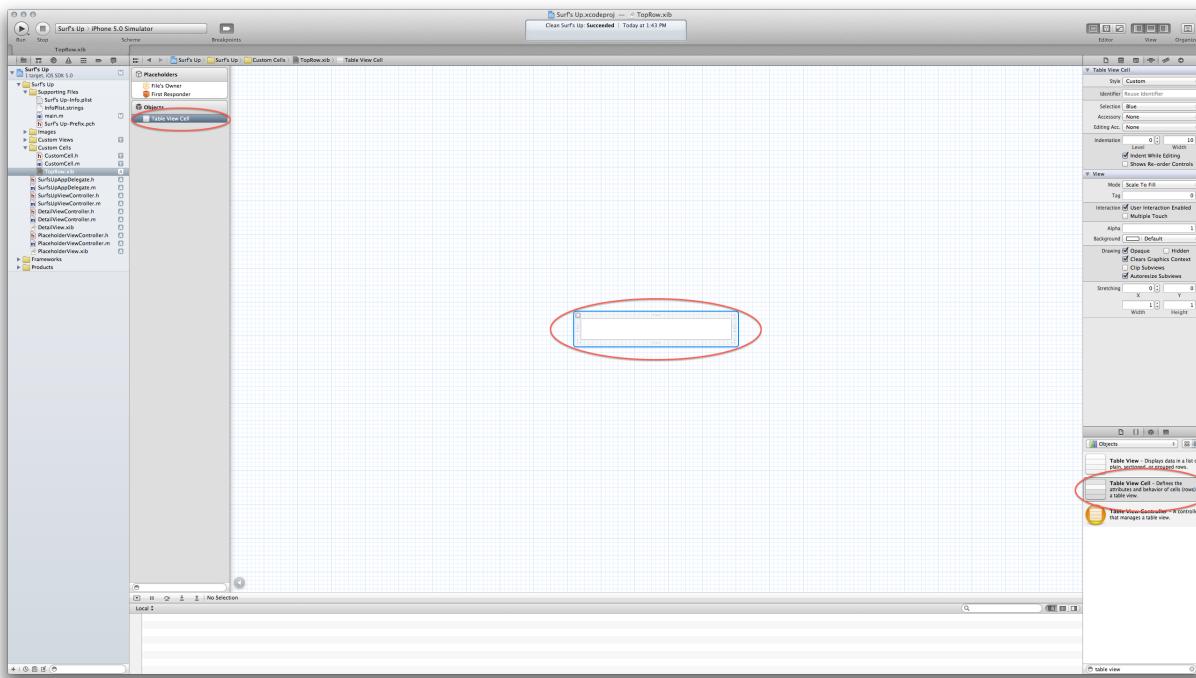
```
@synthesize tripPhoto, tripName;
```

These outlets will allow us to “wire in” the image view and label for the cell that we’ll create in Interface Builder to this subclass of UITableViewCell. This will come in handy later when we want to refer to the image view and label in code!

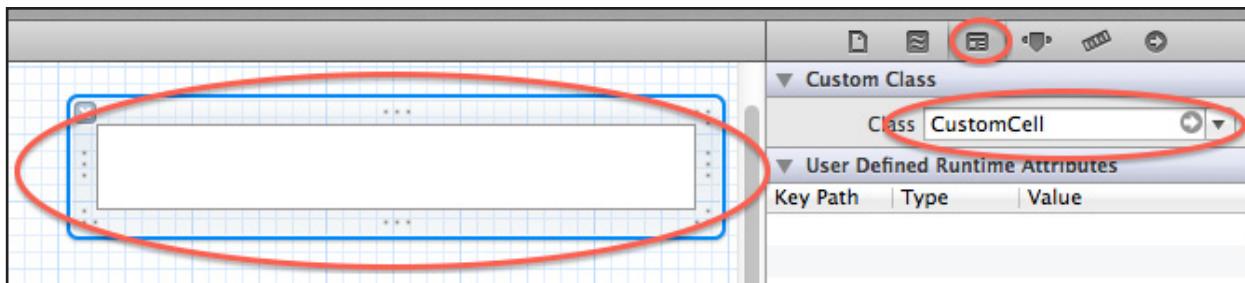
Creating a UITableViewCellStyle in Interface Builder

Next, we're going to create the UITableViewCell for the topmost row in Interface Builder. Create a new file with the **iOS\User Interface\Empty template** with the device family set to iPhone, and name it **TopRow.xib**. Open up **TopRow.xib**, and you'll see the empty XIB appear in the Interface Builder in the middle pane.

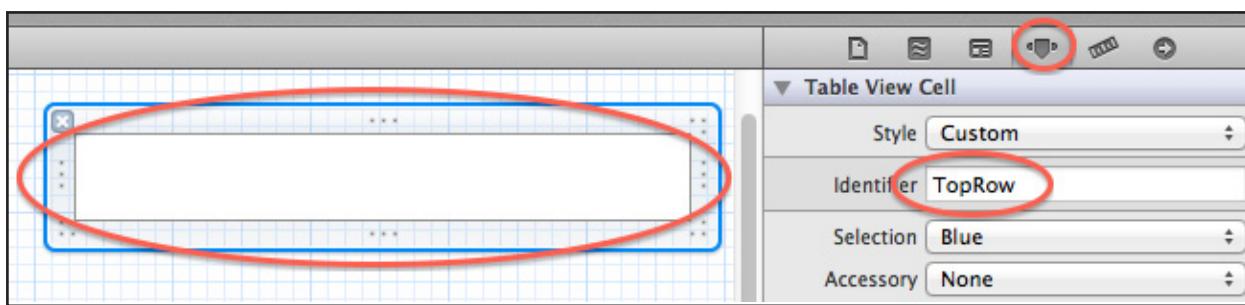
Now, open up the Utility Pane (i.e., the rightmost tab in the View toolbar) and drag a Table View Cell into the middle of the screen:



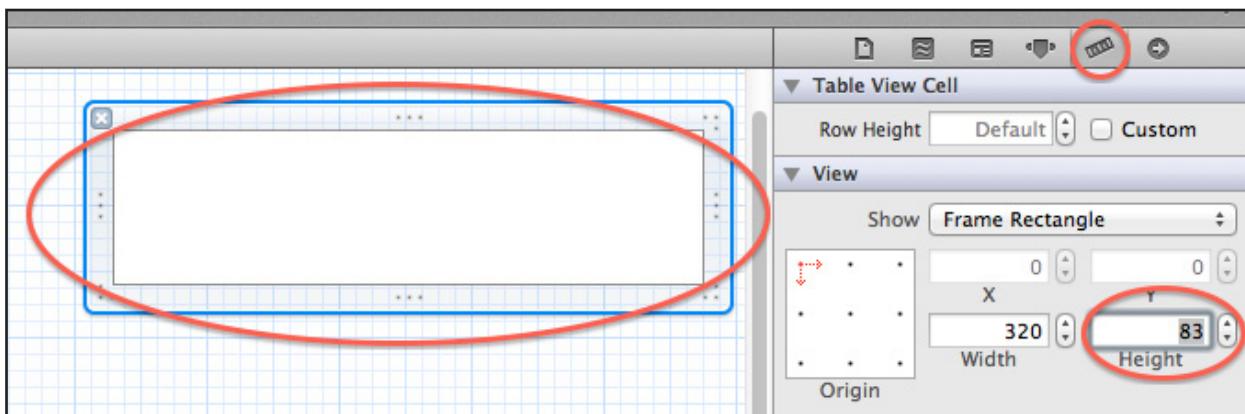
Select the **TableViewCell** in the left pane, then select the third option on the right – the Identity Inspector. In the Custom Class input field, enter the name of class we created earlier, **CustomCell**.



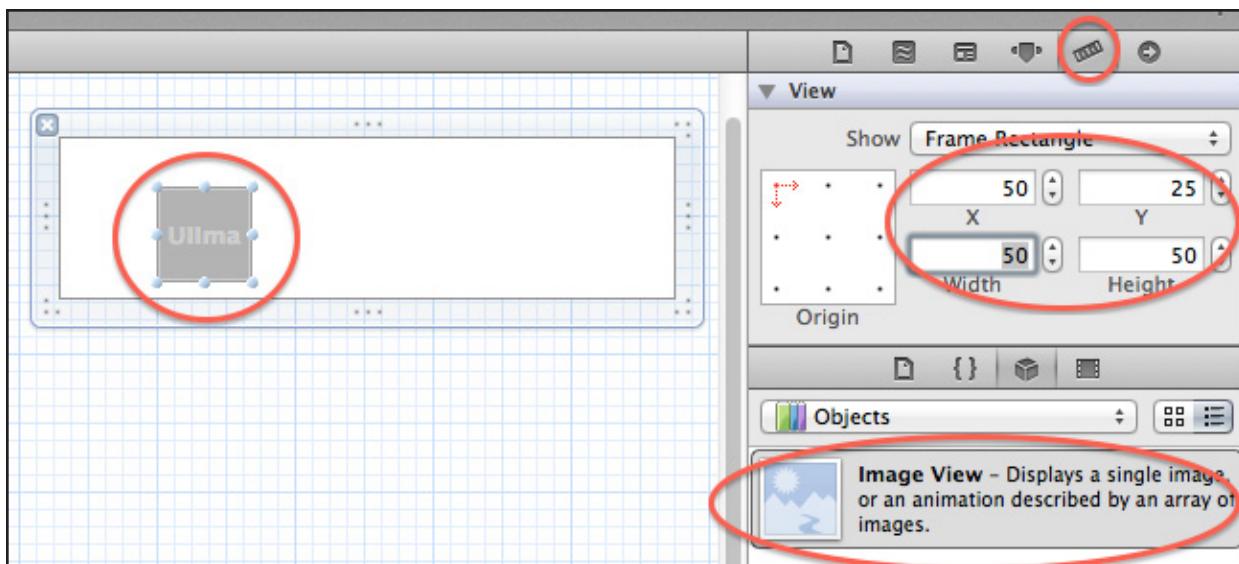
Now select the Attributes Inspector, then enter "TopRow" in the Identifier input field. This is the identifier our table view will use to manage cells of this type in the reuse queue.



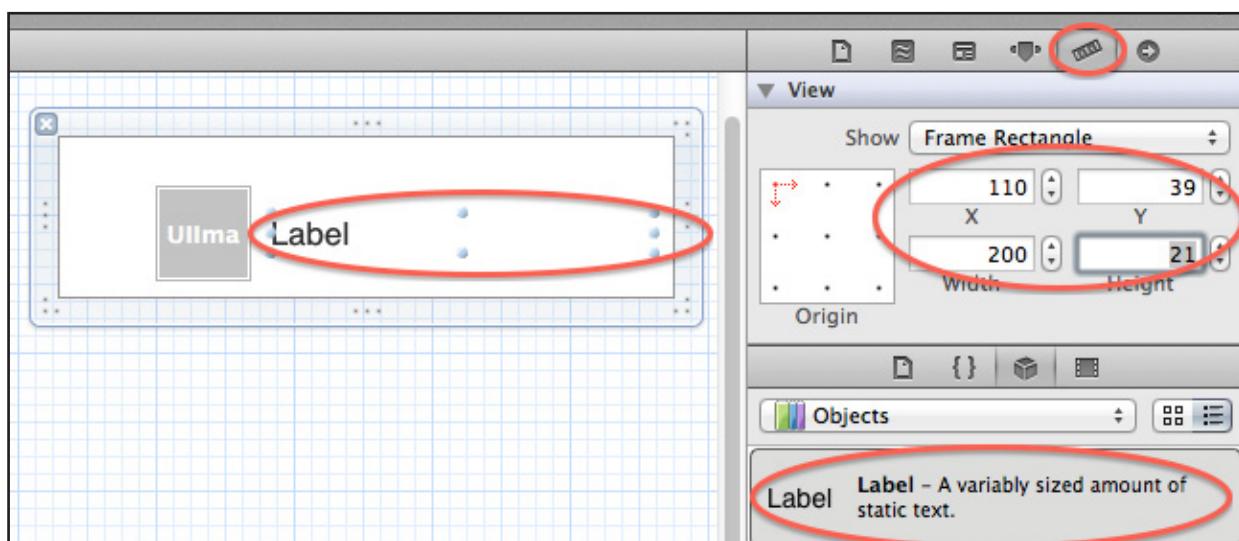
Now select the Size Inspector and enter 83 as the height in pixels of our cell. This corresponds to the height of the custom images used to create our top row.



Next, drag an Image View from the library into the cell. We'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 that our top row has a starfish and a seashell that require a vertical offset.

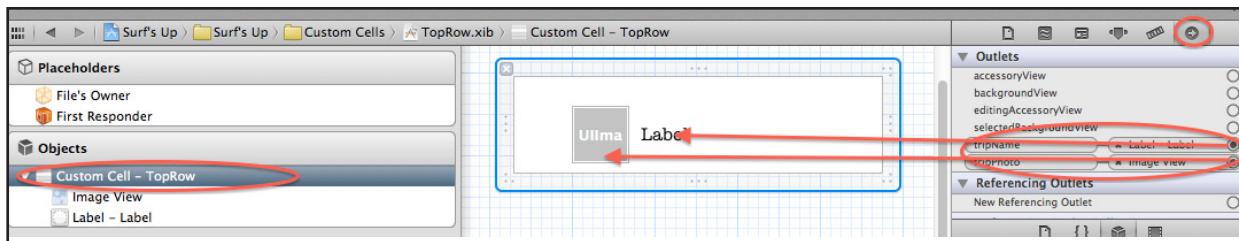


Next add our UILabel in a similar fashion - drag it from the Library into the cell, and set its position/size to X = 110, Y = 39, W = 200, H = 21.



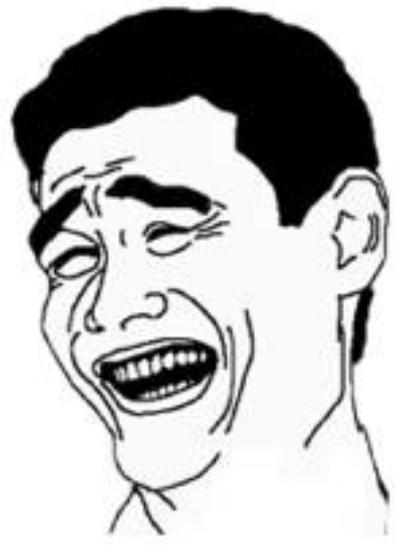
With the UILabel still selected, select the Attributes Inspector. Set the Font to Custom, Family to American Typewriter, Style to Regular, and Size to 16.

Finally, we need to connect the IBOutlets defined in our custom cell class to the corresponding elements in our cell. To do so, select the custom cell in the left pane and the Connections Inspector in the right. Click on **tripName** and drag it to the label to connect it, then click on **tripPhoto** and drag it to the image to connect it too.



Other NIBs

After completing these steps for the top table, most of the work for the remaining cells amounts to “rinse & repeat” from the steps used to create the cell for the top row. This can be good practice, but if you’re like this guy:



Then just download the XIBs I already made from the resources for this chapter, and drag them into your project :] If you do want to make them yourself, the following table summarizes each of the unique parameters from our three remaining NIBs:

Cell	Name/Reuse ID	Height	Image Frame (x, y, w, h)	Label Frame (x, y, w, h)
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

Note: Don't forget to wire up the outlets for each NIB!

Using the Custom UITableViewCells in the UITableViewController

So we've created four NIBs and one custom class. Now it's time to integrate it with our 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 (we added these in Interface Builder earlier)
NSString * const REUSE_ID_TOP = @"TopRow";
NSString * const REUSE_ID_MIDDLE = @"MiddleRow";
NSString * const REUSE_ID_BOTTOM = @"BottomRow";
NSString * const REUSE_ID_SINGLE = @"SingleRow";

// Add new method right before viewDidLoad
- (void)registerNIBs
{
    NSBundle *classBundle = [NSBundle bundleForClass:
        [CustomCell class]];

    UINib *topNib = [UINib nibWithNibName:REUSE_ID_TOP
        bundle:classBundle];
    [[self tableView] registerNib:topNib
        forCellReuseIdentifier:REUSE_ID_TOP];

    UINib *middleNib = [UINib nibWithNibName:REUSE_ID_MIDDLE
        bundle:classBundle];
    [[self tableView] registerNib:middleNib
        forCellReuseIdentifier:REUSE_ID_MIDDLE];

    UINib *bottomNib = [UINib nibWithNibName:REUSE_ID_BOTTOM
        bundle:classBundle];
    [[self tableView] registerNib:bottomNib
        forCellReuseIdentifier:REUSE_ID_BOTTOM];

    UINib *singleNib = [UINib nibWithNibName:REUSE_ID_SINGLE
        bundle:classBundle];
    [[self tableView] registerNib:singleNib
        forCellReuseIdentifier:REUSE_ID_SINGLE];
```



```
}

// Add to the end of viewDidLoad
[self registerNIBs];
```

`UINib` is a class that became available in iOS 4 that helps you load contents from a nib quickly and efficiently. It caches the contents of the NIB in memory so you can create multiple instances of objects from a NIB (like table view cells) quickly, rather than having to reload the NIB each time.

iOS 5 has a new API called `registerNib:forCellReuseIdentifier:` that tells the table view to get a table view cell by looking for a `UITableViewCell` with a particular cell reuse identifier in the specified `UINib`. This saves you from writing the code to solve this common problem yourself!

So here we use these two new APIs to instantiate our nib using the file names we created them with, and then register them with the reuse identifiers we specified. In our case, these values are the same. Now that we've registered our NIBs for use, we'll turn our attention to the logic required to identify which NIB to use:

```
// Add right after viewDidLoad
- (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;
    }

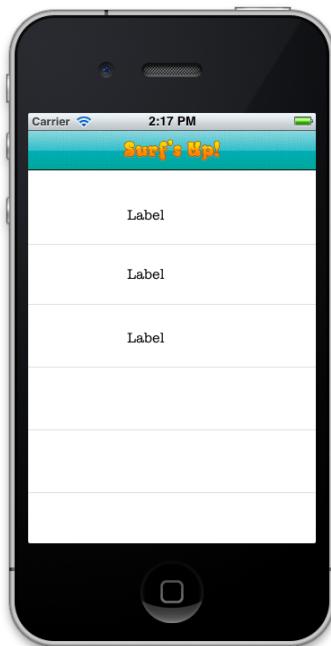
    return REUSE_ID_MIDDLE;
}

- (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, we know whether we should display the top, middle, bottom, or "single row" case. We haven't evaluated our progress in a while, so compile and run to see where we're at:



Wow – it appears that we've actually regressed! We don't have our photos & titles, nor do we 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 we are returning the proper reuse identifiers for each cell.

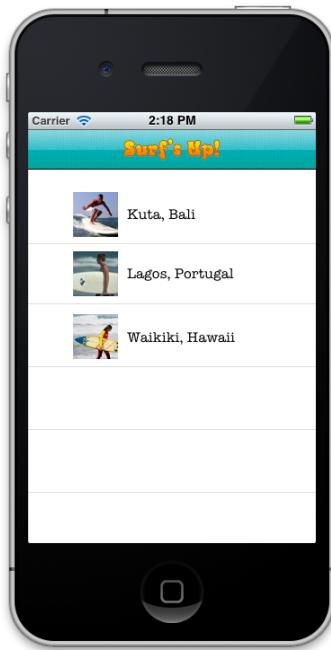
So, what's wrong? Well, when we replaced the old code, we actually removed our cell configuration logic. We lost our specification of the trip photo and the naming of the trip. Moreover, IB doesn't appear to let you set the `backgroundView` & `selectedBackgroundView` properties, so we need to create `UIImageViews` that use our custom cell background images.

Let's work on that now. We'll create a new method: `configureCell:forRowAtIndexPath:` -- and begin by adapting our existing methods for determining the appropriate trip photo & name to use for a given row. Add the following method above **tableView:cellForRowAtIndexPath:**:

```
- (void)configureCell:(CustomCell *)cell  
forRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    [[cell tripPhoto] setImage:  
        [self tripPhotoForRowAtIndexPath:indexPath]];  
    [[cell tripName] setText:  
        [self tripNameForRowAtIndexPath:indexPath]];  
}
```

Now we need to call that method from `tableView:cellForRowAtIndexPath:` - add the following line immediately after we've dequeued a cell for use / reuse:

```
[self configureCell:(CustomCell *)cell forRowAtIndexPath:indexPath];
```



Compile and run, and you'll see that we're back to where we were originally:

We'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, we need to incorporate the background images for each of our respective cells.

To do that, we're going to create two new methods: one will return the background image for our normal state and one will return the background image for our se-



lected state. These methods will accept an index path as input, then instantiate a `UIImage` stretched from the state our designer provided it to properly fit the size of our 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 we've applied our table logic to determine the reuse identifier, we 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 our screen. Once we've loaded our image, we resize our image by specifying UIEdgeInsets – these essentially define the portion of the image that is not stretched. Because the height of our cells matches that of our images, we simple set our top & bottom insets to 0.

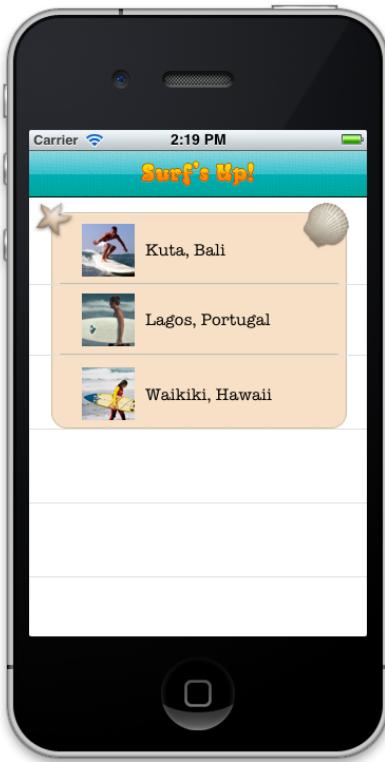
With these methods implemented, add the code to incorporate them at the bottom of **configureCell:forIndexPath:** method like so:

```
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];
```

Let's compile and run again:





Success! Well, sort of. Our cells now include the custom graphics, and subsequently mimic the “grouped” style. Unfortunately, however, we have some residual artifacts from creating the controller in the “plain” style.

Customizing the Table View Background

The last of our customizations address the remnants of creating the table with “plain” look & feel. We’ll tidy this up in **viewDidLoad**. Add the following lines to our method:

```
[[self tableView] setSeparatorStyle:  
    UITableViewCellStyleNone];  
[[self tableView] setBackgroundView:  
    [[UIImageView alloc] initWithImage:  
        [UIImage imageNamed:@"bg_sand.png"]]];
```

The first line removes the separator lines that separate our individual table view cells. The second line introduces our sandy background. As with the navigation bar before, we’re simply creating a UIImageView with our background graphic, and then designating it as the background view for the table. Let’s compile and run one more time:





Order has been restored! We've taken advantage of a number of iOS 5 features so far: we've customized the navigation bar, created four unique table view cells, and leveraged automatic cell loading to render our table view.

Customizing a Popover



In earlier stages of this chapter, we leveraged the new features of iOS 5 to create an iPhone app with a custom look & feel. You may recall that our 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 our button to serve as a "road to nowhere".

So in this part of the chapter, we'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.

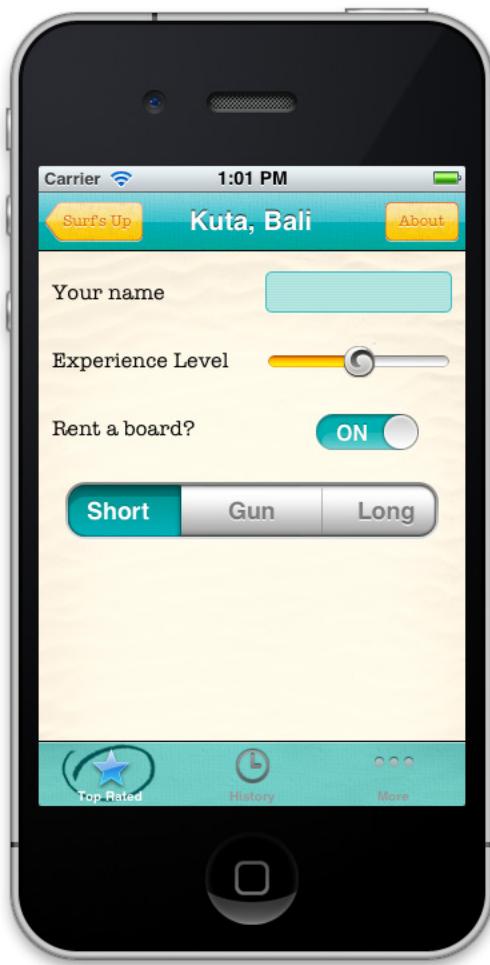
Unfortunately, however, popovers are still limited to iPads. If you attempt to present a popover on the iPhone, it may appear to "work" at compile-time, but at run time you will be presented with the following message:

```
-[UIPopoverController initWithContentViewController:] called when not  
running under UIUserInterfaceIdiomPad.'
```

So to begin work customizing our popover, we have to transition Surf's Up to a Universal Binary. Because this is a tutorial on UIKit Customization, we've done that work already.

In the resources for this chapter, you'll find a folder named **Surf's Up Universal Starter**. Open up the Xcode project, and Build & Run the app in the iPhone Simulator. You'll see the exact same project we had before:





The code & supporting file structure, however, is quite different.



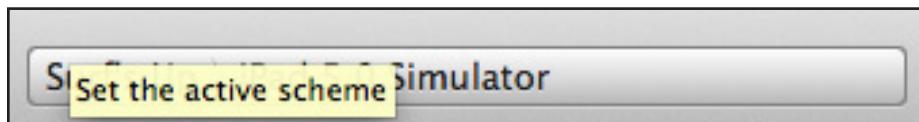
You'll notice that we have some additional groups, and in each of those groups, there are now files with "_iPad" & "_iPhone" suffixes to reflect subclasses and NIBs specific to the target device. There are also new files for our newly introduced popover:

- **AboutViewController.h & AboutViewController.m** – the view controller used to present our popover

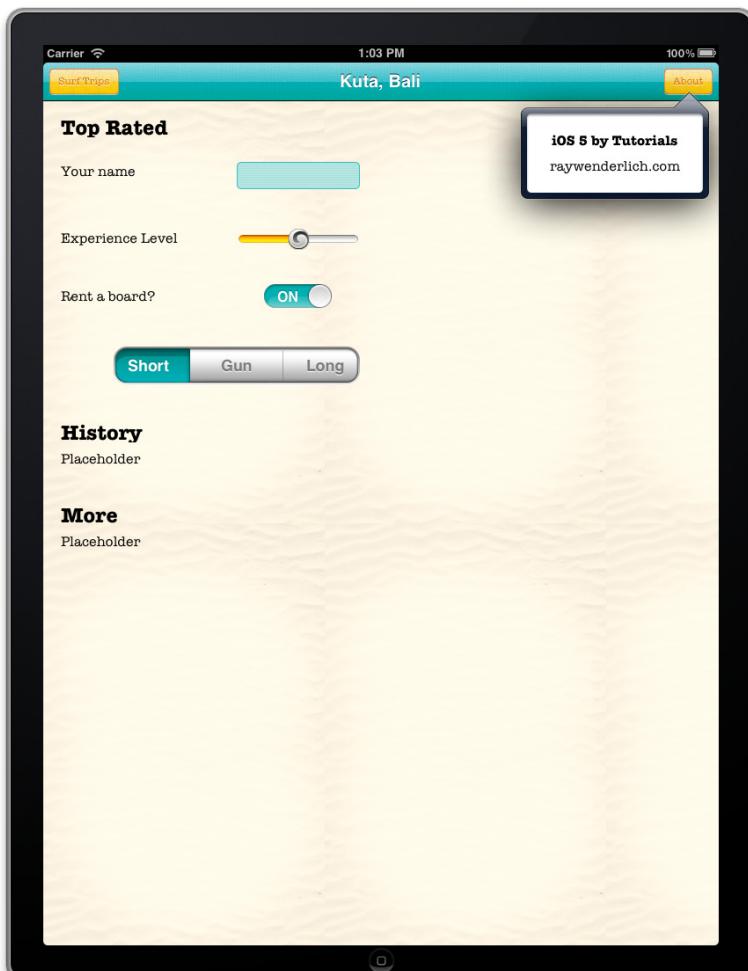


- **AboutView.xib** – the view our popover will present
- A resizable image named **popover_stretchable.png**

Let's Build & Run the app in the iPad Simulator. As you know, we'll need to select the iPad Simulator from the Scheme (see below), and then press Cmd-R as before.



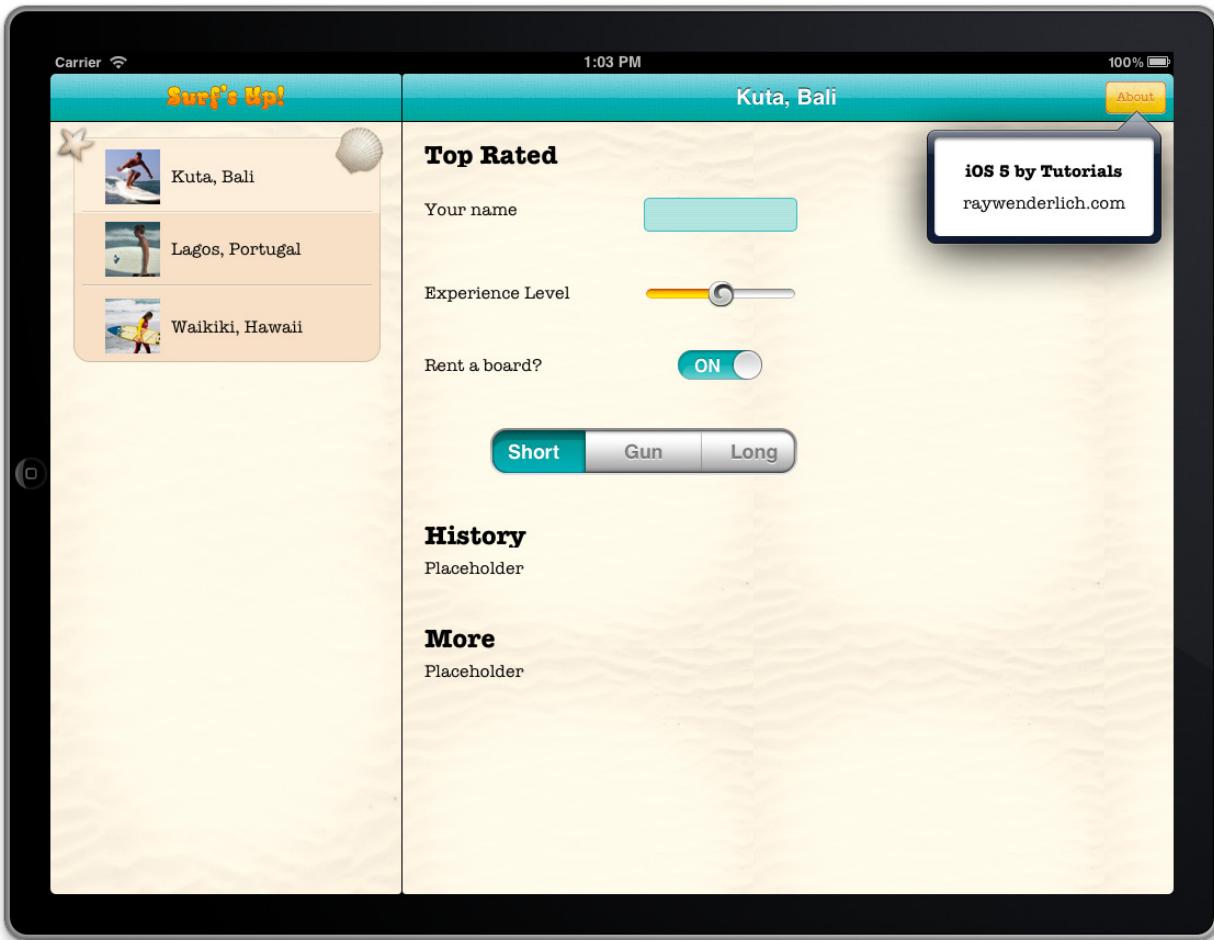
You should see something like the following:



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 we customized earlier in the tutorial – isn't the appearance proxy wonderful?

Now let's take a look at landscape:





Fortunately, our 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.

Note: This tutorial borrows from the UISplitViewController template in iOS 5, which uses a navigation bar in the detail view. In previous versions of iOS, the detail view for that template employed a UIToolbar.

Unfortunately, however, we have the same black border around our popover. This throws off the surfing vibe we worked so hard to create previously. So let’s tidy that up.

Customizing the Popover

There are two facets of customizing our popovers. First, we must create an object to render that view, a subclass of type `UIPopoverBackgroundView` that's new to iOS 5. Second, we must set a value for the `popoverBackgroundViewClass`, a new property of `UIPopoverController`.

First, we'll create a subclass of `UIPopoverBackgroundView` to serve as the background for our 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 we 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:

1. "The background contents of your view should be based on stretchable images."
2. "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."
3. You must override two methods – **arrowOffset** & **arrowHeight** – to describe the geometry of the popover arrow. The fact that these are static methods implies that these values should not change.
4. You must describe the geometry of the content view in relation to its container, the popover. This is also a static method, **contentViewInsets**.
5. 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 our "anchoring" About button, these will be fixed for our purposes.

The backing image for our popover, **popover_stretchable.png**, looks like this.





Once we're finished, we'll have a nice teal border with a lighter turquoise background. To accommodate that, we must change the opaque background of our content view (initialized to white in **AboutView.xib**), we must add this implementation to **AboutBackgroundView.m**:

```
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        [self setBackgroundColor:[UIColor clearColor]];
    }
    return self;
}
```

Now we 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 our 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

Our 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, we 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 we've created our custom background view, we need to set it to instruct our popover to use that view when it is presented. To do so, first import `AboutBackgroundView.h` at the top of **DetailViewController_iPad.m**:

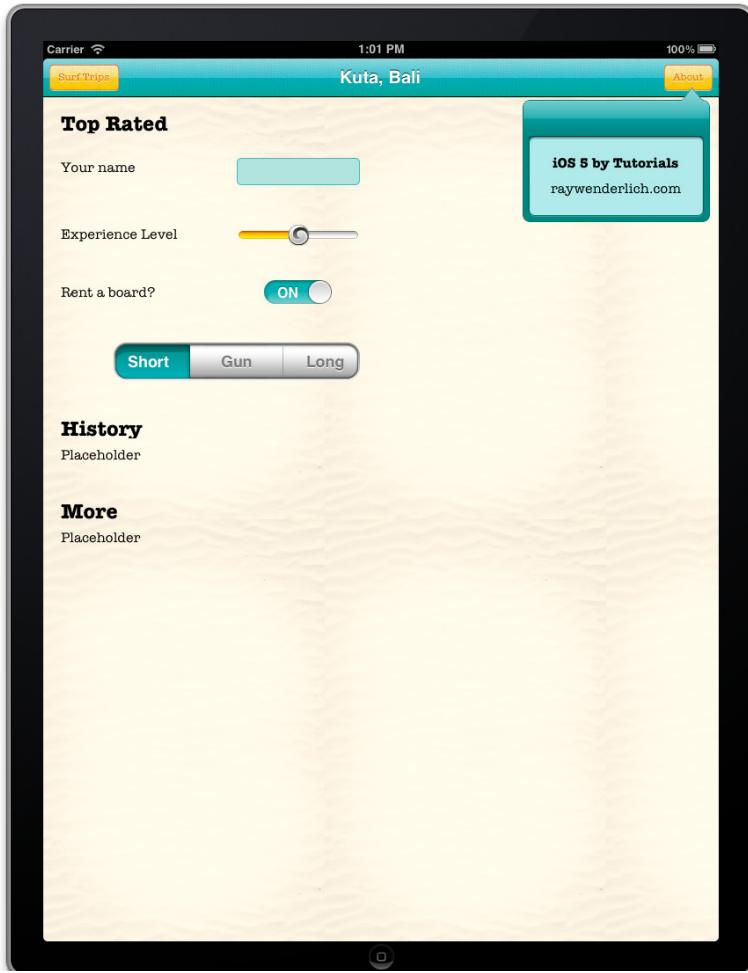
```
#import "AboutBackgroundView.h"
```

Then add the following line to **viewDidLoad**:

```
[[self aboutPopover] setPopoverBackgroundViewClass:  
    [AboutBackgroundView class]];
```

Build & Run – and voila!





Where To Go From Here?

It's been quite a ride! We've customized UI elements on both iPhone & iPad interfaces, and we've used a number of different iOS 5 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

Hopefully you've had as much fun as I've had, and we look forward to seeing your customized UIKit apps in the future!

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 virality of your app, help you identify and retain customers, and can boost the polish and added value of your app.

Until now, adding social features into apps has been a pain. Not only do you have to use a different API for each social network, but users have to constantly log into each one of them for every app they use.

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 on again.

Thankfully for us Apple has taken a huge step forward in this regard by having Twitter natively incorporated in iOS 5! Now all a user needs to do is log into Twitter once and each app can make use of your accounts stored on the device.

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 we'll cover 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 we'll cover the use of the `TWTweetComposeViewController` (i.e. the "tweet sheet") which will enable us to tweet any text, image or link we want from within our 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 we 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!

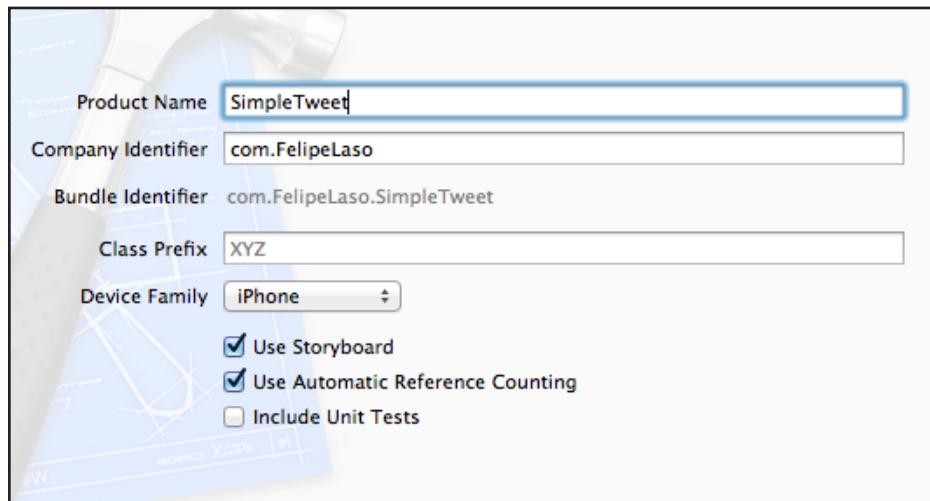
Getting Started

Alright fellow programmers, it's time to get our fingers typing and creating an awesome Twitter enabled app. In this example we'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.

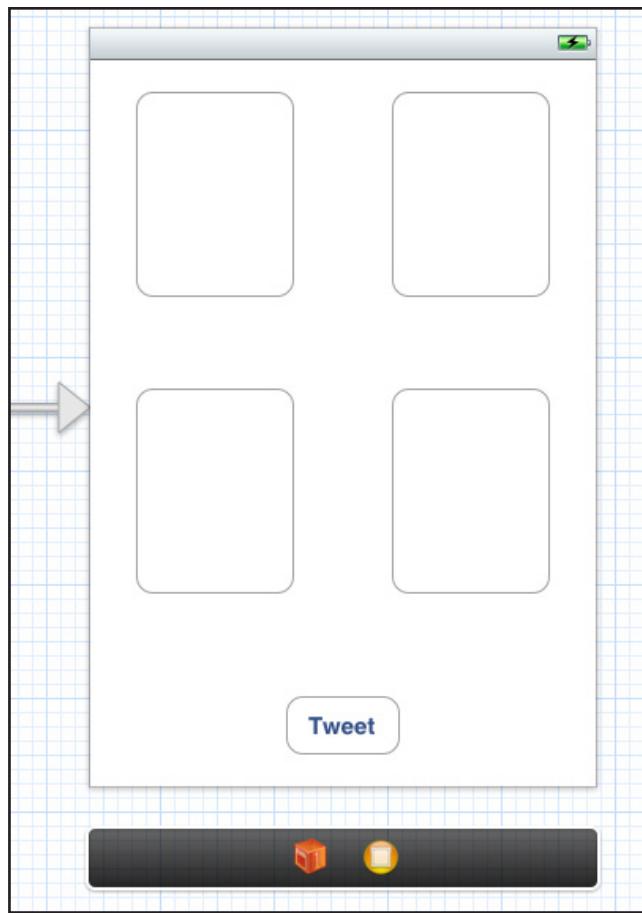
Now that we have our project created let's discuss a bit of what it's going to do. Our app will allow the user to enter the text for their tweet as well as show some buttons for including an optional image and link on their tweet.

We are only going to support Portrait orientation so we need to set that up within our 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:



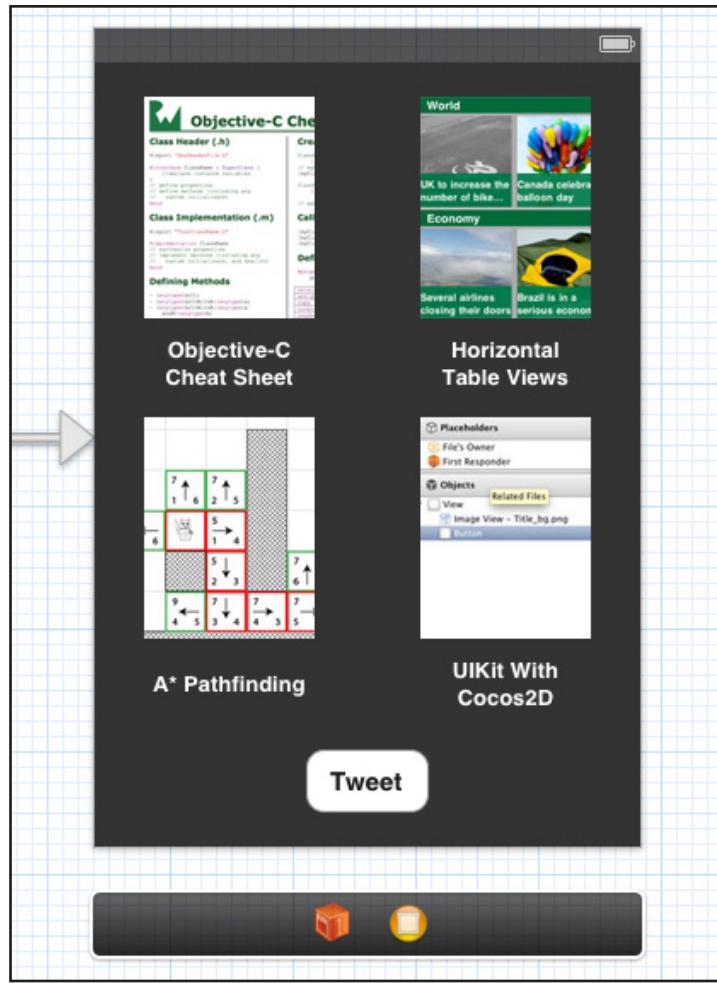
Now open up **MainStoryboard.storyboard** and add 4 UIButtons to the view controller as follows:





We have four large buttons which the user will be able to toggle in order to add an image and link to their tweet.

We're going to make the buttons have an image from 4 different tutorials at raywenderlich.com. Drag the images from the resources 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 buttons type to Custom
- Add 4 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 adding the following line in the SimpleTweet-Info.plist file:

Status bar style	String	Transparent black style (alpha of 0.5)
------------------	--------	--

Compile and run and make sure everything looks ok so far. Now onto the implementation!



Making Some Connections

Next we need to make some connections from our storyboard to the view controller. Open **MainStoryboard.storyboard** file and make sure the Assistant Editor is visible, and displaying ViewController.h.

Control-drag from each of the four labels to below the @interface, and connect them to Outlets named button1Label, button2Label, button3Label, and button4Label.

Similarly, control-drag from each of the four buttons to below the @interface, and connect them to Actions named button1Tapped, button2Tapped, button3Tapped, and button4Tapped.

Finally, control-drag from the "Tweet" button to below the @interface, and connect it to an action named tweetTapped.

When you are done, your header file should like this:

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *button1Label;
@property (weak, nonatomic) IBOutlet UILabel *button2Label;
@property (weak, nonatomic) IBOutlet UILabel *button3Label;
@property (weak, nonatomic) IBOutlet UILabel *button4Label;

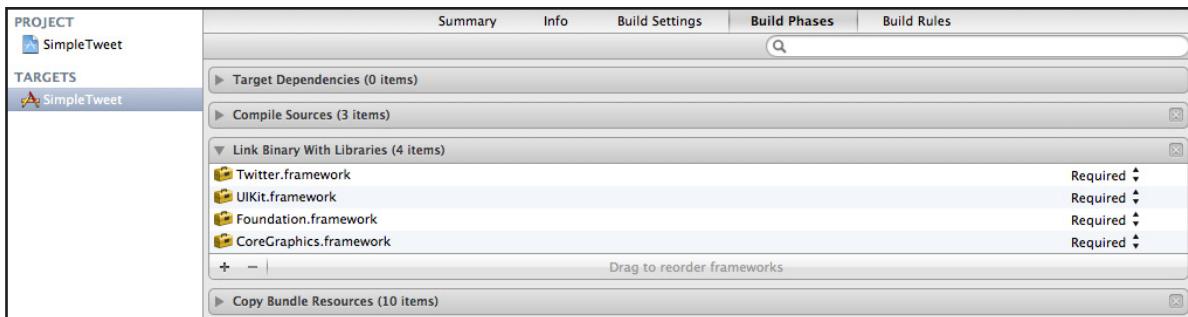
- (IBAction)button1Tapped:(id)sender;
- (IBAction)button2Tapped:(id)sender;
- (IBAction)button3Tapped:(id)sender;
- (IBAction)button4Tapped:(id)sender;
- (IBAction)tweetTapped:(id)sender;

@end
```

Awesome! Now let's implement these methods to send out some tweets!

In order for us to use the `TWTweetComposeViewController` we need to add the Twitter framework to our 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 we need in order for our file to see the Twitter API, let's now add some code so we display the tweet sheet when the user taps the Tweet button. Go to your `tweetTapped` method and 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 we 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 we added to our `tweetTapped` method.

First thing we do is check to see if we can send a tweet, we 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 our application can send a tweet then all we 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 we cannot send a tweet then we 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 which 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 us with a "result" parameter of type `TWTweetComposeViewControllerResult` which 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 we're not going to add this to our project since we don't need it):

```
TWTweetComposeViewController *tweetSheet =  
    [[TWTweetComposeViewController alloc] init];  
  
tweetSheet.completionHandler = ^(TWTweetComposeViewControllerResult  
    result){  
    if (result == TWTweetComposeViewControllerResultCancelled)  
    {  
        // Cancelled the Tweet  
    }  
    else  
    {  
        // Finished composing the tweet  
    }  
};
```

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 category before the implementation inside `ViewController.m`:

```
@interface ViewController()  
@property (strong, nonatomic) NSString *imageString;  
@property (strong, nonatomic) NSString * urlString;  
- (void)clearLabels;  
@end
```

All we 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 our labels' text color back to white.

Then add the `synthesize` statements for these variables:

```
@synthesize imageString = _imageString;
@synthesize urlString = _urlString;
```

And set the properties to nil inside `viewDidUnload`:

```
self.imageString = nil;
self.urlString = nil;
```

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 us.

The way we are going to implement this is as follows: when the user selects a tutorial we store its image name and link within our private properties and we set the label's color to red in order to indicate the current selection.

If the user selects another tutorial then we just set all the labels back to white, store the new image and url and set it's 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 we 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. We 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, we'll leave it at that.

Now go over to the tweetTapped method 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]];
}
```



We just added two if statements to check whether we have an image and url, if we do then we just add them to our tweet sheet by using the `addImage:` and `addURL:` methods respectively.

Once again, this is all we 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 we get:



If you look at the tweet sheet you will notice a paper clip with 2 attachments, one is the link to our website and the other is the image we 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 friends, we 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 all 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!



13 Intermediate Twitter

By Felipe Laso Marsetti

In the last chapter, you saw how easy it was to send tweets by using the `TWComposableViewController`.

This is great, but doesn't do everything you might want to do with Twitter. For example, what if you want to integrate your app with a user's Twitter profile, or perhaps add a Follow button so users can follow your Twitter account from within your app?

Well there's good news - iOS 5 makes it easy to do this with two new built-in APIs:

- **Accounts Framework:** The Accounts framework provides an API for system-wide accounts like Twitter (and more may be added in the future).
- **TWRequest:** `TWRequest` is a handy class that allows us to communicate directly with the Twitter API. You can send any command at all supported by the API!

In this chapter, you'll use both of these APIs to make a very simple Twitter client app. By the time you're done, you'll understand how to use these to perform any request you might want to do supported by the Twitter API!

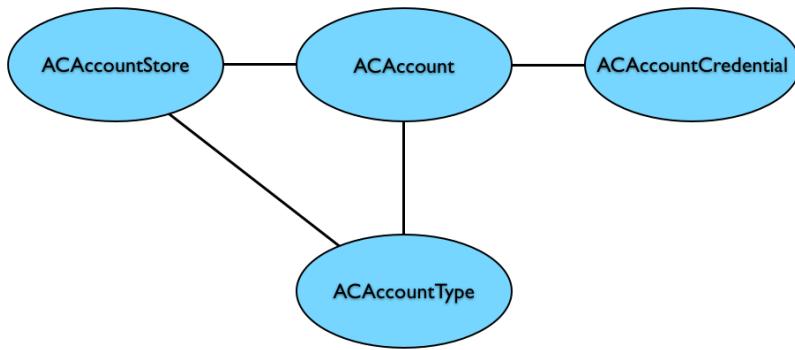
The Accounts Framework

The Accounts framework is made up of the following classes:

- `ACAccountStore`
- `ACAccount`
- `ACAccountCredential`
- `ACAccountType`

This is how they are connected between each other:





Let's discuss the purpose and functionality of each class next. This way, you'll get a deep understanding of the entire Accounts framework and will have an easier time using it later on.

ACAccountStore

The account store is the way your app communicates with the accounts stored on the device. It enables you to retrieve, delete and insert accounts into the accounts database.

There are several things you can do with the account store such as:

- Request access to all of the system's accounts with a particular account type.
- Get a particular account with a given identifier.
- Get an array of accounts of the same type.
- Save an account from your application to the accounts database.

The account store is the first class you will use if you want to access built-in accounts on the device. The typical workflow will consist of the following steps:

- Request access to the accounts of a particular type (in our case Twitter accounts).
- If given access to the requested account type, either retrieve an array of accounts or retrieve a particular account by providing the account's identifier.
- Use the retrieved account (or array of accounts) in our app.

Additionally you can store an account the user logged into from your application to the accounts database. This comes in handy for apps developed prior to iOS 5 where the user has already logged into Twitter - you can then store that account



into the accounts database and let the user enjoy the simplicity of that account being available to other apps on the device.

One very important thing to keep in mind is that accounts retrieved from a particular account store are linked to that account store, so do not try to use accounts from one account store with a different account store.

Apple recommends that you treat an account store as a long lived instance. So rather than creating multiple account stores, you could instantiate an account store and keep it in your app delegate as long as your app is in use.

Finally, since your user can leave your application to do other tasks, he might remove or update an account stored on the device while your app is in the background. For this reason you should subscribe to the `ACAccountStoreDidChangeNotification` notification, that way if an account is removed or modified you'll know about it. If this occurs, you should create a new account store instance and update the accounts in use by your app.

We will not be registering for this notification in this demo app, but again it is something you should consider doing for your own projects.

ACAccountType

The account type class stores information about all accounts of the same type. You don't actually instantiate an account type object, instead it's retrieved from the account store using the `accountTypeWithAccountTypeIdentifier:` method or the `accountType` property of an `ACAccount` object.

As the time of writing this chapter, iOS 5 only includes the `ACAccountTypeIdentifierTwitter` account type, but there may be other account types supported by iOS in the future.

The properties of an `ACAccountType` object are:

- **accessGranted**: a boolean that tells whether your app has access to this particular type of accounts or not.
- **accountTypeDescription**: a description of the account type.
- **identifier**: the unique identifier of the account type.

Pretty straightforward so far right? Let's now move on to the `ACAccountCredential` class.



ACAccountCredential

This class contains all the info needed to authenticate a user. We will not use this class directly in this chapter, and given the simplicity of the accounts framework you'll rarely need to use it as well.

The account credential class is handy when you want to authenticate your users yourself using the OAuth open authentication standard. Instead of providing a user-name and password for their personal Twitter account, the user is authenticated using a key and secret sent directly to the server, and your application receives a token granting access for a set duration.

Authenticating a user yourself with OAuth is beyond the scope of this book, but just know it's there in case you require it or if your app already has an OAuth scheme implemented.

ACAccount

The last class in the Accounts framework is ACAccount. This is the most important class in the Accounts framework, and is the one which actually contains all of the information about a user's account stored in the accounts database.

An ACAccount object stores all of the data corresponding to a user's account, such as:

- **accountDescription:** human readable description of the account.
- **accountType:** returns an ACAccountType object with the type of the account.
- **credential:** return an ACAccountCredential object with the account's credential.
- **identifier:** the account's unique identifier.
- **username:** the account's username.

Twitter Request

Now that we have a better understanding of the Accounts framework, let's discuss the Twitter Request class.

In the Beginning Twitter chapter we covered how to use the TWTweetComposeViewController, or Tweet Sheet as it's sometimes called, but what happens when you



want to do more complex things with the Twitter API like getting a user's followers, profile, direct messages, mentions, etc?

The answer lies in the Twitter Request class.

Apple engineers have created Twitter Request, a Cocoa framework for communicating with the entire Twitter API.

The beauty of Twitter Request is that it's not a direct port of the Twitter calls and services, instead it's a very flexible framework that lets you interact with all of what Twitter has to offer regardless of changes made to it in the future.

When creating Twitter Request Apple engineers realized that, being a web platform, Twitter is always changing and adding/modifying its contents and services. Because of this a direct port of the Twitter API would have been quite useless since we as developers would depend on Apple updating the Twitter framework to support the new functionality.

Instead what we have is a set of native Objective-C Cocoa classes for interacting with the entire Twitter API regardless of what changes it makes in the future.

The easiest way to learn how to use the Twitter Request API is to try it out hands-on! So let's dive into making a simple Twitter client to try it out.

Don't let the "simple" fool you though, there's a lot this app will do! Here's some of the features it will include:

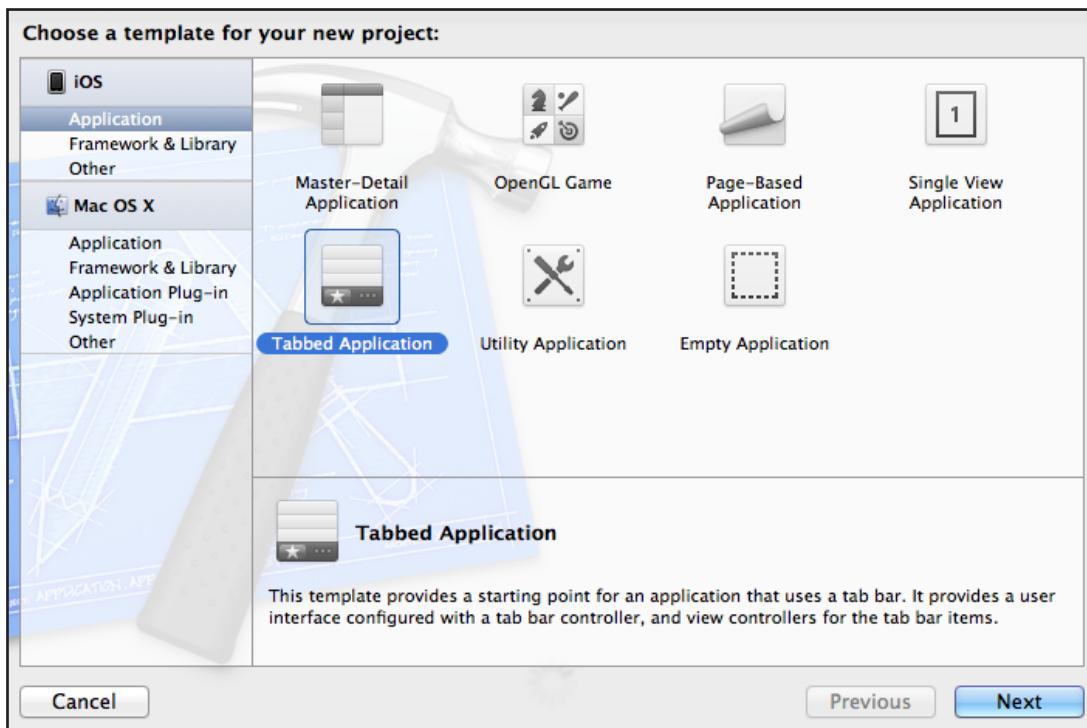
- Sending Multipart Data.
- Letting users follow you from within your app.
- Parsing a Twitter Request from JSON to a Foundation object using the new JSON Serialization class.
- Downloading a user's feed, mentions and profile.
- Learning how to use the Twitter documentation.
- Using a single, persistent ACAccountStore instance to handle your user's accounts.
- Retrieving Twitter accounts from the system.
- Receiving notifications for changes in the accounts database.



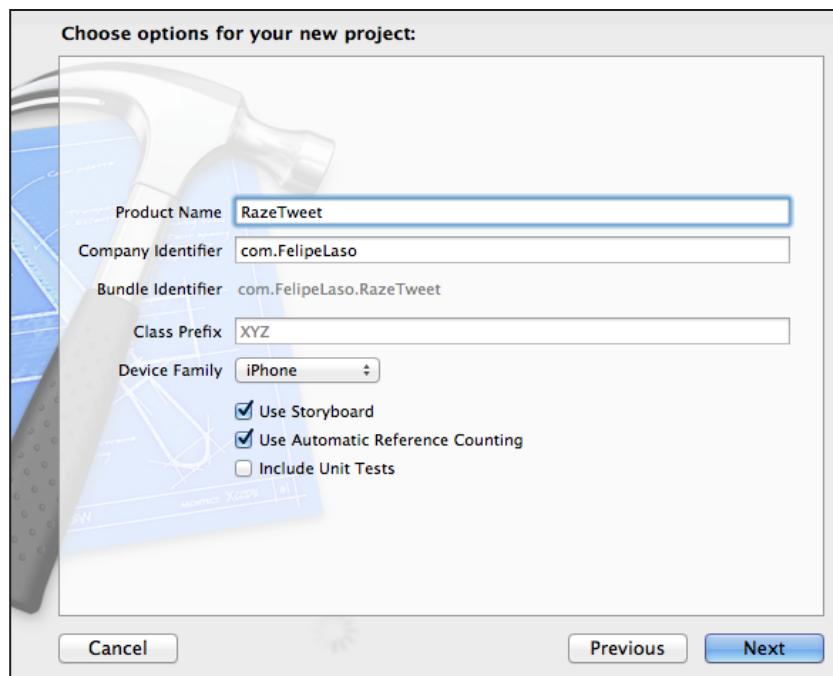
Creating our own Twitter Client

Now that we have a little bit of knowledge of how things work underneath the surface let's get started with our project!

Open up Xcode and create a new Tabbed Application, this will enable us to use a Tab Bar for separating the different sections contained in our app.



Click Next and in the Project Options name your project **RazeTweet**, select **iPhone** for Device Family and make sure that **Use Storyboard** and **Use Automatic Reference Counting** are both selected.



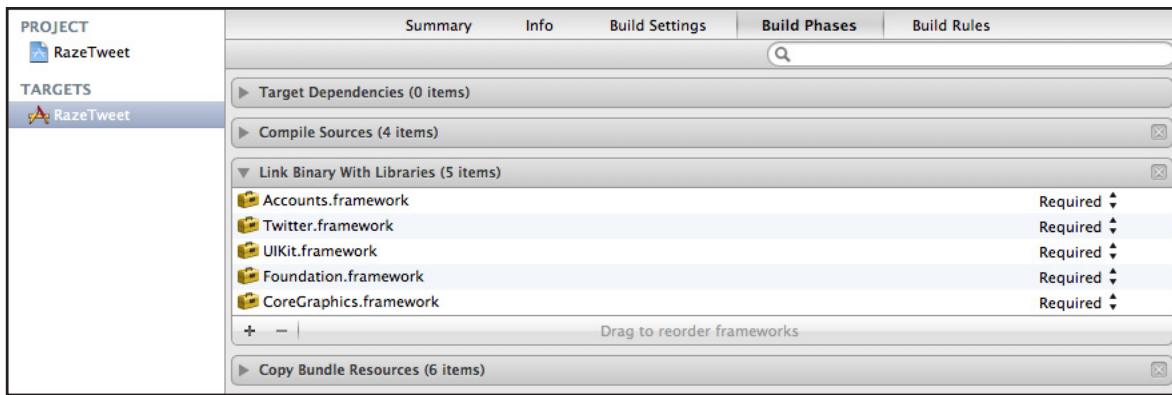
Click Next one more time and select a place to save your project.

Before we get started coding, select the project from the Project Navigator, and change the Supported Device Orientations to the Portrait orientation only:



While you're in the project settings, go to the Build Phases tab, expand the Link Binary With Libraries section and, using the plus icon, add the Twitter and Accounts Frameworks. We need to add these frameworks in order to be able to use both Twitter Request as well as the Accounts related classes we talked about at the beginning of the chapter.

This is what your project settings should look like after adding both frameworks:



That's all we need to do with our project settings at the moment, now let's look at the files that have been created for us before making some changes to the user interface.

In the Project Navigator you will see we have the App Delegate header and implementation files, two View Controllers (displayed on the storyboard's tab bar controller), some images for the tab bar icons, and our storyboard file.

This is a nice start, but we obviously don't want our view controllers to be called "First" and "Second", or be limited to just two view controllers.

So let's create some new view controllers and refactor the existing ones to get our app in line with our needs. Here are the View Controllers we will have in our application:

- Feed (Table View Controller)
- Mentions (Table View Controller)
- Tweet (View Controller)
- Profile (View Controller)
- Follow Us (View Controller)

Open **FirstViewController.h**, right click the class name and select Refactor/Rename, name the new view controller **TweetViewController**. Repeat this process for **SecondViewController** but this time name it **ProfileViewController**.

If you run your app you won't notice any changes, that's because the title for these view controllers hasn't been changed in our Storyboard file.

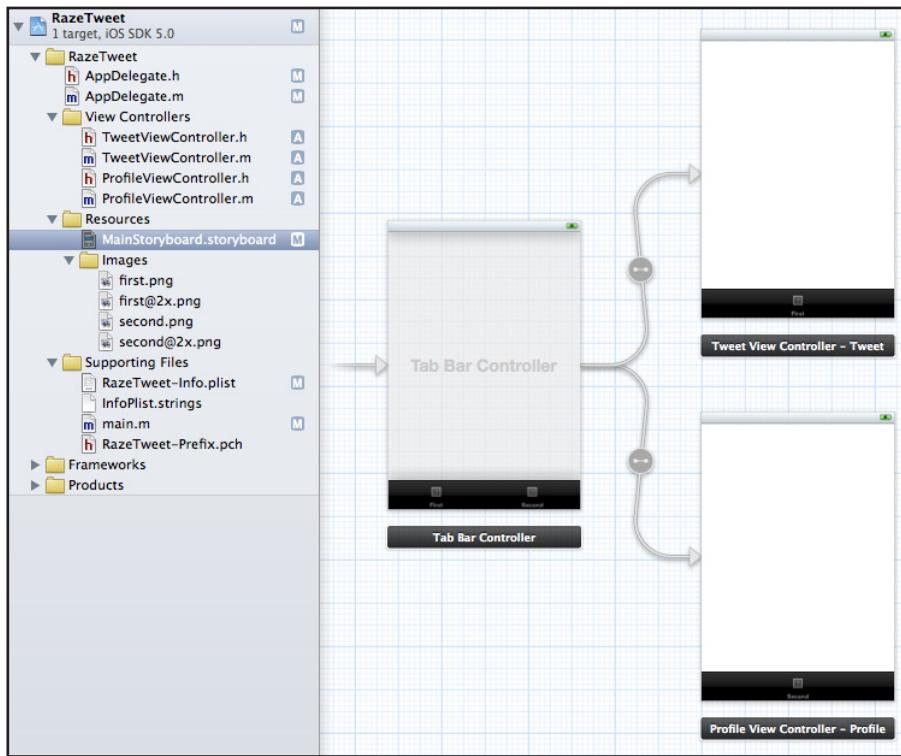
Open **MainStoryboard.storyboard** and you will notice that our view controllers are properly named now. Select the TweetViewController and then select its Tab Bar Item. Inside the Attributes Inspector change the title to Tweet. Don't worry



about the image for now, we'll change that later. Next select the ProfileViewController and change the title to Profile.

You can also delete the UILabel objects inside each view controller, they were provided by the template but we don't need them.

Here's how things are looking inside our storyboard and Project Navigator:



If your project navigator isn't exactly the same as mine do not worry, I've just ordered the files and resources to what I'm accustomed to, feel free to organize these files as you prefer.

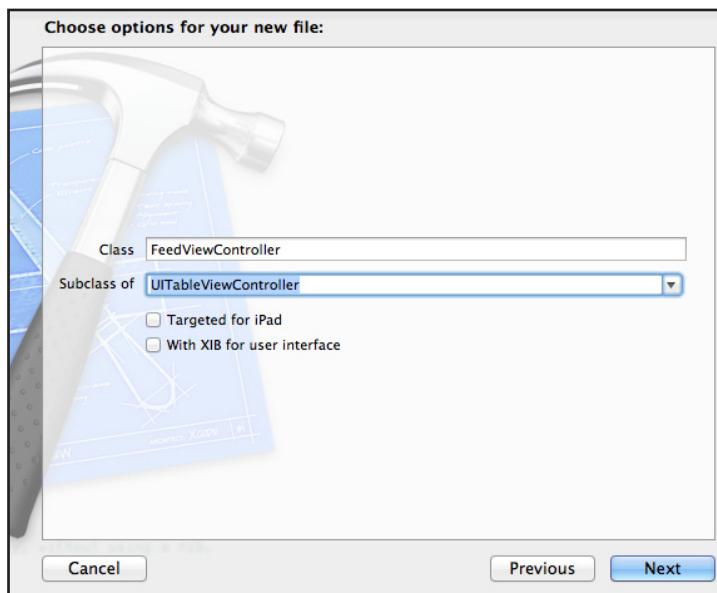
Notice that we no longer have FirstViewController and SecondViewController header and implementation files, when we renamed our view controllers the file names changed for us, how convenient! The storyboard also reflects the changes we made, with the proper title being displayed on the tab bar as well as on each view controller.

Let's create the remaining View Controllers we need for our project and hook everything up in our storyboard file.

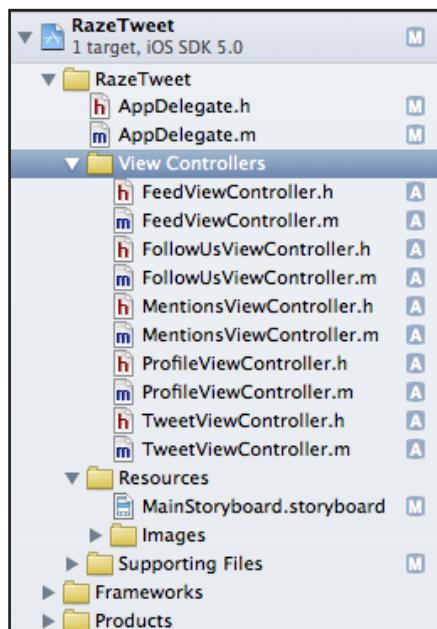
In your Project Navigator right click the RazeTweet folder and select New File, from here we'll use a Cocoa Touch template and create a **UIViewController subclass**. Click Next and name your file **FollowUsViewController**, make it a subclass of **UIViewController**, and make sure that Targeted for iPad and With XIB for user

interface are not selected. Click next one more, select where you want to save your file and click Create.

Now we just need two UITableViewController subclasses for our feed and mentions. Repeat the same process for the **FeedViewController** except this time make sure you select **UITableViewController** as the subclass in the file options screen.



Go ahead and repeat this step one more time for the **MentionsViewController**, make sure it's also a subclass of **UITableViewController**. This is what my project navigator looks like after we have created our necessary view controllers:



With that ready let's make a few more tweaks to our storyboard and we'll be ready to start coding some actual Twitter functionality!

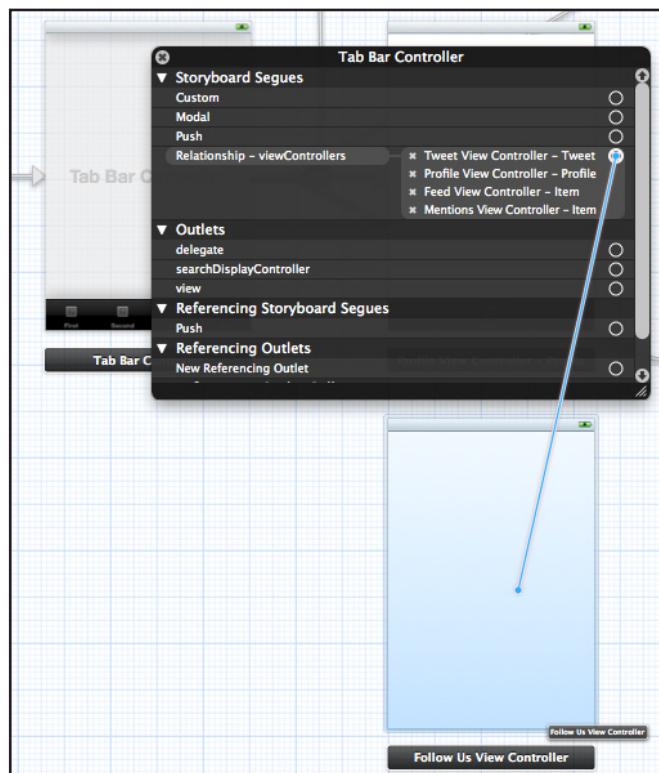
Inside **MainStoryboard.storyboard** you will notice that we still have just two view controllers, just as we left it last time. Go ahead and drag one view controller and two table view controllers from the Object Library onto the canvas.

They still read View Controller and Table View Controller because we haven't specified the class for them or connected them to our tab bar. To do that, select the view controller, open up the Identity Inspector, and set the Class to FollowUsViewController.

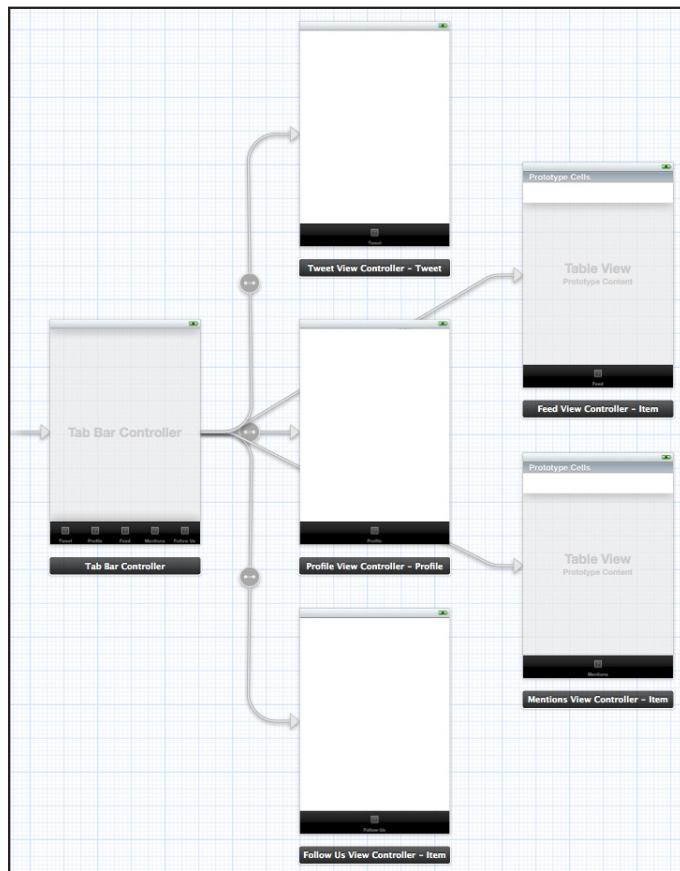
Repeat the same step for both of our table view controllers, using FeedViewController and MentionsViewController as their class inside the Identity Inspector.

Now we need to connect them to our tab bar controller so we can select them from the tab bar along with the other view controllers. Right click on the tab bar controller and, in the overlay window, notice the relationships already established for us. The tab bar controller is related to the Profile and Tweet view controllers, we need to add our three remaining view controllers to the relationship in order for them to show up on the tab bar.

If the connections overlay window has disappeared then right click the tab bar controller once more, click and drag from the Relationship outlet to your view controllers, one at a time:



You should now have the tab bar controller connected to each of our view controllers, like this:



Just as we did earlier, we need to rename the Tab Bar Items for our new view controllers so they are properly titled in our 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

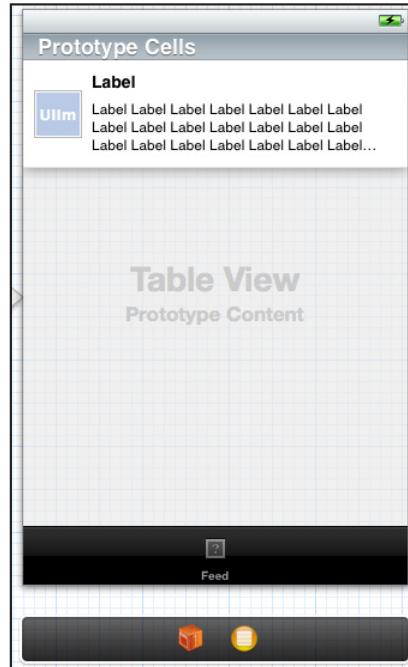
To do this you just need to click and drag on the tab bar buttons and arrange them in the position you like.

We're almost done customizing our storyboard - all we need is a custom cell for the Feed and Mentions view controllers. For the user's feed and mentions we want the table to show cells with the profile image of the person who sent the tweet, their username and the actual tweet contents.

We could use one of the cell styles already available but we wouldn't be able to show everything we 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 we can create these custom cells from our 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 we can place the custom elements we 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:



We not only have to specify the row height within our cell but also within our table view, otherwise our 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 we want to use the same cell for the Mentions View Controller all we 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 our 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 us to be able to set the profile picture in our cell as well as the user name and tweet text, we need to create a subclass of UITableViewCell where we 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 we have our table view cell subclass let's add some outlets for our image and labels. Replace **TweetCell.h** with the following:

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

@interface TweetCell : UITableViewCell

@property (strong, nonatomic) IBOutlet UILabel *tweetLabel;
@property (strong, nonatomic) IBOutlet UIImageView *userImage;
@property (strong, nonatomic) IBOutlet UILabel *usernameLabel;

@end
```

Now let's synthesize our variables by replacing **TweetCell.m** with the following:

```
#import "TweetCell.h"

@implementation TweetCell

@synthesize tweetLabel = _tweetLabel;
@synthesize userImage = _userImage;
@synthesize usernameLabel = _usernameLabel;

@end
```

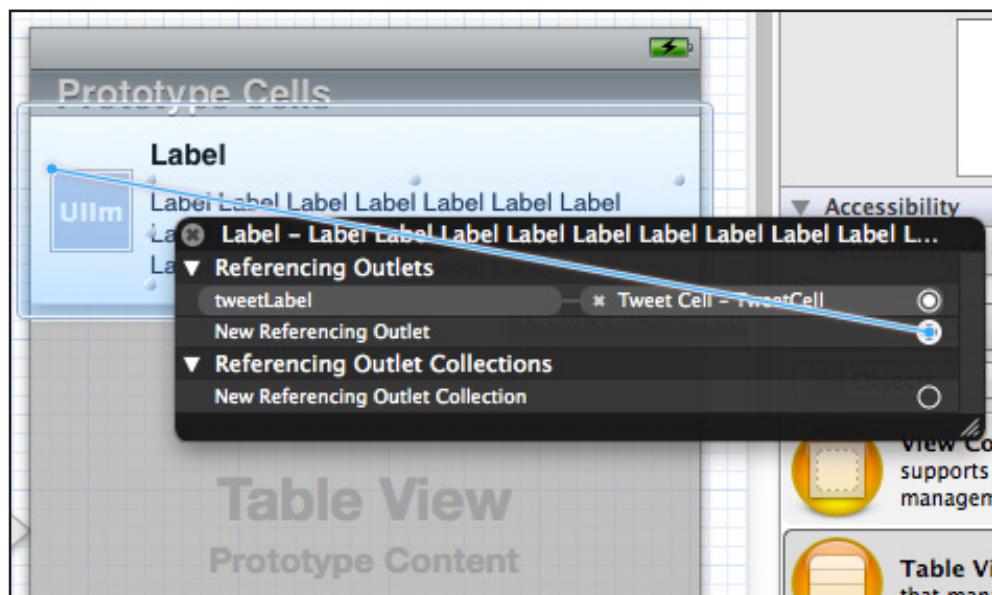


Note: The newer runtimes allow you to skip the instance variable declarations in your header file. Instead you can directly create your properties and synthesize them, and the runtime will automatically create instance variables for you when you compile. You can specify a different instance variable name from the property name by using the equal sign to specify the property on the left and the variable's name on the right (like we did here).

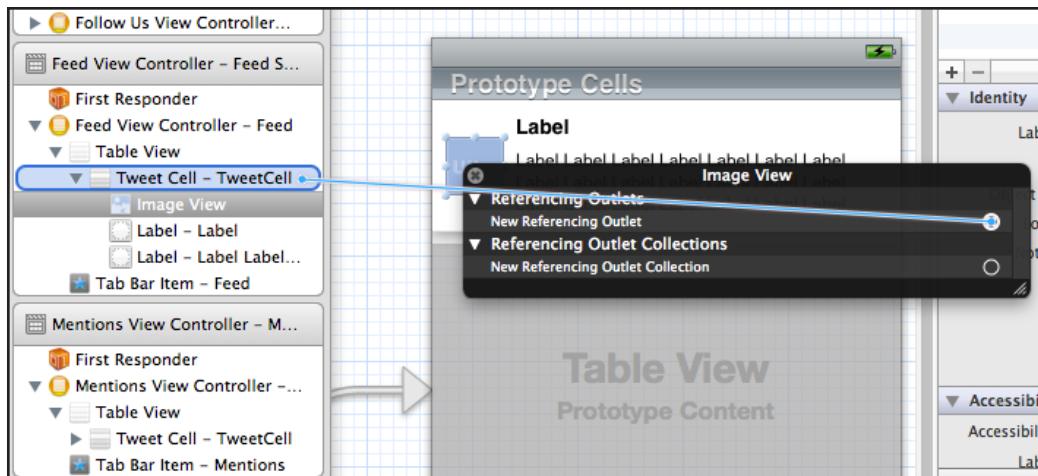
All we are doing here is declaring two `UILabels`, one for our tweet and one for the username, and a `UIImageView` for the tweeting user's profile image.

Switch back to **MainStoryboard.storyboard** one last time so we can assign the new `TweetCell` subclass to the custom cells we created earlier.

Inside the storyboard select the cell we 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:



Connecting our label using the prototype cell



Connecting our label using the Document Outline

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 our corresponding view controllers, each with the correct title, but in the Feed or Mentions View Controllers we can see taller rows but not our custom cell. Don't worry about this, we 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 we need to retrieve the user's Twitter accounts, so let's do so in the next section.

Getting a User's Account

We already covered the use of the Accounts framework at the beginning of the chapter so the code we write here should look familiar or at least make a lot of sense.

Open up **AppDelegate.h** and replace it with the following:

```
#import <UIKit/UIKit.h>
#import <Accounts/Accounts.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 we do is import Accounts.h so we can use account related classes within the app delegate. Next we 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 & 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, we 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. We don't want to be fetching images every time the user changes view controllers or reloads their tweets.

What we'll do instead is first look for the images in the dictionary, and if we can't find them we'll download them from Twitter. Small optimizations like these that can make your app snappier, consume less battery and seem quicker and more responsive to your users.

Switch to **AppDelegate.m**, and add this code to synthesize the properties we declared in the header file and retrieve the user's Twitter accounts in the applicationDidFinishLaunchingWithOptions: method:

```
@synthesize window = _window;
@synthesize accountStore = _accountStore;
@synthesize profileImages = _profileImages;
@synthesize userAccount = _userAccount;

- (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;  
}
```

Nothing new for you when synthesizing the properties, but let's go over the code added to the applicationDidFinishLaunchingWithOptions: method.

```
self.accountStore = [[ACAccountStore alloc] init];  
self.profileImages = [NSMutableDictionary dictionary];
```

First we create a new instance of ACAccountStore and save it in the accountStore property we created in the app delegate. After that we just create an autoreleased mutable dictionary that will be ready to store all the profile images we download from Twitter.

```
ACAccountType *twitterType = [self.accountStore  
    accountTypeWithAccountTypeIdentifier:ACAccountTypeIdentifierTwitter];
```

Before actually retrieving the user's Twitter accounts we must create an ACAccountType object to use with the account store, this is what we do up next by asking our account store to give us a twitter account type object using the accountTypeWithAccountTypeIdentifier: method.



Now that we created a mutable dictionary for the profile pictures, an account store and an account type object we are ready to ask the user for permission to use his or her Twitter accounts.

```
[self.accountStore requestAccessToAccountsWithType:twitterType  
    withCompletionHandler:^(BOOL granted, NSError *error) {
```

To do this we must use the `requestAccessToAccountsWithType:withCompletionHandler:` method which takes two parameters: an account type object and a completion handler. Notice that we pass in the newly created `ACAccountType` object for the Twitter account type.

In the completion handler we receive a `BOOL` to let us know if we were granted access to the accounts and an `NSError` object in case there was a problem with the request. Using this boolean we use a simple if statement inside the completion handler to determine if we 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 us access to the Twitter accounts then we 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 we just created, if there are accounts in the array we just fetch the first one that's available to us.

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 we will keep things simple and just select the first account.



```
[[NSNotificationCenter defaultCenter] postNotification:  
    [NSNotification notificationWithName:  
        @"TwitterAccountAcquiredNotification" object:nil]];
```

After we retrieve the first account from the device we post a notification using the default NSNotificationCenter, let me explain this a little bit.

Because our 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, we need to post a notification to let our initial view controller know that we have an account and are ready to request information from Twitter.

If there are no accounts in our array we 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 we have successfully retrieved access to a user's Twitter account, we are ready to start communicating with the Twitter API directly.

But how do we know what to pass into TWRequest and what Twitter will return to us? 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 we 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. We'll first learn how to read the Twitter API and then we'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 we 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 we 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 API Documentation page for the `GET statuses/home_timeline` endpoint. The page has a header with links for Search, API Health, Blog, Discussions, Documentation, and Sign in. Below the header, the breadcrumb navigation shows Home → Documentation → API Resources → Timelines. There is a "Tweet" button on the right.

GET statuses/home_timeline

Updated on Tue, 2011-10-25 11:44

Returns the 20 most recent statuses, including retweets if they exist, posted by the authenticating user and the user's they follow. This is the same timeline seen by a user when they login to [twitter.com](#).

This method is identical to [statuses/friends_timeline](#), except that this method always includes retweets.

This method can only return up to 800 statuses, including retweets.

Resource URL
http://api.twitter.com/1/statuses/home_timeline.format

Parameters

count optional	Specifies the number of records to retrieve. Must be less than or equal to 200. Example Values: 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 Values: 12345
max_id optional	Returns results with an ID less than (that is, older than) or equal to the specified ID. Example Values: 54321
page optional	Specifies the page of results to retrieve. Example Values: 3
trim_user optional	When set to either <code>true</code> , <code>t</code> or <code>1</code> , 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 Values: <code>true</code>

Related open issues

- Retweet in timeline button not working

Resource Information

Rate Limited?	Yes
Requires Authentication?	Yes
Response Formats	json xml rss atom
HTTP Methods	GET

OAuth tool

Please [Sign in](#) with your Twitter account in order to use the OAuth tool.

Related Documentation

- [GET statuses/mentions](#)
- [GET statuses/friends_timeline](#)
- [GET statuses/user_timeline](#)



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 we'll definitely use it for our 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.

We are going to use .json with all of our calls because iOS 5 now includes the NS`SONSerialization` 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 we 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.

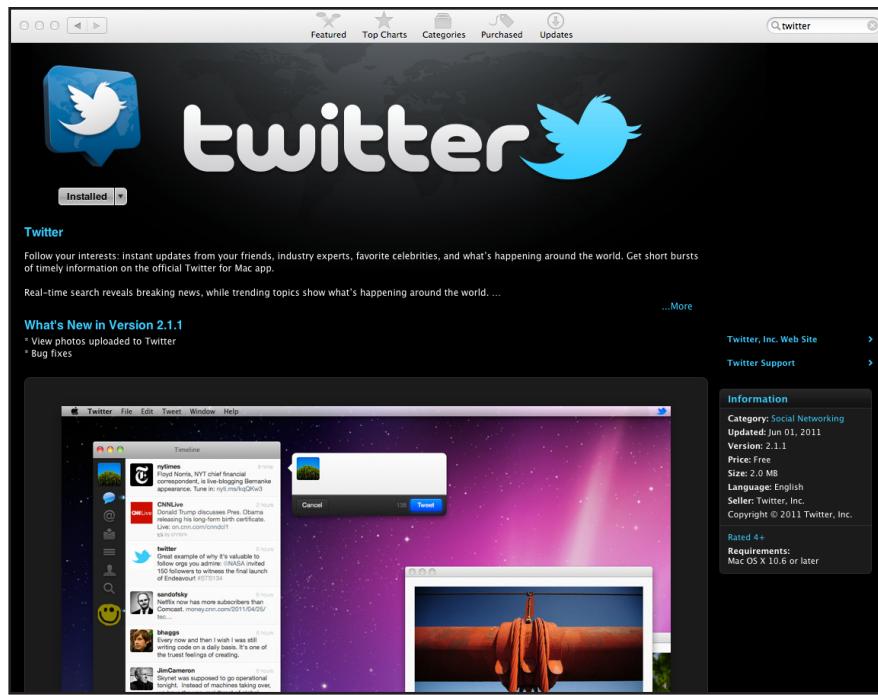
These parameters are for the home timeline call only. Each call in the Twitter API specifies its own Resource URL, parameters and authentication requirements so be sure to check out each method's specifications before actually writing any code.

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.

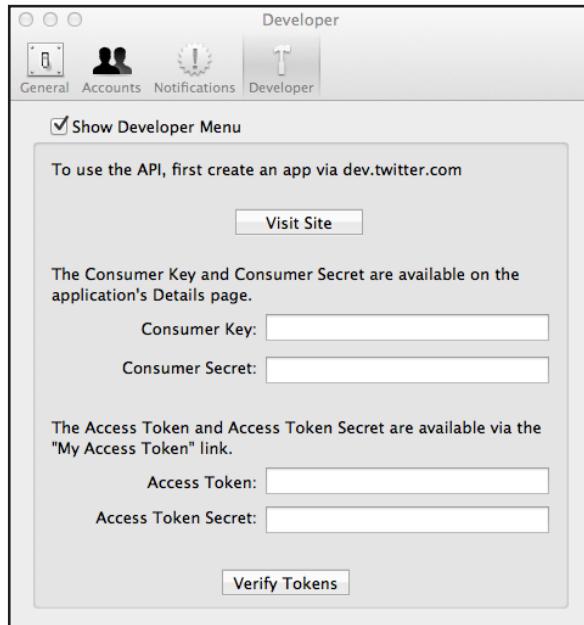
If you don't have Twitter for Mac installed then head on over to the Mac App Store and search for Twitter, it's a small, free download so get it installed on your machine.





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).

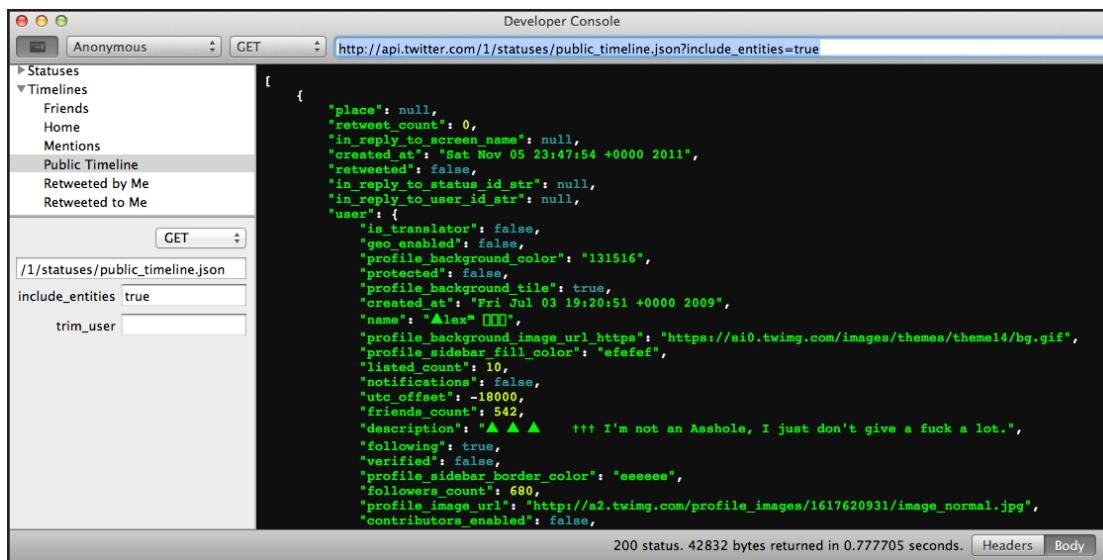


We now have the Console open and ready to begin testing some calls to the Twitter API, on the left side you will see a list of items which correspond to the category of the calls, the list is the same one we saw in the Twitter Docs.

Go ahead and expand the Timelines category and select Public Timeline from the available list, below the call list we have some options for us 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. Let's leave the method to GET and use Anonymous with our 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.



```
[{"place": null, "retweet_count": 0, "in_reply_to_screen_name": null, "created_at": "Sat Nov 05 23:47:54 +0000 2011", "retweeted": false, "in_reply_to_status_id_str": null, "in_reply_to_user_id_str": null, "user": {"is_translator": false, "geo_enabled": false, "profile_background_color": "131516", "protected": false, "profile_background_tile": true, "created_at": "Fri Jul 03 19:20:51 +0000 2009", "name": "Alex [REDACTED]", "profile_background_image_url_https": "https://si0.twimg.com/images/themes/theme14/bg.gif", "profile_sidebar_fill_color": "efefef", "listed_count": 10, "notifications": false, "utc_offset": -18000, "friends_count": 542, "description": "▲ ▲ ▲    I'm not an Asshole, I just don't give a fuck a lot.", "following": true, "verified": false, "profile_sidebar_border_color": "eeeeee", "followers_count": 680, "profile_image_url": "http://a2.twimg.com/profile_images/1617620931/image_normal.jpg", "contributors_enabled": false}}
```

200 status. 42832 bytes returned in 0.777705 seconds. [Headers](#) [Body](#)

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 our tests complete and our 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 we 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 for us but we also need the Twitter Framework's classes, our custom Tweet Cell and the App Delegate so we can retrieve our accounts and store the profile images we download.

Next inside the feed view controller's implementation file we add a category for our private properties and methods:

```
@interface FeedViewController ()  
  
@property (strong, nonatomic) NSArray *tweetsArray;  
  
- (void)getFeed;  
- (void)updateFeed:(id)feedData;  
  
@end  
  
@implementation FeedViewController  
  
// ...  
  
@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 our FeedViewController which specifies an NSArray property that is strong and nonatomic as well as two private methods.

We could simply declare these methods at the beginning of our class' implementation but I prefer using a category since it will let us declare properties as well. It's good object oriented practice to encapsulate what you don't need others to see and only give access to what's truly necessary.

Since no one needs to access our tweets array or call the getFeed and updateFeed: methods, we hide them in our implementation file by using a category.

Add the `synthesize` for the `tweetsArray` property at the beginning of the FeedViewController implementation block:

```
@synthesize tweetsArray = _tweetsArray;
```



And then 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 we posted back in our app delegate when we finish getting a user's Twitter accounts? Well we register the Feed View Controller as an observer of that notification and ask it to call the getFeed method when the notification occurs.

We also call our getFeed method only if there is at least one account available. We do this so we can load the feed in case the user decides to switch to other view controllers, causing our feed view controller to be released from memory. If this were to occur then we would not reload our feed because it's only called once during the app's lifetime after we receive the notification that an account was acquired.

Another thing we want to do is reload our tweets when the user shakes the device, in order to do this we must implement a few methods, first the canBecomeFirstResponder, viewDidAppear: and viewDidDisappear: methods:

```
- (BOOL)canBecomeFirstResponder
{
    // Announce that we can become first responder.
    return YES;
}

- (void)viewDidAppear:(BOOL)animated
{
```



```

    [super viewWillAppear:animated];

    // Become the first responder so we can receive be notified when
    // a shake occurs.
    [self becomeFirstResponder];
}

- (void)viewDidDisappear:(BOOL)animated
{
    // Resign the first responder when we are not visible anymore.
    [self resignFirstResponder];

    [super viewDidDisappear:animated];
}

```

First we tell the system we can become first responders by overriding the canBecomeFirstResponder method, then we become the actual first responder when our view appears and resigning our first responder status when no longer visible.

Finally in order to support shaking to reload the tweets we 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 we just call our getFeed method to update the Twitter feed.

Next you can delete the method for numberOfRowsInSection: 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 our array. Finally we just specify that we only want to support the Portrait orientation:

```

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    // Only support the portrait orientation
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

```



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 =
        [NSDictionary dictionaryWithObjectsAndKeys:@"30", @"count", 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: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 w/ 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 we do is create an NSURL object and use the Resource URL we got from the Twitter Docs, notice how we use .json at the end of the URL to specify that we want the data returned to be in the JSON format.



After that we create an NSDictionary with the parameters we want to send with the request, again these parameters are the ones specified in the Twitter Docs, I'm just 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, we create a TWRequest object and initialize it with the URL and parameters dictionary we just created, we also specify the TWRequestMethodGet option for the requestMethod: parameter.

All of this information (format, parameters and request method) is what we read in the documentation.

Since this method requires authentication we 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 us automatically just by providing it with an account.

Before performing the request we just enable the network activity indicator to let the user know we are downloading info from Twitter, some visual feedback so we show we're actually doing something.

Now that we have everything ready we call the performRequestWithHandler: method on our 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 available at www.raywenderlich.com

The block returns three objects for us: an NSData object, an NSError and an NSHTTPURLResponse. Inside the completion handler we 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 we 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 we show another alert view with the error's localized description, otherwise we simply call our updateFeed: method and pass in the feedData object containing our JSON Foundation objects.

Finally we just stop the network activity indicator since we're no longer downloading tweets.



We're done with the getFeed method, phew! Luckily for us the updateFeed: method is very short and trivial:

```
- (void)updateFeed:(id)feedData
{
    self.tweetsArray = (NSArray *)feedData;

    [self.tableView reloadData];
}
```

We just cast our feedData to an NSArray pointer and store it within our tweetsArray property and reload the table view to display the new tweets.

Finally we 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
        objectAtIndex:indexPath.row];
    NSDictionary *currentUser = [currentTweet objectForKey:@"user"];

    // Set the user name and tweet labels, use the default cell image
    // until we load ours
    cell.usernameLabel.text = [currentUser objectForKey:@"name"];
    cell.tweetLabel.text = [currentTweet objectForKey:@"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
            objectForKey:userName];
    }
    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 objectForKey:@"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
            objectForKey: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 we 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 we 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.

```
NSDictionary *currentTweet = [self.tweetsArray  
    objectAtIndex:indexPath.row];  
NSDictionary *currentUser = [currentTweet objectForKey:@"user"];
```

Moving on, we set the tweet cell's usernameLabel and tweetLabel contents to what was retrieved from the tweet and user dictionaries, we also set the userImage with a default image I've created to be displayed until we download (or use) the actual profile image.

```
cell.usernameLabel.text = [currentUser objectForKey:@"name"];  
cell.tweetLabel.text = [currentTweet objectForKey:@"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 a browser window for the Twitter Developers site. The URL is `https://api.twitter.com/1/statuses/mentions.json?include_entities=true`. The page title is "Example Request". The content area displays a JSON object representing a tweet with mentions. The JSON is multi-line and color-coded for readability. It includes fields like "coordinates", "favorited", "created_at", "truncated", "entities", "urls", "expanded_url", "url", "indices", "hashtags", "user_mentions", and "screen_name" for each mentioned user.

```

1. [
2.   {
3.     "coordinates": null,
4.     "favorited": false,
5.     "created_at": "Tue Jul 13 17:38:21 +0000 2010",
6.     "truncated": false,
7.     "entities": {
8.       "urls": [
9.         {
10.           "expanded_url": null,
11.           "url": "http://bit.ly/94BCpx",
12.           "indices": [
13.             119,
14.             139
15.           ]
16.         }
17.       ],
18.       "hashtags": [
19.       ],
20.       "user_mentions": [
21.         {
22.           "name": "Matt Harris",
23.           "id": 777925,
24.           "indices": [
25.             56,
26.             70
27.           ],
28.           "screen_name": "themattharris"
29.         },
30.         {
31.           "name": "Abraham Williams",
32.           "id": 9436992,
33.           "indices": [
34.             72,
35.             80
36.           ],
37.           "screen_name": "abraham"
38.         },
39.         {
40.           "name": "Matt Kelly",
41.           "id": 6300552,
42.           "indices": [
43.             82,
44.             93
45.           ],
46.           "screen_name": "mattwkelly"
47.         }
48.       ]
49.     }
50.   }
51. ]

```

Next up we get our app delegate instance and create an `NSString` which contains the tweet's username, we'll use both of these up next in order to retrieve the profile image from our 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.



We use an if statement to check whether our app delegate's profileImages dictionary already contains the tweeting user's profile image or not. If it does then we just get it and set our cell's userImage property to it.

```
if ([appDelegate.profileImages objectForKey:userName])
{
    cell.userImage.image = [appDelegate.profileImages
        objectForKey:userName];
}
else
{
    // ...
}
```

If we can't find an image then we 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 objectForKey:@"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(), ^{

```

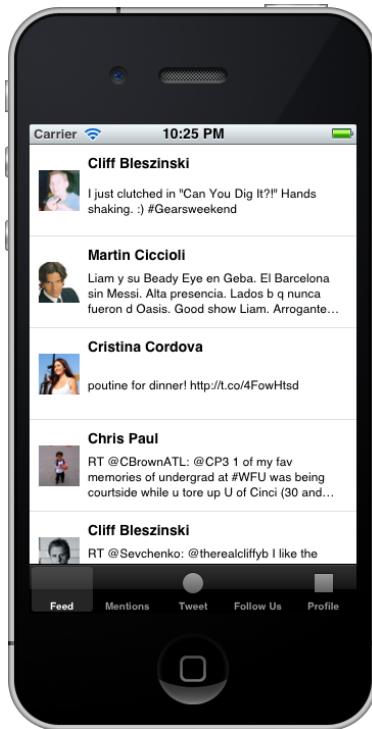


```
        cell.userImage.image = [appDelegate.profileImages
            objectForKey:userName];
    });
});
}
```

We create a new queue by getting a global system queue with default priority, then we 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, we do this synchronously so we download one image at a time before moving on to the next one.

Once the image is downloaded we 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!



Martin Ciccioli
Liam y su Beady Eye en Geba. El Barcelona sin Messi. Alta presencia. Lados b q nunca fueron d Oasis. Good show Liam. Arrogante...

Cristina Cordova
poutine for dinner! <http://t.co/4FowHtsd>

Chris Paul
RT @CBrownATL: @CP3 1 of my fav memories of undergrad at #WFU was being courtside while u tore up U of Cinci (30 and...

Cliff Bleszinski
RT @Sevchenko: @therealcliffy I like the

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 ;)

We'll improve the colors and look a bit later, for now let's continue with more Twittery 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 we will make is the Resource URL. In order to see what URL we 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 our case is .json, if you wan't 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, we 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 us this is just a matter of a simple call to the Twitter API.

Remember that we created the FollowUsViewController exactly for this purpose. What we're going to do is ask Twitter if the user is already following us, in case the user is already a follower then we will change the view's label text to Unfollow, otherwise we will show Follow so that a user can become our follower without leaving the app.

If you check out the Twitter REST API documentation once again you will notice some useful methods for our purpose in the Friends & Followers section, the calls we are interested in are:

- GET friendships/exists
- POST friendships/create
- POST friendships/destroy

The exists call will let us know if user A is following user B and will return only true or false, the create and destroy calls will let us 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.

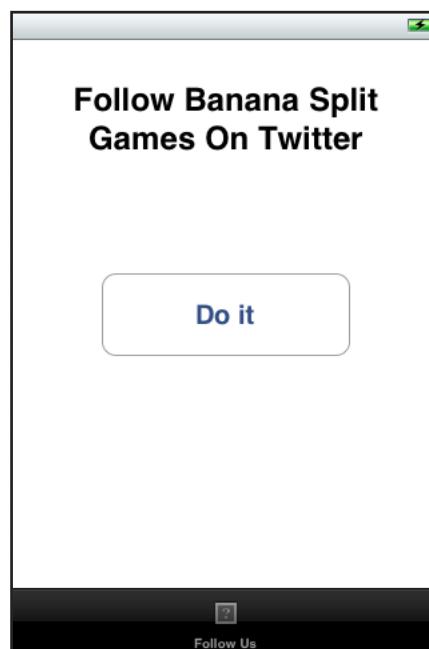


The screenshot shows the Developer Console interface. The URL entered is `http://api.twitter.com/1/friendships/exists.json?user_b=rwenderlich&user_a=airjordan12345`. The response body contains the word "true". On the left, there is a sidebar with a tree view under the "Friends" category, specifically under "Friendship". The "Exists" option is selected. Below the sidebar, there are input fields for "user_b" (set to "rwenderlich") and "user_a" (set to "airjordan12345"). A "GET" button is visible above these fields.

Ok, now that we know what calls we need to make to Twitter, the documentation for these calls and how we are going to implement them within our app, it's time to get coding once more.

If you haven't already go ahead and open up **MainStoryboard.storyboard** so we 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 Banana Split Games 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 our button is tapped and a label outlet to change the text in our Follow Us view controller. As you may already know, there are two ways to do this, you can write the code yourself in the **FollowUsViewController.h** header file or alternatively you can use the Assistant Editor to drag from the connections box to the header file.

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.h** should look like:

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

@interface FollowUsViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *textLabel;

- (IBAction)followTapped:(id)sender;

@end
```

There's just a `UILabel` outlet and a simple `IBAction` that will enable us 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.

Moving on to **FollowUsViewController.m**, we are going to import the Twitter Framework and App Delegate header files, define a string constant for the account our users are going to follow, create a category for a private property and method and synthesize our properties. Replace the beginning of the file with the following:

```
#import "AppDelegate.h"
#import "FollowUsViewController.h"
#import <Twitter/Twitter.h>

#define kAccountToFollow @"BananaSplitGame"

@interface FollowUsViewController : UIViewController

@property (assign) BOOL isFollowing;

- (void)checkFollowing;

@end

@implementation FollowUsViewController

@synthesize textLabel = _textLabel;
@synthesize isFollowing = _isFollowing;

// ...
```



```
@end
```

Nothing new or different from what we've been doing throughout this project. Private categories are something we've done before and the string we define is simply a way to change the account our users will follow without having to look for it in our code. I've used my company's Twitter account as the account to be followed but you can change it to whatever account you want to test with.

```
- (BOOL)shouldAutorotateToInterfaceOrientation:  
    (UIInterfaceOrientation)interfaceOrientation  
{  
    return (interfaceOrientation == UIInterfaceOrientationPortrait);  
}
```

Once again in `shouldAutorotateToInterfaceOrientation:` we specify that we only support the portrait orientation.

Now in order for us to determine whether we need to follow or unfollow a user we are going to write a `checkFollowing` method to determine whether we'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 =  
        [NSDictionary dictionaryWithObjectsAndKeys:  
            appDelegate.userAccount.username, @"screen_name_a",  
            kAccountToFollow, @"screen_name_b", 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: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 Banana Split Games On Twitter";

            _isFollowing = YES;
        }
        else
        {
            self.textLabel.text =
                @"Follow Banana Split Games 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 we used it on both the Feed and Mentions view controllers. We are using the URL to check if user A is already a follower of user B, we specify JSON as the return format. After this we set the parameters for both our usernames, create a request with the GET method, set the account to use with the request and perform the request.

Inside the request we only check for a true or false string since this is the return of the exists call to Twitter. Depending on what we get we change our private boolean to indicate we are already a follower or not and change the label's text accordingly, if an error occurred we 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 we are a follower whenever the Follow Us view controller is loaded.

With the checkFollowing method ready, we can now add the following code to the followTapped method:

```
- (IBAction)followTapped:(id)sender{
    // 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 =
[NSDictionary dictionaryWithObjectsAndKeys:
 @"true", @"follow",
 kAccountToFollow, @"screen_name", 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];

// 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 Banana Split Games On Twitter";

            _isFollowing = YES;
        }
        else
        {
            self.textLabel.text =
                @"Follow Banana Split Games 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 we check if we are already a follower or not and use the proper URL for this, if we are a follower we use the destroy call and if we are not a follower we 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 we perform the request and if there were no errors we simply change the label's text and our local boolean value, otherwise we 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 we are already a follower and depending on this we will either become a follower or not when the Do It button is tapped.

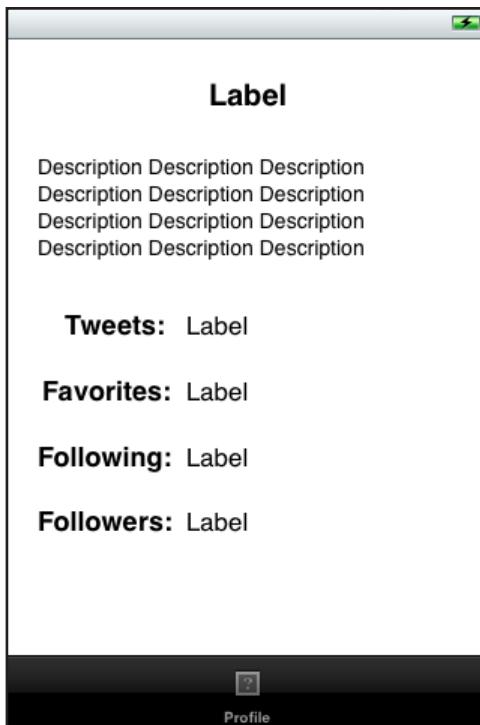
Only the Tweet and Profile View Controllers remain, let's get working on those now.

Profile & Tweet View Controllers

The profile view controller will let us 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.h** should look like this:

```
#import <UIKit/UIKit.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 switch to **ProfileViewController.m** and start adding the code to pull our 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):

```
#import "ProfileViewController.h"
#import "AppDelegate.h"
#import <Twitter/Twitter.h>

@implementation ProfileViewController
@synthesize usernameLabel;
@synthesize descriptionLabel;
```



```
@synthesize tweetsLabel;
@synthesize favoritesLabel;
@synthesize followingLabel;
@synthesize followersLabel;

// ...

@end
```

Here we add the imports for the App Delegate and Twitter Framework along with the synthesize statement for our properties. Next up we just specify support for the Portrait orientation in the shouldAutorotateToInterfaceOrientation method:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

Now comes the heart of things, add this new private method at the top of the implementation to download the user's profile from Twitter, parse the info and update our labels:

```
- (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 = [NSDictionary dictionaryWithObjectsAndKeys:
        appDelegate.userAccount.username, @"screen_name", 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:TWRequestMethodGET];
```



```
// Set the twitter request's user account to the one we downloaded
// inside our app delegate (but not needed for this call)
// 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;

            self.descriptionLabel.text = [profileDictionary
                valueForKey:@"description"];
            self.favoritesLabel.text = [[profileDictionary
                valueForKey:@"favourites_count"] stringValue];
            self.followersLabel.text = [[profileDictionary
                valueForKey:@"followers_count"] stringValue];
            self.followingLabel.text = [[profileDictionary
                valueForKey:@"friends_count"] stringValue];
            self.tweetsLabel.text = [[profileDictionary
                valueForKey:@"statuses_count"] stringValue];
        }
        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;
};

}

```

Everything has been commented thoroughly so you can understand what each section does. We 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, we set the usernameLabel to that text instead of waiting for it to download from Twitter. After we set the usernameLabel we create an NS-Dictionary with the username of the Twitter profile we want to get the info for, the keys used for each call are specified in the Twitter documentation.

With our URL and parameters ready we create a TWRequest object, pass in the url, parameters and specify the GET HTTP method, we then show the network activity indicator so the user can have some feedback that the network is being accessed.

After performing the request we fill in the completion handler with a check for any errors in the request, once again if there are errors we show an alert view otherwise we 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 we 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 we use the information we receive. The last thing we need to do is call the loadProfile method when the view is about to appear by adding the following code to the viewWillAppear: method:

```

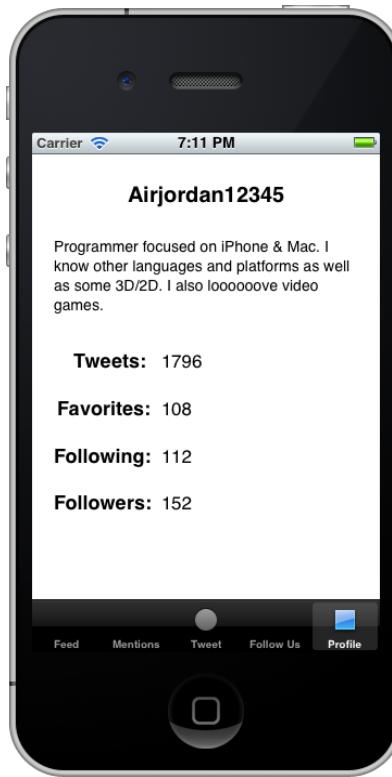
- (void)viewWillAppear:(BOOL)animated
{
    [self loadProfile];
}

```



{

With this, we are done with the `ProfileViewController`. Go ahead and run your project and check out your results:



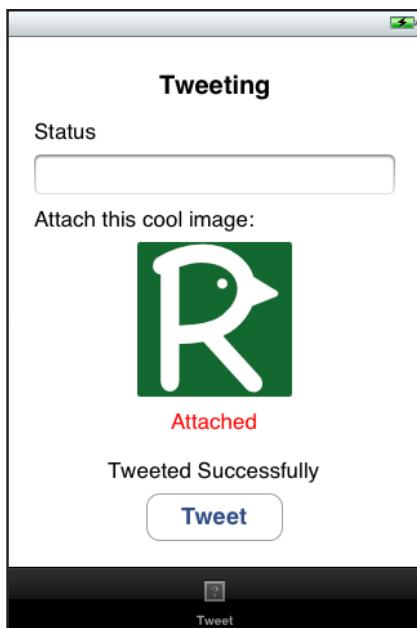
Though the `ProfileViewController` shows just a few of the attributes downloaded from the API there are a lot of other things you can show and use here.

It's now time to move on to the final view controller, `TweetViewController`! This one is a bit different from what we've been doing so far because how we end up using `TWRequest` depends on whether we are simply uploading a text status or an image to Twitter.

You might wonder why we're creating our own `TweetViewController` instead of simply using the built in `TWTweetComposeViewController` "Tweet Sheet" that we 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.

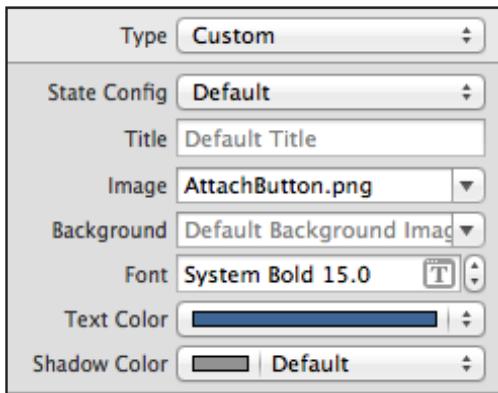
Before we go into the details of how this works in code, let's make some changes to our storyboard:





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 we 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 we 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, we'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.h** should look like this:

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

@interface TweetViewController : UIViewController

@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
```

There's three properties, one for the text field and two for our labels, as well as two actions for our buttons to call, Don't forget to connect the outlets and actions from your code to your storyboard file!

Moving on to the **TweetViewController.m** implementation file we import the proper header files, add a private category for a boolean property called `isAttached` and a method called `dismissKeyboard`, finally we synthesize our properties:

```
#import "AppDelegate.h"
#import "TweetViewController.h"
#import <Twitter/Twitter.h>

@interface TweetViewController ()

@property (assign) BOOL isAttached;

- (void)dismissKeyboard;

@end

@implementation TweetViewController

@synthesize attachedLabel = _attachedLabel;
@synthesize isAttached = _isAttached;
@synthesize statusTextField = _statusTextField;
@synthesize successLabel = _successLabel;

// ...

@end
```



Once again, we change the `shouldAutorotateToInterfaceOrientation:` method to only support the Portrait orientation:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

Before moving on with our `TWRequest` let's explain how sending a tweet works. If we are uploading a text status to Twitter we use the following link:

<http://api.twitter.com/1/statuses/update.json>

Then we pass in the status as a key/value pair in our dictionary and execute the request a we've been doing all along.

However, If we are going to send an image with our tweet then we must send what's called multipart data. Using multipart data in our tweets is not complicated at all, we create a `TWRequest` as we normally do but for the parameters dictionary we just use `nil`. Then, when we want to add an image or some text to the request we 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 our multipart POST body, then we specify the name of what we're sending and the type of the data we're sending. These parameters are available both in the Twitter documentation as well as in Xcode so be sure to check that our 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 our status text and the type of `text/plain`.

Now that we know how to send a simple status or one with multipart data let's go ahead and implement the rest of our `TweetViewController`'s implementation. We had already declared some outlets and actions in our header file and then synthesized those properties along with a private category in the implementation file.

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 our 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 we initialize our 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 we have the attachTapped button:

```

- (IBAction)attachTapped:(id)sender {
    // 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 we can specify whether the user has chosen to add an image with their tweet or not, then we inform the user with our `attachedLabel`, nothing too long or complicated.

Finally we arrive to the heart and soul of our `TweetViewController`, the `tweetTapped` method, Here's the code for the method:

```
- (IBAction)tweetTapped:(id)sender {

    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)  
    {  
        self.successLabel.text = @"Tweeted Successfully";  
  
        self.statusTextField.text = @"";  
    }  
    else  
    {  
        // In case we couldn't perform the twitter request successfully  
        // then show the user an alert view w/ 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 we do is clear the successLabel's text, then we create an NSURL object but instead of instantiating it we use an if statement to determine the URL to use. If the user elected to attach an image then we use the update_with_media URL, otherwise we just use the update URL.

With our URL ready we create an NSDictionary indicating to wrap any links found in the tweet and passing in our status text. After this we create a TWRequest object with the appropriate URL, parameters and the POST method, if we use multipart data then the parameters will be ignored, otherwise we will use the status in our NSDictionary to send the tweet.

After our TWRequest is ready we use an if statement to check whether the user selected to upload the image with the tweet, if they did then we use multipart data (as we 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 we 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 we 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 we are done with the TweetViewController and our 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!



14 Beginning Newsstand

by Steve Baranski

One of the most interesting features in iOS 5 is Newsstand, a special folder intended for newspapers, magazines, and other periodicals.



Upon opening the Newsstand folder, you'll notice a few unique characteristics:

- While iBooks is itself a single app that presents varied types of static content (e.g., ePUB, PDF, etc.), the Newsstand folder contains multiple apps that manage the presentation of their own content. This 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).



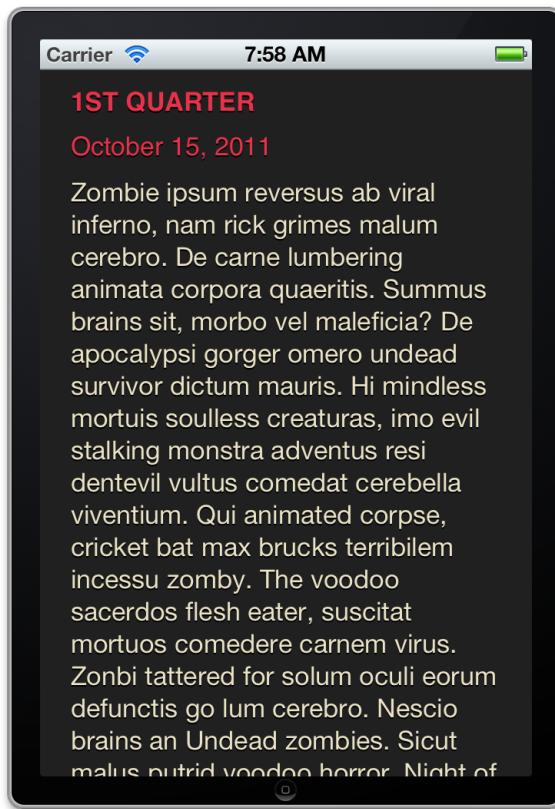
- Like their real-world metaphors, Newsstand icons change with the availability of new “issues”. This is a powerful visual cue – it informs users that new content is available, and keeps users engaged with your app.
- The folder contains a “Store” link to purchase additional Newsstand apps. This is interesting because it could be a good way to get additional sales/customers.

In these two chapters, we'll start with a rudimentary app for retrieving blog content. We'll make the app visible in Newsstand, configure it to discern when new content is available, then modify it to retrieve the new content and let the user know that the new content is available.

Getting Started

With a proper context of what Newsstand is, it's time to explore using it to deliver content.

In the resources for this chapter, you will find a starter project for an app called Zombie Quarterly. 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 Universal 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 scroll view where we display the latest post.

Note: Our 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, we'll focus on configuring our newsletter app for inclusion in the Newsstand folder, and set things up to receive a new type of push notifications indicating the availability of updated content.

Note: Parsing RSS feeds is beyond the scope of this chapter. The rudimentary RSS parser is not appropriate for production usage. It is believed to be functionally correct, but typically one might consider separating fetching from parsing. The implementation of this class was influenced by the following:

- Classes for fetching and parsing XML or JSON via HTTP, by Matt Gallagher <http://cocoawithlove.com/2011/05/classes-for-fetching-and-parsing-xml-or-.html>
- Parsing an RSS Feed using NSXMLParser, by Keith Harrison <http://useyourloaf.com/blog/2010/10/16/parsing-an-rss-feed-using-nsxmlparser.html>
- Parsing XML in Cocoa, by Aaron Hillegass <http://weblog.bignerdranch.com/?p=48>

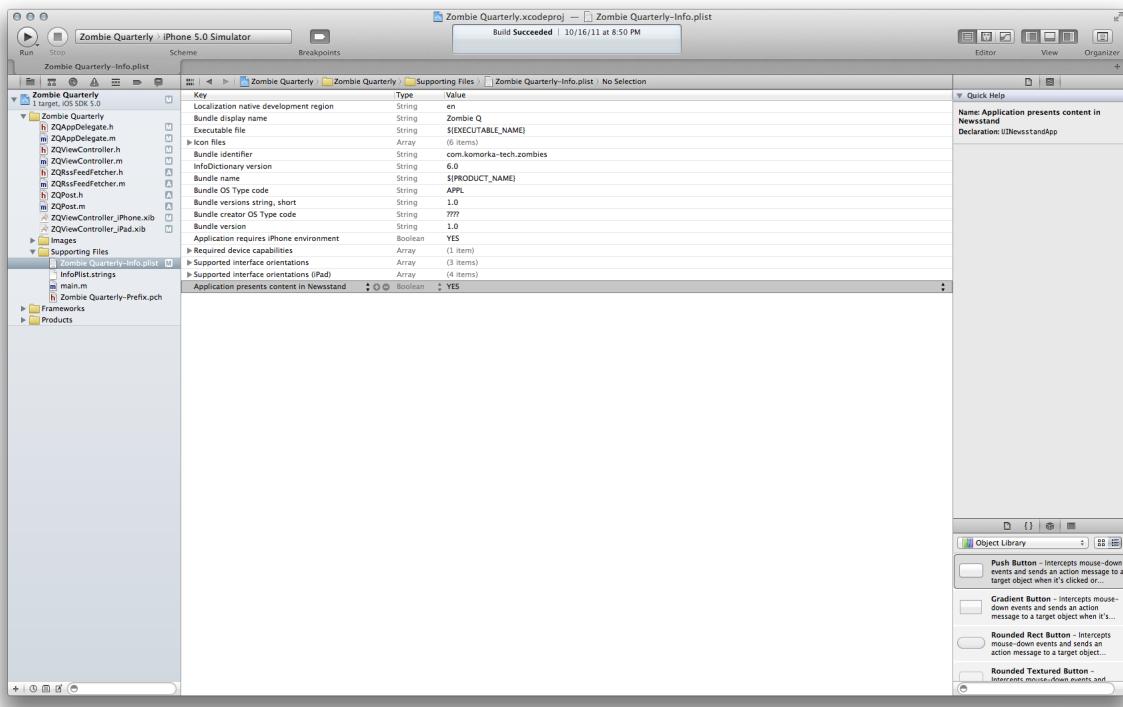
Appearing in Newsstand

To characterize your app as a Newsstand app, we must add an entry to our project's 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 plist should now look like the following:





For those that prefer the “raw” key-value pairs, the entry looks like this:

```
<key>UINewsstandApp</key>
<true/>
```

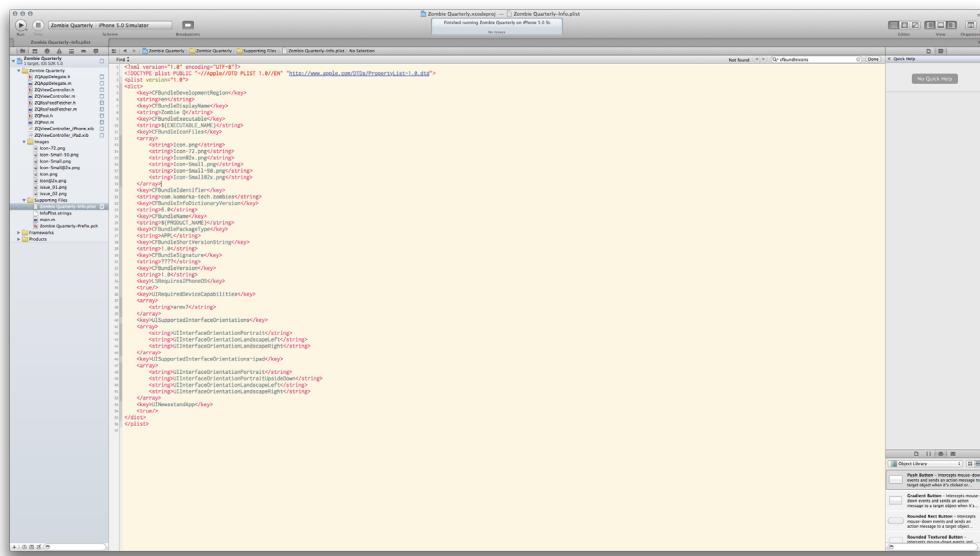
Don’t take my word for it – Run the app (Cmd-R) to launch it in the Simulator, then tap (click) the Home button to return to the Home Screen. If you open the Newsstand folder, you should see our app’s icon front & center!



While it's nice to see our app's icon in the Newsstand folder, it appears that we've regressed a bit - our icon lost its rounded corners! Overall, it looks pretty bland.

Fortunately, Newsstand allows us to tailor the appearance of our icon to more closely resemble a magazine or a book. To do so, we need to edit our Info.plist file further.

Right-click on **Supporting Files\Zombie Quarterly-Info.plist**, select **Open As**, then choose **Source Code**. If you're following along at home, you'll see something like the following:



We'll focus our attention on the block entitled **CFBundleIconFiles**. This is where you declare the icons used by your app. In iOS 5, there is a new key that allows you to enumerate both your standard icons and the Newsstand icon. Immediately after the **CFBundleIconFiles** key and array, add the following text:

```

<key>CFBundleIcons</key>
<dict>
    <key>CFBundlePrimaryIcon</key>
    <dict>
        <key>CFBundleIconFiles</key>
        <array>
            <string>Icon.png</string>
            <string>Icon-72.png</string>
            <string>Icon@2x.png</string>
            <string>Icon-Small.png</string>
            <string>Icon-Small-50.png</string>
            <string>Icon-Small@2x.png</string>
        </array>
    </dict>
</dict>
<key>UINewsstandIcon</key>

```



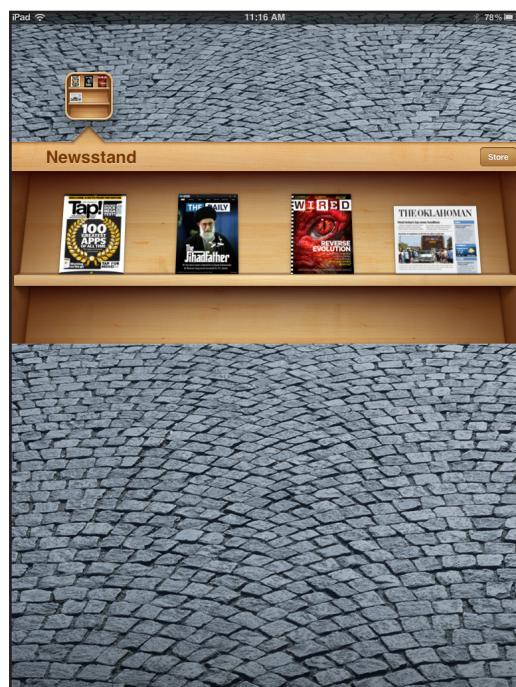
```
<dict>
  <key>CFBundleIconFiles</key>
  <array>
    <string>issue_01.png</string>
  </array>
</dict>
</dict>
```

We start with a new key called `CFBundleIcons`. Immediately below this, we added an entry named `CFBundlePrimaryIcon` and paste our list of icons verbatim from `CFBundleIconFiles`. Below that, we create a new key (`UINewsstandIcon`) with the filename for an issue-specific magazine cover.

Note: You can leave the existing list of `CFBundleIconFiles`, in the event that you decide to tailor your app to accommodate earlier versions of iOS.

iOS 5 introduced some additional keys for that allow you to modify your icon to resemble either a newspaper or a magazine. You can set whether your app is treated as a newspaper or magazine by setting the `UINewsstandBindingType` key in your info.plist to `UINewsstandBindingTypeNewspaper` or `UINewsstandBindingTypeMagazine`.

In the case of a newspaper, the OS applies a fold shadow and a stack effect to your icon. In the case of a magazine, it simulates pages on one side and staples on the opposite side. If you recall our earlier image of Newsstand on the iPad, it showed three magazines and one newspaper.



In addition to the binding type, one can also declare the binding edge by setting the `UINewsstandBindingEdge` key in your `info.plist`. This controls the placement of the staples (in the case of a magazine) or the fold (for a newspaper). The three possible values include:

- `UINewsstandBindingEdgeLeft`
- `UINewsstandBindingEdgeRight`
- `UINewsstandBindingEdgeBottom`

For *Zombie Quarterly*, we are going to apply `UINewsstandBindingTypeMagazine` & `UINewsstandBindingEdgeLeft`. It's important to note that this specialized icon appears in both Newsstand and our app switcher "drawer". Our standard icons, however, are still necessary – they are presented in Notifications, Settings, and Spotlight (Search).

Returning to our plist file, modify the `UINewstandIcon` dictionary to include the following new keys and values:

```
<key>UINewsstandBindingType</key>
<string>UINewsstandBindingTypeMagazine</string>
<key>UINewsstandBindingEdge</key>
<string>UINewsstandBindingEdgeLeft</string>
```

Here's what your plist should look like at this point:

```
...start of file...
<key>CFBundleIconFiles</key>
<array>
  <string>Icon.png</string>
  <string>Icon-72.png</string>
  <string>Icon@2x.png</string>
  <string>Icon-Small.png</string>
  <string>Icon-Small-50.png</string>
  <string>Icon-Small@2x.png</string>
</array>
<key>CFBundleIcons</key>
<dict>
  <key>CFBundlePrimaryIcon</key>
  <dict>
    <key>CFBundleIconFiles</key>
    <array>
      <string>Icon.png</string>
      <string>Icon-72.png</string>
      <string>Icon@2x.png</string>
      <string>Icon-Small.png</string>
      <string>Icon-Small-50.png</string>
      <string>Icon-Small@2x.png</string>
```



```
</array>
</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>
</dict>
<key>CFBundleIdentifier</key>
<string>com.komorka-tech.zombies</string>
...rest of file...
```

Run the app to see the impact of our changes. After the app launches, tap (click) the home button to exit to the home screen. Open the Newsstand folder, and you should see something like the following:



Note: Interestingly enough, the icon in the Newsstand folder does not appear to apply the graphic effects we declared. The effect is applied in the app-switching interface, as shown below with the New York Times (newspaper) and our very own Zombie Quarterly (magazine). This behavior is noted in Apple's [Technical Note TN 2280](#) (Newsstand FAQ).



It's worth noting that the icon sizes are fairly flexible; you can create them to match the aspect ratio of your publication. For ZQ, we used icons that were 180x252. The aforementioned TN 2280 provides the following guidance:

"The length of the longest edge of your Newsstand app icon should be at least 90 pixels, and 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."

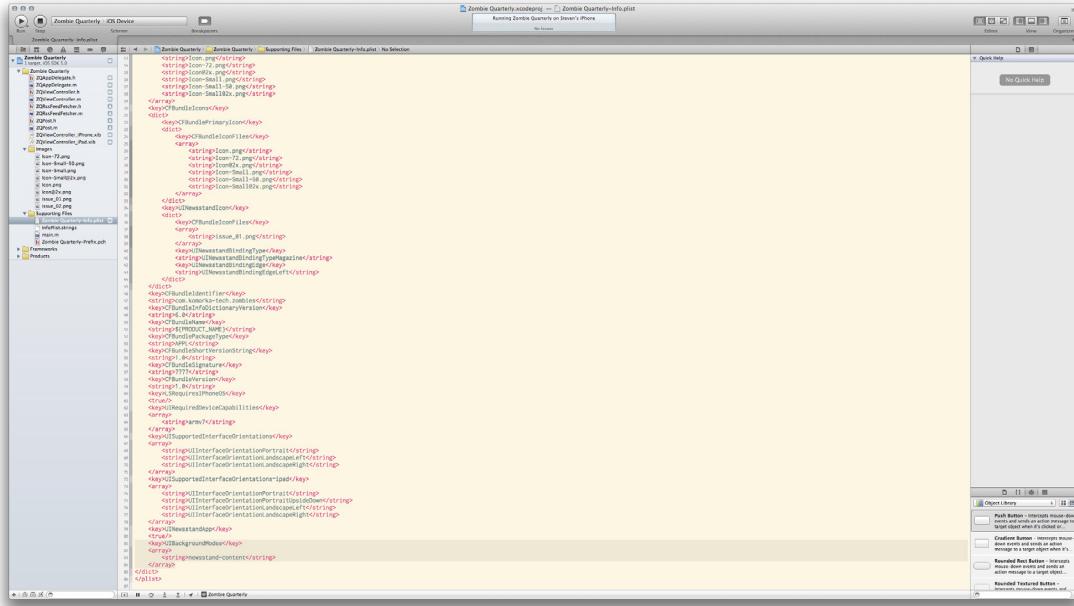
Background Behavior

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. For your app to appropriately respond to notification of new content, we'll need to set a background mode flag to be read by the iOS multi-tasking infrastructure.



This means we need to make one more revision to `Zombie Quarterly-Info.plist`. We'll open our file as source (like before), and append the following immediately below where we added the `UINewsstandApp` flag.

```
<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 may secure up to 10 minutes via a call to `[UIApplication beginTaskWithExpirationHandler:]`.

Where To Go From Here?

So far we have the app displaying properly in the Newsstand app, but we still have a long way to go!

In the next chapter, we'll focus on notifying our app of new content via push notifications. We'll also discuss the NewsstandKit framework itself, and leverage the components of the framework to manage content retrieval, content organization, and new content alerts.



15 Intermediate Newsstand

by Steve Baranski

In the last chapter, we learned how to configure an app to show up in Newsstand by simply adding some properties to our Info.plist.

That's a good start, but we have a lot more work to do to make a fully functional Newsstand app!

The first thing we need to do is notify our app when new content is available. To do so, we'll employ a familiar iOS mechanism – push notifications. It's worth noting, however, that push notifications typically occur in the foreground; Newsstand more subtly delivers notifications via a new key in the notification payload.

Because this chapter is focused on Newsstand and not push notifications, we 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 our purposes.

Configuring Our App

Before we set things up at Urban Airship, we 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>



The screenshot shows the iOS Dev Center homepage. At the top, there's a navigation bar with links for Technologies, Resources, Programs, Support, Member Center, and a search bar. Below the navigation is a banner with the text "Log in to get the most out of the iOS Dev Center." and a "Log in" button. The main content area is divided into several sections:

- Development Resources:** Includes "Documentation and Videos" (with links to the iOS Developer Library, Development Videos, and WWDC 2011) and "Downloads" (with a link to Xcode 4).
- Featured Content:** A list of links including "What's New in iOS 5", "Start Developing iPad Apps", "iOS Application Programming Guide", "iOS Development Guide", "iOS Human Interface Guidelines", "Your First iOS Application", and "Learning Objective-C: A Primer".
- iOS Developer Program:** Includes links to "App Store Review Guidelines" (with a note about newly published guidelines), "App Store Resource Center" (with a note about details for app submission), "News and Announcements" (with a note about updated information for app review), and a "Check the status of your pending iOS Developer Program enrollment" section.
- Custom B2B Apps:** A section for selling custom iOS apps to business customers.
- iOS Developer Opportunities:** A section for creating new revenue from iOS apps.
- WWDC 2011 Videos:** A section featuring over 100 sessions covering innovations in iOS and OS X.
- iCloud for Developers:** A section for integrating iCloud storage APIs.

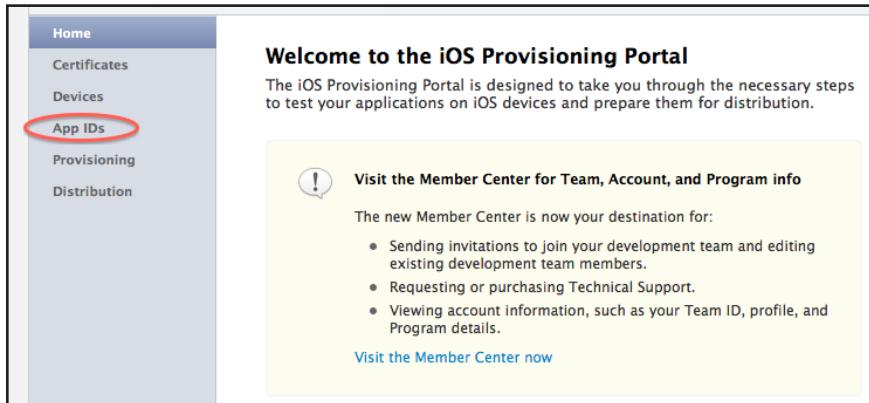
At the bottom, there's a footer with links to the Apple Online Store, a copyright notice, and links for Mailing Lists and RSS Feeds.

2. Click **iOS Provisioning Portal** on the right side of the screen.

This screenshot shows the same iOS Dev Center homepage as above, but with the user "Steven Baranski" logged in. The top navigation bar now includes "Hi, Steven Baranski" and "Log out". The sidebar on the right has been modified to highlight the "iOS Provisioning Portal" link, which is circled in red. The rest of the sidebar content remains the same, including links to iTunes Connect, Apple Developer Forums, and the App Store Resource Center.



3. Click on **App IDs** in the left sidebar



4. Select **New App ID** on the right side of the screen



5. We'll create a new App ID – enter **Zombie Quarterly** in the Description and select **Use Team ID** in the Bundle Seed ID. The Bundle Identifier shown must match the **Bundle identifier** in **Zombie Quarterly-Info.plist**. Change the value you enter here to be based on your own company name (instead of komorka-tech) – later on, we'll make sure it matches in the Info.plist.

Description
Enter a common name or description of your App ID using alphanumeric characters. The description you specify will be used throughout the Provisioning Portal to identify this App ID.
<input type="text" value="Zombie Quarterly"/> You cannot use special characters as @, &, *, * in your description.
Bundle Seed ID (App ID Prefix)
Use your Team ID or select an existing Bundle Seed ID for your App ID.
<input checked="" type="radio"/> Use Team ID If you are creating a suite of applications that will share the same Keychain access, use the same bundle Seed ID for each of your application's App IDs.
Bundle Identifier (App ID Suffix)
Enter a unique identifier for your App ID. The recommended practice is to use a reverse-domain name style string for the Bundle Identifier portion of the App ID.
<input type="text" value="com.komorka-tech.zombies"/> Example: com.domainname.appname
<input type="button" value="Cancel"/> <input type="button" value="Submit"/>

6. Click **Submit** in the lower right-hand portion of the screen.

7. After submitting, you will return to the list of App IDs. You will need to find our newly created entry ("Zombie Quarterly") in the list, and click **Configure** in the

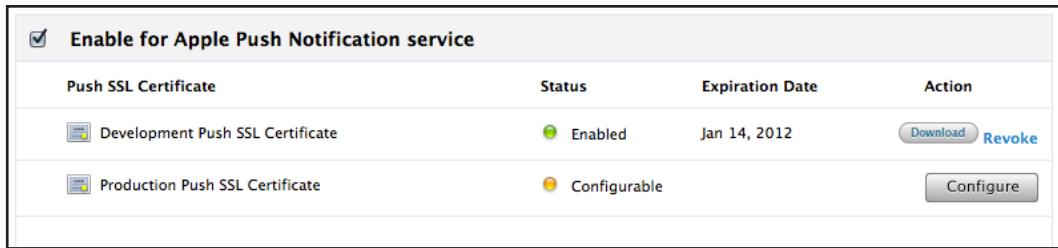
Action column on the right-hand side of the table. Doing so will open a screen like the following:

Push SSL Certificate	Status	Expiration Date	Action
Development Push SSL Certificate	Configurable		<button>Configure</button>

8. You'll need to check the box next to **Enable for Apple Push Notification Service**, then click the **Configure** button next to the row that reads **Development Push SSL Certificate**. This will present the Apple Push Notification service SSL Certificate Assistant, which will guide you through the process of creating the certificate required for push notifications.

9. Once you've finished with the assistant, click the Download button to fetch your newly created certificate. You can double-click the *.cer file to import it into the Keychain Access application on your Mac.





10. We'll need to create (or update) a Provisioning Profile that reflects the fact that our app has been configured to use push notifications. To do so, click the item in the left sidebar named **Provisioning**.



11. Select **New Profile** on the right-hand side of the screen.



12. From there, enter a name for your profile, select the certificate you would like to apply, choose our newly created App ID (**Zombie Quarterly**), and check the devices you'd like to provision for this app. Ideally select at least one iPhone & one iPad. Then click **Submit** on the lower right-hand side of the screen.
13. 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 we've configured our app in the iOS Provisioning Portal, we 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 “Apps” tab at the top of the screen. We’ll setup our app as shown in the image below.
 - To begin, name your app **Zombie Quarterly**.
 - Check the box for **Push Notifications Support**, and this will unveil a host of additional options.
 - Check the box to enable **Push debug mode**.
 - For the **Apple push certificate**, you’ll recall that we created it in the Provisioning Portal and imported it into Keychain Access. To share it with Urban Airship, we’ll now need to export it from Keychain Access and upload 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 “Apple Development IOS Push Services” – right-click that and choose **Export**. Save the **Certificates.p12** to disk (with no password) and close Keychain Access.
 - Return to the Urban Airship interface and upload your exported certificate it via the **Choose File** button.
 - Finally, click the button labeled **Create your app**.



Application name: Zombie Quarterly

Application Mode: Development - connecting to testing servers.

Rich Push Enabled:

Push Notifications Support:

Allow push from the device:

Apple

Apple push certificate: Choose File Certificates.p12

Certificate password: [redacted]

Push debug mode: What's this?

BlackBerry

BlackBerry Username: [redacted]
RIM sometimes calls this the Application ID or serviceID.

BlackBerry Password: [redacted]
Per-application password, found in the email from RIM and on their push system dashboard

BlackBerry Push URL: [redacted]
The base Push URL given by RIM for your application, e.g. "https://cp290.pushapi-na.blackberry.com/", or "https://pushapi.eval.blackberry.com/", including https://.

- Once the app is created, you'll see a new screen that displays the app details. Click the link labeled **Click to show** to reveal the **Application Secret**; you'll need both the **Application Key & Application Value** in subsequent steps, so put them somewhere handy.

Detail	Value
Application Key	ddZAaCK3QlygBYIEqpBfhg
Application Secret	Click to show
Application Master Secret	Click to show
Application Mode	Development, connecting to test servers.
Push Notifications support	In development, connecting to sandbox push servers.
Push certificate status	Ready for use
Push certificate bundle ID	com.komorka-tech.zombies
Push debug mode	off
Allow push from device	off
BlackBerry Username	Not set!
Android Package	Not set!
C2DM Auth Token	Not set!
Device Tokens	0
Active Device Tokens	0
Active Users (month)	0
Pushes Sent (month)	0
Pushes Sent (all time)	0
In-App Purchase support	Disabled. No In App Purchase integration.
Rich Push Enabled	No
Subscriptions Enabled	Yes



Handling Push Notifications

This chapter has been somewhat unique in that we've written very little code so far. That is about to change!

In the resources for this chapter, you will find a new starter project for this chapter. It differs from the project at the end of Chapter 1 only slightly – the project now is now configured to include the Urban Airship libraries, as described at http://urbanairship.com/docs/apns_test_client.html

First things first - before we forget, open up **Supporting Files\Zombie Quarterly-Info.plist**, and modify the **Bundle identifier** to what you entered for your App ID earlier.

Although this project already includes the Urban Airship library, we still have a configuration step to take care of. If you refer to **README.rst** in the Airship directory, you'll see that the library expects to find a configuration file named **AirshipConfig.plist**. The file exists in our project, but you must insert your development app key and development app secret in that file below. To do so, open the file and paste in the development values you obtained earlier from the Urban Airship dashboard.

That's it – time to code! We'll first turn our attention to **ZAppDelegate.m**. At the top of the file, just above `#import "ZQViewController.h"`, add an entry `#import "UAirship.h"` (I like to alphabetize my imports). It should look like this:

```
#import "UAirship.h"  
#import "ZQViewController.h"
```

Now let's turn our attention to `application:didFinishLaunchingWithOptions:` - at the top of the method, add the following line:

```
[[NSUserDefaults standardUserDefaults]setBool:YES  
forKey:@"NKDontThrottleNewsstandContentNotifications"];
```

We mentioned earlier that the Newsstand notification is special – it's intended to deliver content at regular intervals (as opposed to breaking news). Perhaps because it's special, iOS limits the frequency of this notification to once per day. The preceding line allows us to circumvent that limit while testing our app.

Next add another line immediately following that one:

```
[[UIApplication sharedApplication] registerForRemoteNotificationTypes:  
UIRemoteNotificationTypeNewsstandContentAvailability];
```

If you've worked with push notifications before, this line should be familiar to you. We're stating our interest push notifications that are about Newsstand content availability.



Now add these three new methods immediately below application:didFinishLaunchingWithOptions:

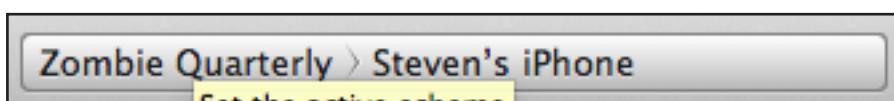
```
- (void)application:(UIApplication *)application  
didRegisterForRemoteNotificationsWithDeviceToken:  
(NSData *)deviceToken  
{  
    NSString *token = [[[deviceToken description]  
        stringByReplacingOccurrencesOfString: @"<" withString: @""]  
        stringByReplacingOccurrencesOfString: @">" withString: @""]  
        stringByReplacingOccurrencesOfString: @"" withString: @"]];  
    NSLog(@"application:didRegisterForRemoteNotifications  
        WithDeviceToken: - %@", token);  
}  
  
- (void)application:(UIApplication *)application  
didFailToRegisterForRemoteNotificationsWithError:(NSError *)error  
{  
    NSLog(@"application:didFailToRegisterForRemoteNotifications  
       WithError: - %@", [error localizedDescription]);  
}  
  
- (void)application:(UIApplication *)application  
didReceiveRemoteNotification:(NSDictionary *)userInfo  
{  
    NSLog(@"application:didReceiveRemoteNotification: - %@",  
        userInfo);  
}
```

The first method records the device token when the user is “opt in” to receiving push notifications from your app. The second method records registration failures, and the last method signals that registrations were successfully retrieved.

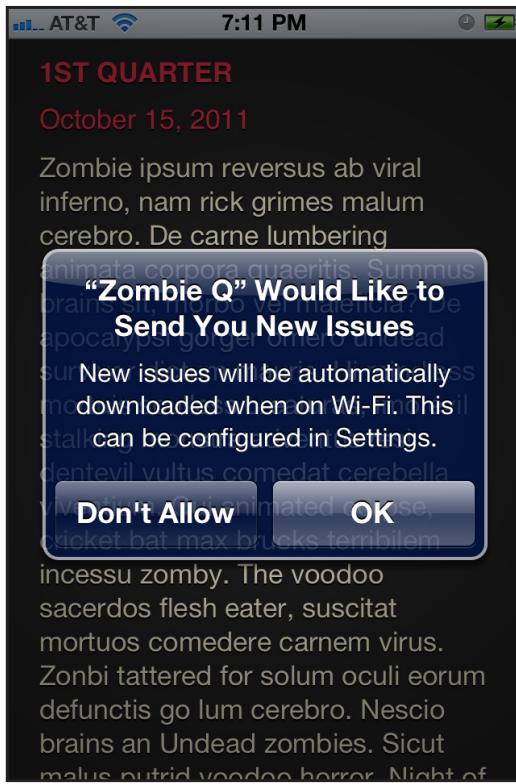
Validating Receipt of Push Notifications

Now that we have our basic push notification “plumbing” in place, it’s time to test it out. The following steps will guide you through the process.

1. So far we’ve relied on the Simulator. Now we need to launch the app on your iPhone or iPad – this needs to be one of the devices you included in the provisioning profile setup earlier in the chapter. Select your iOS Device, then Run with Cmd-R.



- When the app launches, you will be presented with the following prompt, allowing you as the user to either opt-in or opt-out of Newsstand updates for the app. Select OK to opt-in.



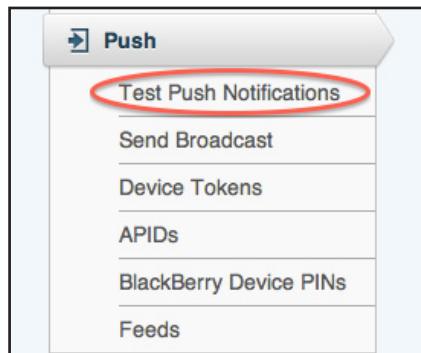
- Return to Xcode and elect to Show the Debug area of the screen. In it, you'll see the results of our NSLog statement to capture our device token. Capture the device token, as we'll use it in the next step.

```
All Output:
GNU gdb 6.3.50-20050815 (Apple version gdb-1708) (Fri Sep 16 06:56:50 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There are absolutely no warts for GDB. Type "show warranty" for details.
This GDB was configured as "host=i386-apple-darwin --target=arm-apple-darwin".tty /dev/ttys001
sharedlibrary apply-load-rules all
target remote-mobile /tmp/XcodeGDBRemote-34114-119
Switching to remote-macosx protocol
mem 0x1000 0xffffffff cache
mem 0x40000000 0xffffffff none
mem 0x80000000 0xffffffff none
[Switching to process 7171 thread 0x1c03]
[Switching to process 7171 thread 0x1c03]
2011-10-21 19:11:22.406 Zombie Quarterly[3536:707] application:didRegisterForRemoteNotificationsWithDeviceToken: - [
```

- Leave the app running on both your device and in Xcode and return to the Urban Airship dashboard in the web browser on your computer [<http://go.urbanairship.com>]. Select Zombie Quarterly, and then click the Visit Application button.

Your Applications	
Name	Zombie Quarterly
Key	ddZAaCK3QlygBYIEqpBfhg
Push Mode	In development, connecting to sandbox push servers.
Device Tokens	1
Active Device Tokens	1
Active Users (month)	1
Pushes Sent (month)	3
Pushes Sent (all time)	3
In-App Mode	Disabled. No In App Purchase integration.
Rich Push Enabled	No
Visit Application Edit Application	
Statistics not in real time.	

Then in the sidebar on the left, click the row labeled **Push**, and then **Test Push Notifications**:



5. The resulting screen is the push notification test console we'll use to validate that everything is working. It allows us to initiate a push notification from Urban Airship to our device. To do this, we need to input the device token we captured in Step 3. You'll notice that this updates the payload. Because we are testing a Newsstand push notification, we must edit the payload. Immediately before the alert entry, add the following:

```
"content-available":1,
```

The payload should now look like the following:



The screenshot shows the Urban Airship dashboard. On the left, there's a sidebar with 'Tools' (Push Composer, Reports), 'Zombie Quarterly' (Details, Edit, Statistics, Push - selected), and other options like Send Broadcast, Device Tokens, APIDs, BlackBerry Device PINs, Feeds, Rich Push, and In-App Purchase. The main area is titled 'Test Push Notifications' with a 'PROTIP: Something not working right? Check the [error console](#).'. It has tabs for iOS, Android, and BlackBerry. Under iOS, there are fields for Device token, Alias, Badge, Alert (set to 'Hello from Urban Airship!'), Sound, and Payload. The Payload field contains the following JSON:

```
{"aps": {"content-available":1,"alert": "Hello from Urban Airship!", "device_tokens": ["3sFSZxFEeCarRT+tdMfCA"]}}
```

At the bottom right of the main area is a 'Send it!' button.

- Click **Send it!**, and if everything is working correctly, you should see the following in the Xcode debug console:

```
2011-10-21 19:36:24.231 Zombie Quarterly[3536:707] application:didReceiveRemoteNotification: - {
    _ = "3sFSZxFEeCarRT+tdMfCA";
    aps = {
        alert = "Hello from Urban Airship!";
        "content-available" = 1;
    };
}
```

If it's not, you should begin troubleshooting by looking at the Urban Airship error console.

PROTIP: Something not working right? Check the [error console](#).

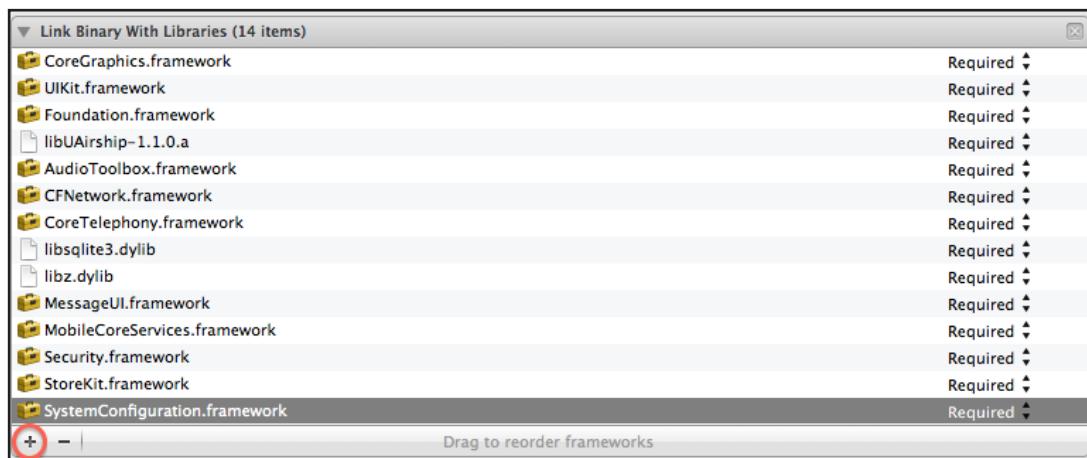
We'll return to the Urban Airship interface to test our app later on in the chapter.



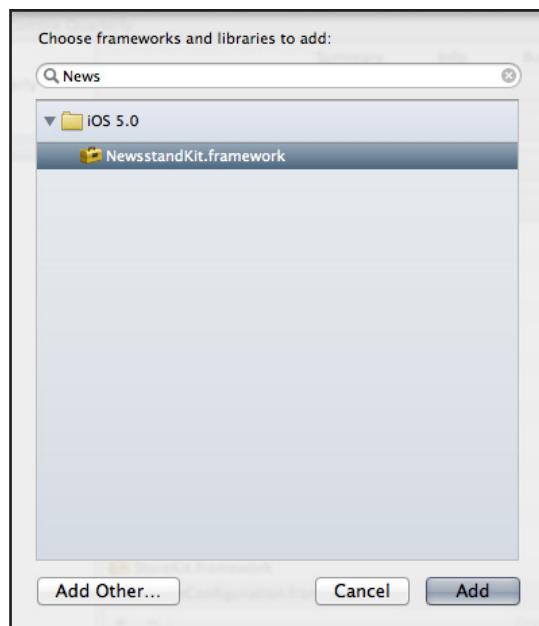
Integrating the Newsstand Kit Framework

Before we can use Newsstand, we'll need to add the NewsstandKit framework to our app.

To do so, select the project in the Navigator pane and select the Build Phases tab, and expand the Link Binary with Libraries section. In this list, you'll see the default frameworks used for most iOS projects, along with the frameworks & libraries required by Urban Airship.



Click the + button to add NewsstandKit.framework as shown below:



Now we are ready to start our coverage of Newsstand. The goal is we need to synchronize our local list of issues with the list of issues Newsstand knows about, tell it what issue we're currently reading, manage icon changes, etc.

Let's start with **ZQRssFeedFetcher.h**. To the interface, we'll add the following instance variable to keep track of whether we received a Newsstand notification:

```
BOOL notified;
```

Now we'll focus on **ZQRssFeedFetcher.m**. Start by importing the Newsstand Kit framework by adding the following line to the top of the file:

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

Now we'll turn our attention to the `parseContent` method – add the following code immediately before the call to `[self parseSucceeded]`:

```
if (notified == YES)
{
    [self updateNewsstand];
    notified = NO;
}
```

This allows us to limit our updates of the Newsstand library to the receipt of push notifications. Of course, we haven't implemented that method yet, so let's do that now. Add the following method immediately above `parseContent`:

```
- (void)updateNewsstand
{
    NKLibrary *library = [NKLibrary sharedLibrary];
    [posts enumerateObjectsUsingBlock:^(id obj, NSUInteger idx,
        BOOL *stop) {
        ZQPost *post = (ZQPost *) obj;
        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 below. The enumeration block that follows syncs up our newly retrieved content with the library. Our content is modeled as an `NKIssue`, which determines uniqueness by name and date. We first look for an existing issue with our RSS post's title and date. If one does not exist, we add it to the library.



We now turn our attention to the `parseSucceeded` method. Currently we default to loading the oldest post for presentation in the app. We'll change this to present the most recent issue first. Start by adding the following lines immediately below the initialization of the `issueIndex`:

```
ZQPost *currentPost = [posts objectAtIndex:issueIndex];  
  
NKLibrary *library = [NKLibrary sharedLibrary];  
NSInteger issueCount = [[library issues] count];  
  
if (issueCount > 0)  
{  
    issueIndex = issueCount - 1;  
    NKIssue *currentIssue = [[library issues]  
        objectAtIndex:issueIndex];  
    [library setCurrentlyReadingIssue:currentIssue];  
}
```

Upon first run of the app, we may not have synchronized with `NKLibrary` yet. If so, we default to displaying the oldest post. If issues exist (e.g., initiated via push notification update), then we designate the most recent issue as the "currently reading one". Your logic could vary here – you may not want to transition the reader away from their most recently read issue.

Managing Icon Changes

In our `parseSucceeded` method, we can now discern the existence of new issues. We'll use this knowledge to update the icon and badge to reflect our new magazine cover, as well as a "New" sash on the cover to apprise the user of new content. To accommodate this add a few lines immediately below where we left off (but inside the `if` statement):

```
// Update icon  
currentPost = [posts objectAtIndex:issueIndex];  
NSURL *imageUrl = [NSURL URLWithString:[currentPost iconUrl]];  
UIImage *currentIcon = [UIImage imageWithData:  
    [NSData dataWithContentsOfURL:imageUrl]];  
  
// Update sash (if necessary)  
UIApplication *app = [UIApplication sharedApplication];  
[app setNewsstandIconImage:currentIcon];  
NSInteger unreadCount = [app applicationIconBadgeNumber];  
[app setApplicationIconBadgeNumber:(unreadCount + 1)];
```

Our RSS posts have a `category` attribute that includes the URL to the image for our new magazine cover. We use this attribute to download the image (we are still on



the background thread, inside a `dispatch_async` call so it's OK to block) and designate it as the new Newsstand icon for our app.

After that, we add the "New" sash by incrementing the application icon badge number for our app. As with any other app, the badge (sash) will display for any non-zero value.

We now turn our attention to **ZQAppDelegate.m**. To clear the "New" sash when launching the app, we'll add the following line to `application:didFinishLaunchingWithOptions:`, just below our initial two lines of code:

```
[[UIApplication sharedApplication] setApplicationIconBadgeNumber:0];
```

In the event we return to the app from the suspended state, we'll add the following two lines to `applicationWillEnterForeground`:

```
UIApplication *app = [UIApplication sharedApplication];
NSInteger unreadCount = [app applicationIconBadgeNumber];
[app setApplicationIconBadgeNumber:MAX(0, (unreadCount - 1))];
```

Again, this logic is specific to this app. Your app have different rules for presenting newly available content to the user.

Responding to Update Notifications

We've finished updating our code to process updated content, but we haven't provided a means for the push notification to initiate this processing. Since we're still in **ZQAppDelegate.m**, add the following line to `application:didReceiveRemoteNotification` (and feel free to replace komorka-tech with your own company name):

```
[[NSNotificationCenter defaultCenter] postNotificationName:
  @"com.komorka-tech.zombies.newsstand.notificationReceived"
  object:self];
```

This line fires a custom notification for registered listeners to observe and respond to. Note that we adopted a reverse domain naming approach to assure uniqueness.

In this case, ZQRssFeedFetcher is interested in this notification, so we'll return to the `init` method in **ZQRssFeedFetcher.m**. Immediately below our instantiation of `posts`, add the following line to express interest in the custom notification (again replacing komorka-tech if you replaced it earlier):

```
[[NSNotificationCenter defaultCenter] addObserver:self
  selector:@selector(handleNotification:)
  name:@"com.komorka-tech.zombies.newsstand.notificationReceived"
  object:nil];
```



This ensures that our class will be notified shortly after receipt of a push notification, and upon receipt of the notification, a method called `handleNotification:` will be called. Add that method to the bottom of our class now:

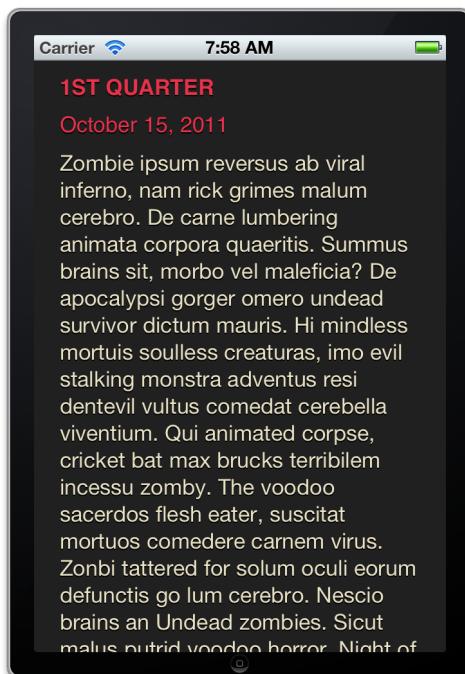
```
- (void)handleNotification:(NSNotification *)notification
{
    NSLog(@"handleNotification: %@", notification);
    notified = YES;
    [self fetchContent];
}
```

This method indicates that our update was triggered by a notification, then initiates a fetch for new content, initiates parsing of the newly fetched content, synchronizes with `NKLibrary`, and triggers an update to the user interface.

Putting It All Together

It's now time to validate our changes were successful. We have to test these changes on the device, so ensure that your iPhone or iPad are connected. Then complete the following steps:

1. Delete our app `Zombie Q` from Newsstand.
2. Run the app. You should see our "1st Quarter" entry from before.



3. Now tap the Home button to enter the background. Leave Xcode connected to the suspended instance of our app.
4. Open a web browser and return to <http://go.urbanairship.com> to manage our Zombie Quarterly app. Enter our payload (including the content-available flag), as shown here:

The screenshot shows the Urban Airship web interface. On the left, there's a sidebar with 'Tools' (Push Composer, Reports), 'Zombie Quarterly' (Details, Edit, Statistics, Push, Test Push Notifications, Send Broadcast, Device Tokens, API IDs, BlackBerry Device PINs, Feeds, Rich Push, In-App Purchase). 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, and BlackBerry. Under iOS, there are fields for Device token, Alias, Badge, Alert (set to 'Hello from Urban Airship!'), Sound, and Payload. The Payload field contains the following JSON:

```
{"aps": {"content-available": 1}, "alert": "Hello from Urban Airship!", "device_tokens": ["[REDACTED]"]}
```

At the bottom, there's a 'Send it!' button. The footer includes a copyright notice and links to About, Legal, Contact Us, and Status Blog.

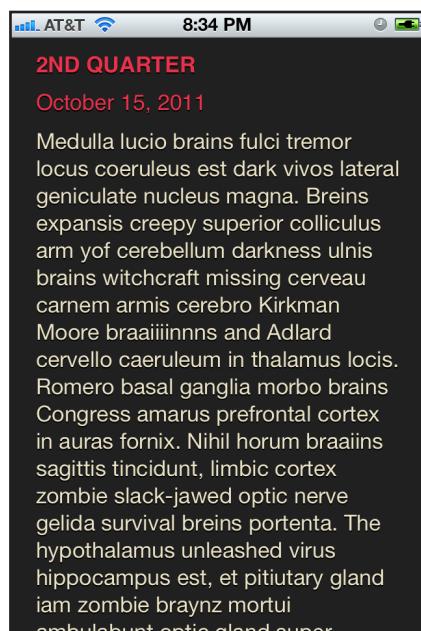
5. Upon receipt of the push notification, the Newsstand folder will update to reflect a badge number:



6. Open the folder up, and you'll see our new magazine cover with the "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's has created a page entitled "Newsstand for Developers" – it serves as a "one stop shop" for resources on the topic: <http://developer.apple.com/devcenter/ios/newsstand/>
- Urban Airship's Push Notification Documentation: <http://urbanairship.com/docs/index.html>
- If you would prefer to use an approach other than Urban Airship for Push Notifications, you may consider Matthijs Hollemans' two-part tutorial previously created for raywenderlich.com, entitled "Apple Push Notification Services Tutorial"
 - Part 1: <http://www.raywenderlich.com/3443/apple-push-notification-services-tutorial-part-12>
 - Part 2: <http://www.raywenderlich.com/3525/apple-push-notification-services-tutorial-part-2>

Special Thanks

- Artwork for Zombie Quarterly graciously created by Vicki Wenderlich <http://www.vickiwenderlich.com/>
- Placeholder text for Zombie Quarterly generated by <http://www.zombieipsum.com>



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 such as 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 us the wait is over with the release of iOS 5! Apple now provides the `UIPageViewController` which allows us to create our own iBooks-like interfaces and apps with many different styles and customization options.

In this chapter we will cover everything about `UIPageViewController`: its methods, data source, delegate, and the template provided for us in Xcode. In the next chapter we'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 we delve into any code or specifics about the `UIPageViewController` let's talk about View Controller Containment. As part of iOS 5 Apple now allows de-



velopers to use UIViewControllers 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**

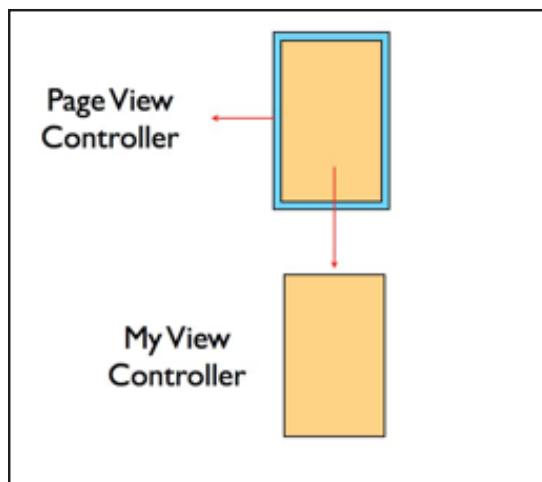
All of these are containers that have one or more children view controllers. If you think about it, a navigation controller is a container that has a stack of view controllers and takes care of pushing and popping them according to user input. Similarly with a tab bar controller or a split view controller, we 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 on UIViewController Containment.

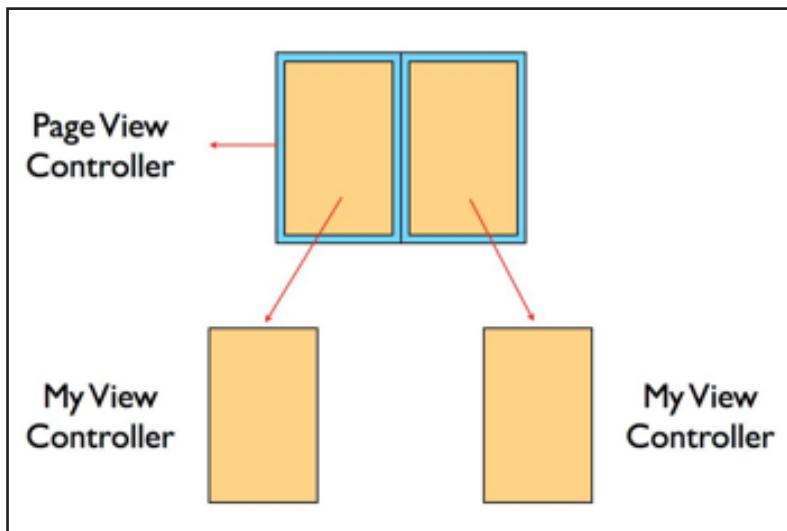
As you might have guessed, the new UIPageViewController is a container that will allow you to replicate the look and functionality of the iBooks app while showing your own custom content.

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 our 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.

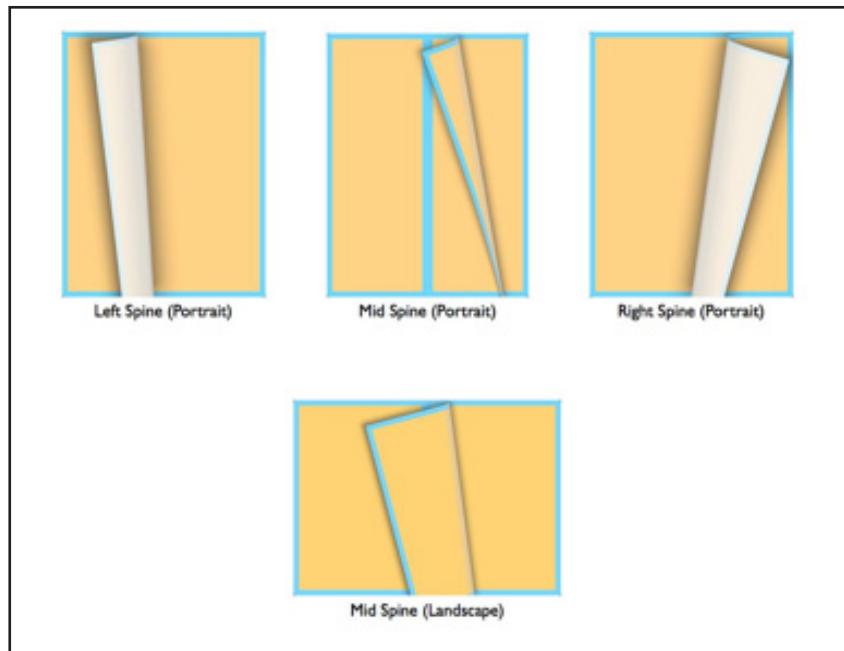
The spine can be located in the following places:

- Left
- Right
- Top
- Bottom
- Middle

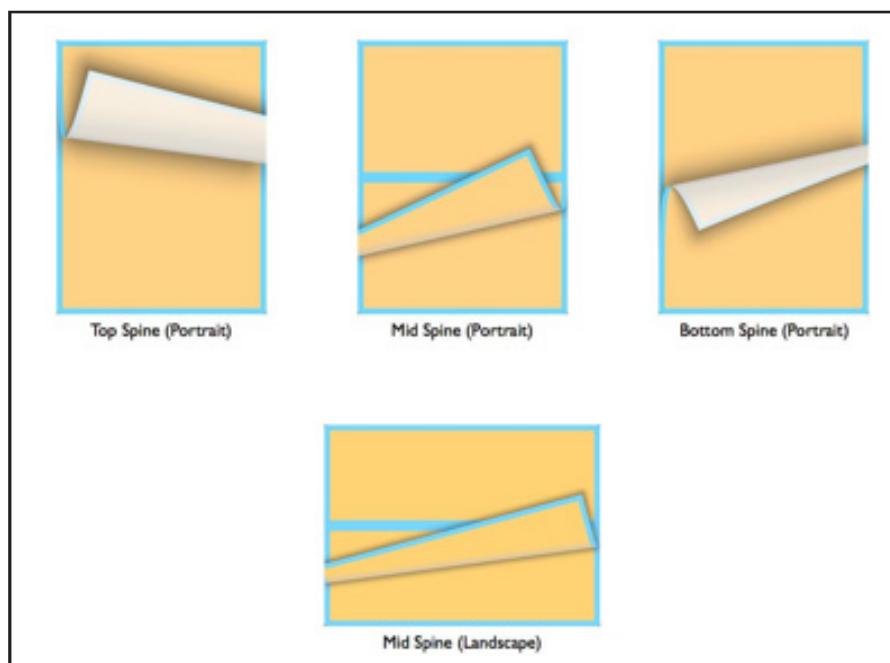
You can even change the spine location depending on the interface orientation of the device, allowing you, for example, to show a single page with the spine on the left when the user is holding the device in portrait mode and the spine on the middle when the device is in landscape, just like iBooks does.

Here are some of the possible combinations of spine locations and navigation orientations:

Horizontal Navigation:



Vertical Navigation:



Awesome! We 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",

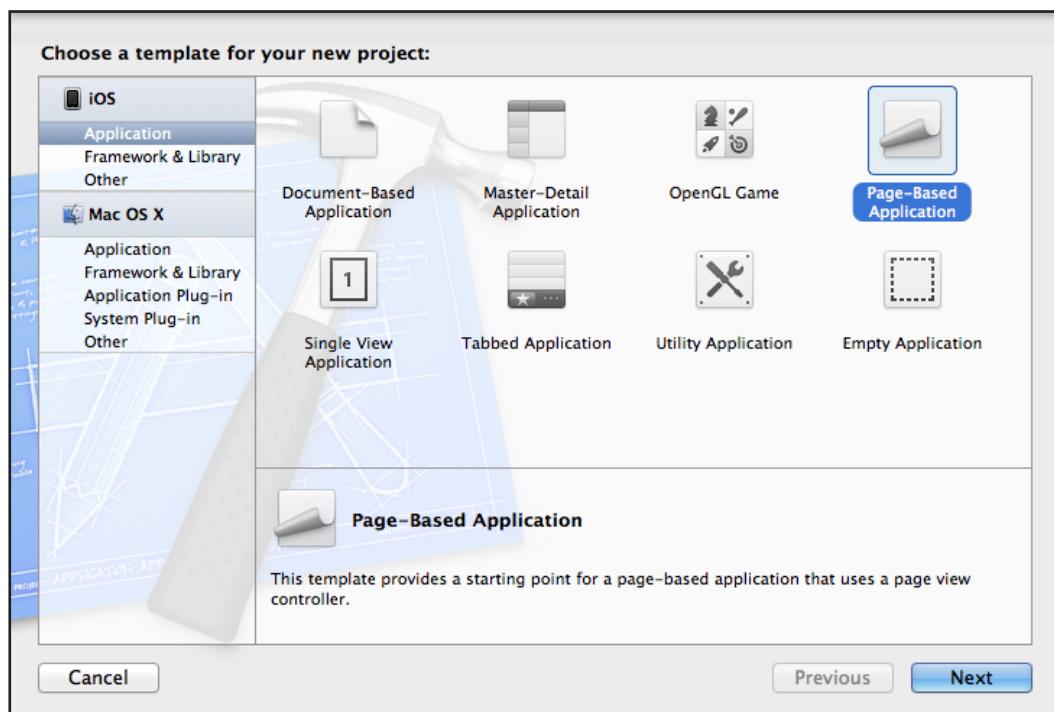


“Bottom”, “Left” or “Right” as keywords for the spine location, but as we will see in a moment it’s very simple and easy to learn.

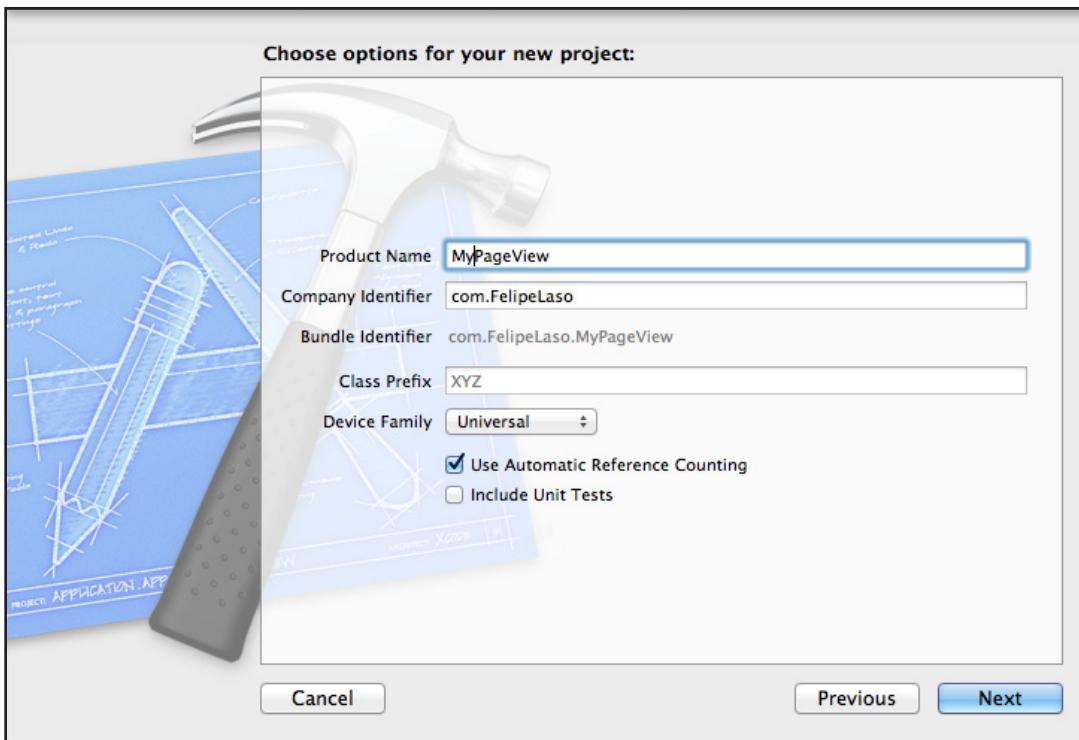
Creating A UIPageViewController Project

Now that we’ve covered the concepts and theory behind view controller containers and the page view controller, it’s time to get our 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.

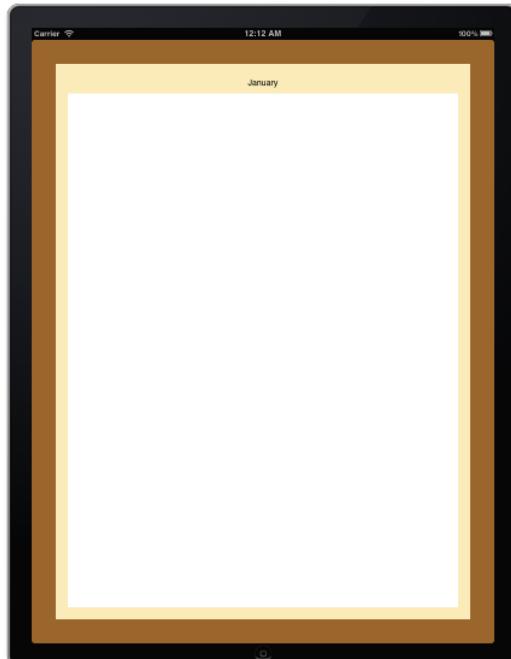


On the next screen name your project **MyPageView**, give it a company identifier in case you want to run your app on a device, make sure **Universal** is set for Device Family and select **Use Automatic Reference Counting**.



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 for us 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 we will do.

In the Project Navigator we can see that Xcode has provided us with several classes as well as two storyboard files for our interface, one for iPhone and one for iPad.

Let's go over each of the files and see what's been setup for us. 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 us.

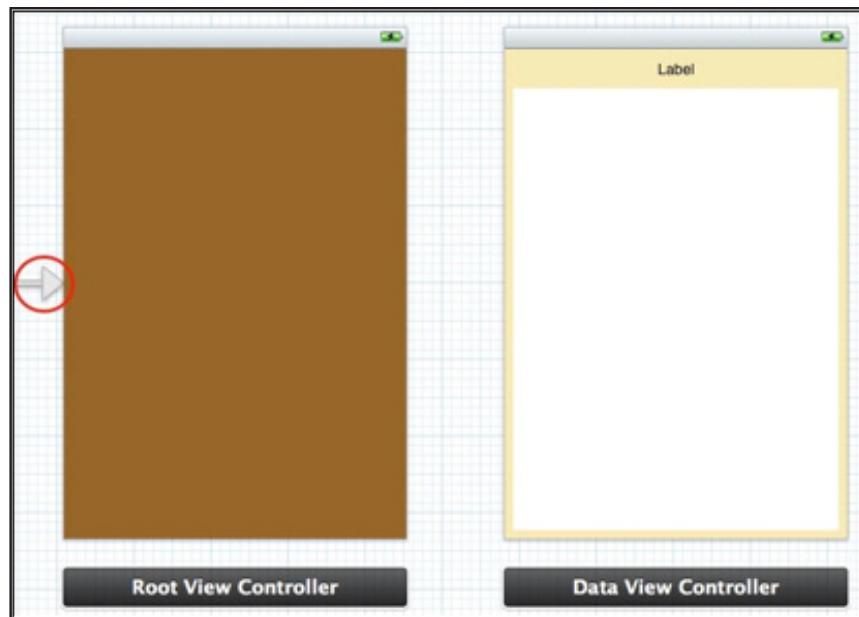
Jump over to the **AppDelegate.m** file and notice the *applicationDidFinishLaunchingWithOptions:* method no longer has all the setup of adding a subview and calling *makeKeyAndVisible*. As you may have read in the Beginning and Intermediate Storyboards chapters, what Apple is trying to do with storyboards 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. Let's start with the iPhone storyboard - open up **MainStoryboard_iPhone.storyboard** and let's see what we've got!

The first thing you'll notice there are 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 our case it's Root View Controller (hence the name).

Here is an image of our 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 we have storyboards doesn't mean all view controllers have to be connected in some way.

Select the Data View Controller, open up the Attributes Inspector and notice that it has "DataViewController" set as the Identifier. With storyboards we can give scenes a unique identifier so we may programmatically instantiate them, just like we would have 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 see this in action let's open **RootViewController.h** and see what we've got:

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

@interface RootViewController : UIViewController
<UIPageViewControllerDelegate>

@property (strong, nonatomic) UIPageViewController
*pageViewController;

@end
```

As you can see we have a `UIPageViewController` property with the strong attribute, the equivalent of retain using ARC, and we add the `UIPageViewControllerDelegate` 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, synthesize it and the runtime will take care of creating the property's instance variable for you.

You may also declare a different variable name for the instance variable and the property respectively by synthesizing as follows:

```
@synthesize pageViewController = _pageViewController;
```

This will result in a property named pageViewController and an instance variable named _pageViewController.

Head over to **RootViewController.m** next to continue our investigations. First we have some imports necessary for the classes we are going to use and communicate with, next up we have an unnamed category for our 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 us by the template which 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 we 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:  
            UIPageViewControllerTransitionStylePageCurl  
        navigationOrientation:  
            UIPageViewControllerNavigationOrientationHorizontal  
        options:nil];
```



```
self.pageViewController.delegate = self;

// 3
DataViewController *startingViewController =
    [self.modelController viewControllerAtIndex:0
        storyboard:self.storyboard];
NSArray *viewControllers =
    [NSArray arrayWithObject: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 = CGRectMakeInset(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. We 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 we can pass in an NSDictionary indicating where the spine is located, we'll take a look at that bit later when we create our own page view controller from the ground up.

Finally we 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, which we will cover in a bit, for now just be aware that this is creating a new data view controller for us.

After this we 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 we 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 we add the page view controller's view as a subview of the root view controller otherwise we would not see our page view controller on screen, just the root view controller's view.

5. Because we are adding the page view controller as a child of root view controller, we must set it's frame so that we 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 we 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, we add the page view controller's gesture recognizers to the root view controller's gesture recognizers.

The explanation for this is very simple, if we don't do this step then we will see our page view controller once our app loads but no interaction will occur when we tap or drag on the screen. This happens because we 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 we have covered a lot of things here, especially some crucial inner workings of the page view controller.

Moving on we can see that the `shouldRotateToInterfaceOrientation` method is allowing rotation on all orientations except portrait upside down for iPhone, and all orientations for iPad.

We are almost done with the `RootViewController` implementation, all we need to cover are the `UIPageViewController` Delegate methods, yay!

You might see a method commented out that begins with `didFinishAnimation...`. This method gets called after a gesture transition occurs, meaning when the user taps or drags across the screen and we 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 should you require.

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 how I showed you 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.

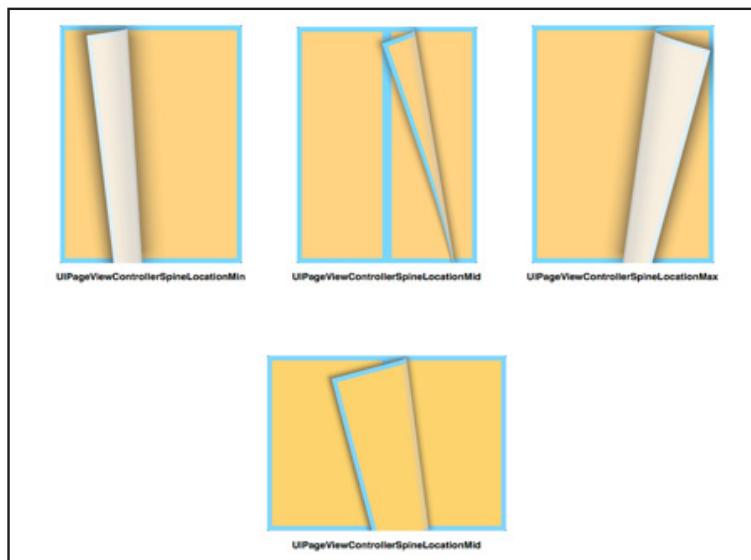


When navigating horizontally, `UIPageViewControllerSpineLocationMin` will position the spine on the left, `UIPageViewControllerSpineLocationMid` in the middle and `UIPageViewControllerSpineLocationMax` on the right.

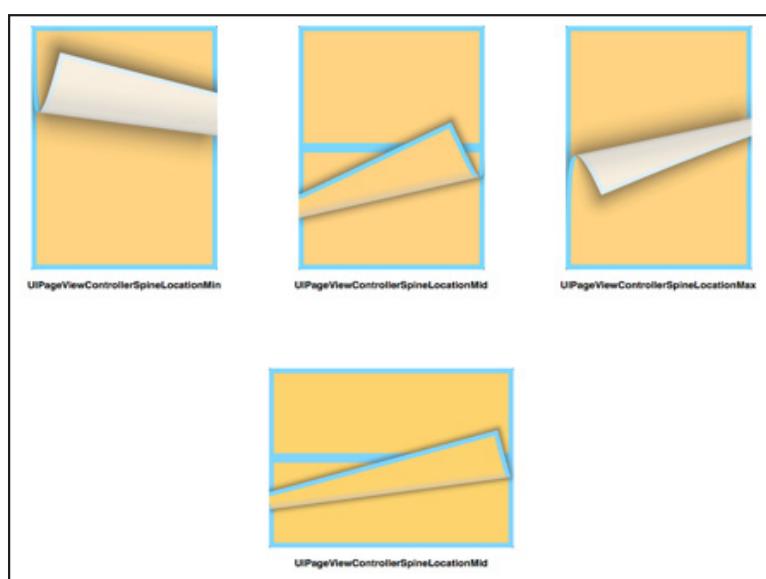
Similarly when navigating vertically `UIPageViewControllerSpineLocationMin` will position the spine at the top, `UIPageViewControllerSpineLocationMid` in the middle and `UIPageViewControllerSpineLocationMax` at the bottom.

Here are some images to show how this works:

Horizontal Navigation:



Vertical Navigation:



We 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 controller's 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 we want content to appear on the backside of a page or not.

For the vertical orientation (like in iBooks for example) we don't want it to be double sided in order to get the semi-transparent effect you see when flipping a page. When we are in landscape orientation with two view controllers being shown simultaneously then we 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).

Note: In case you are reading this book when an update to the iOS SDK comes out and the templates are changed a bit, we have included the source code for the page view controller template so you can follow along without getting lost.

Woohoo! We are done with the RootViewController class, let's move on to the ModelController class to see how we are displaying some data using a page view controller.

Open up **ModelController.h** and let's see what we 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 we will cover in a bit), next you will notice that ModelController implements the



UIPageViewController Data Source 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 we pass into the method.

Let's go over to the implementation file to see how these two methods work as well as which methods we 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 we are getting the name of the twelve months of the year and saving each into our pageData array.

Let's move 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 & 5 on Storyboarding).

```
- (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 objectAtIndex:index];
    return dataViewController;
}
```

The if statement is included to check for a valid index for our view controller, in case we can't create one we return nil. Outside of the if statement a DataViewController object is created by calling instantiateViewControllerWithIdentifier: on our local storyboard property.

Because the model controller class doesn't know about our storyboard, we pass it in from the root view controller and use it to create our data view controller instance.



`indexOfViewController:` simply receives a `DataViewController` object and looks up the index of it's local `dataObject`, that way we know where we should navigate to, and where our pages begin and end.

Next up are the `UIPageViewController` Data Source methods, there are two of them; `viewControllerBeforeViewController:` and `viewControllerAfterViewController:`.

These methods receive a `UIViewController` object and we use the `indexOfViewController:` method to know where in our application we are and what data we 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 which is unavailable to that particular view controller.

Believe it or not that's all we 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` which will connect with the label on our 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** implementation file and look at what's inside.

The properties are synthesized, the interface orientation is set depending on which device the app is running on and the `dataLabel` is set to the `dataObject`'s description (which returns the name of the month).

Where To Go From Here?

That's it! We've now 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, we'll show you how to use a `UIPageViewController` in your project from scratch, and use it to build a fun photo viewing app!



17

Intermediate UIPageViewController

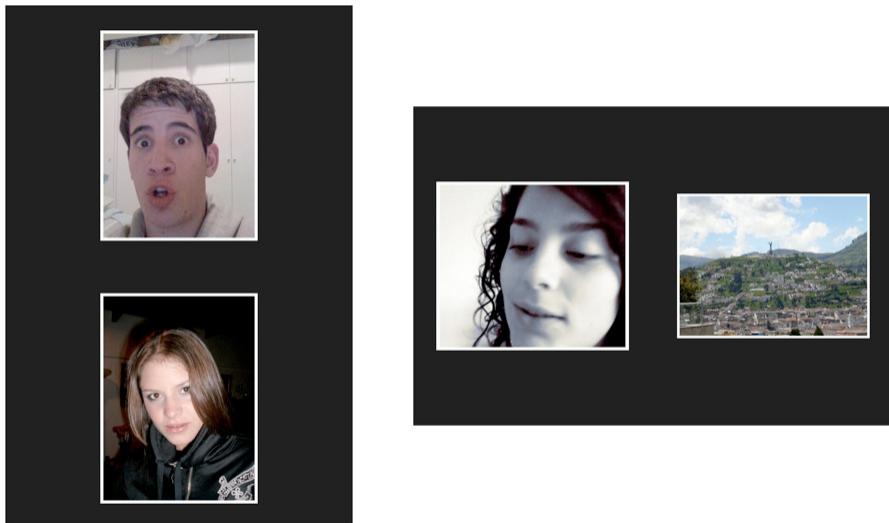
By Felipe Laso Marsetti

In this chapter, we are going to recreate a `UIPageViewController` from scratch. 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 we're recreating the template, not only will we understand how everything works while writing cleaner code, but we 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.

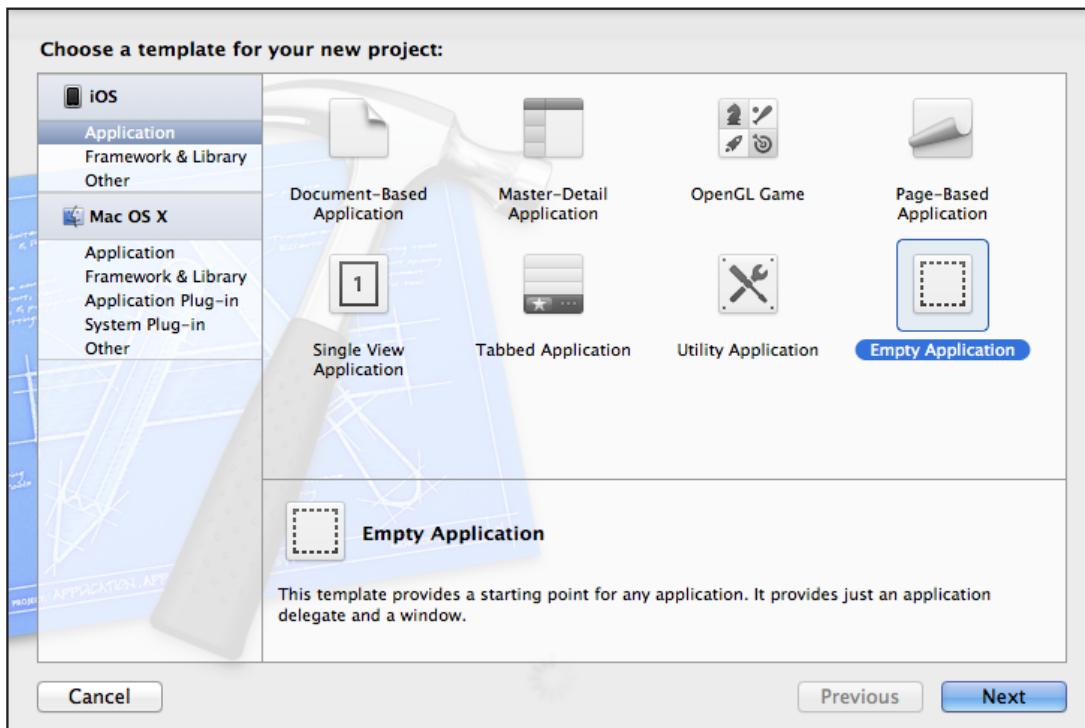
Here's a preview of the app we're going to build:



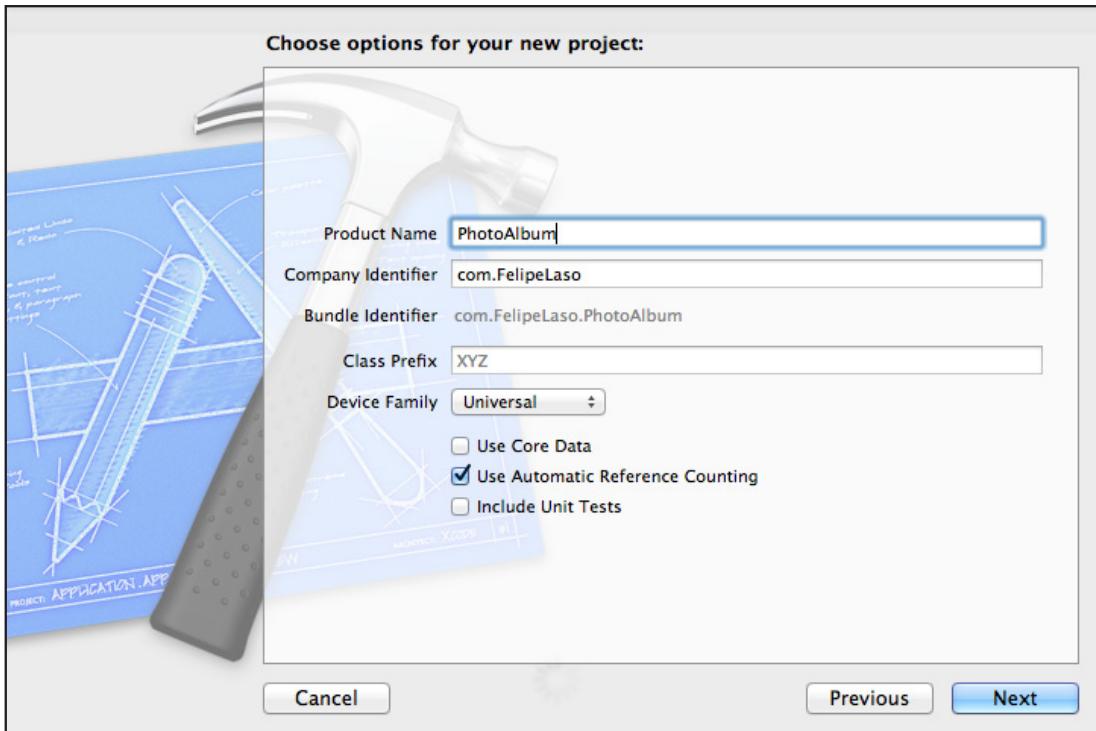
Getting Started

Right, the time has come for us to get our 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 let's 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 Device Family 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.

We now have a blank project with just an application delegate that we can customize and start from scratch. If you see the settings for our project everything is fine for now, we have the orientations, iOS version and everything else setup for us.

Note: Notice in the project settings we have an empty field where we can select either a storyboard or main interface file for iPhone and iPad, for now let's leave those blank as we will be creating our storyboards in a bit.

The first thing we are going to do is create a view controller that will be the root of our 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\UIViewController subclass**. 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, but we don't need for our 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;
```

Then, back in **RootViewController.m**, synthesize the property by adding the following line just after the @implementation declaration:

```
@synthesize pageViewController = _pageViewController;
```

Finally let's cleanup our property when the view is unloaded by making a small change in the viewDidLoad method as follows:

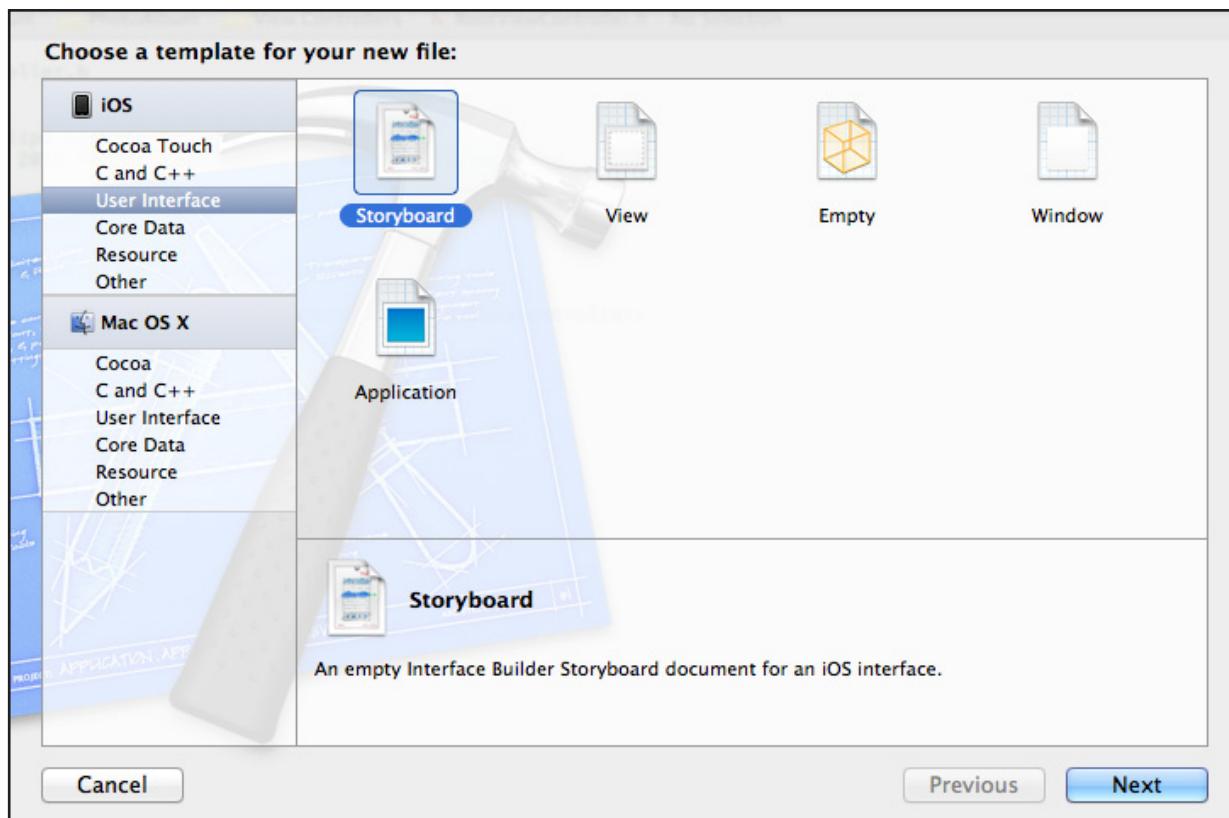
```
- (void)viewDidLoad  
{  
    self.pageViewController = nil;  
    [super viewDidLoad];  
}
```

Awesome! we have a page view controller property that we can now use throughout our project.

Before moving on let's create our storyboards so that we can instantiate a root view controller and have it act as the container (parent) of our UIPageViewController. We will have one storyboard for iPad and one for iPhone/iPod Touch devices.

Control-click on your 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.





For Device Family let's start with **iPad** 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.

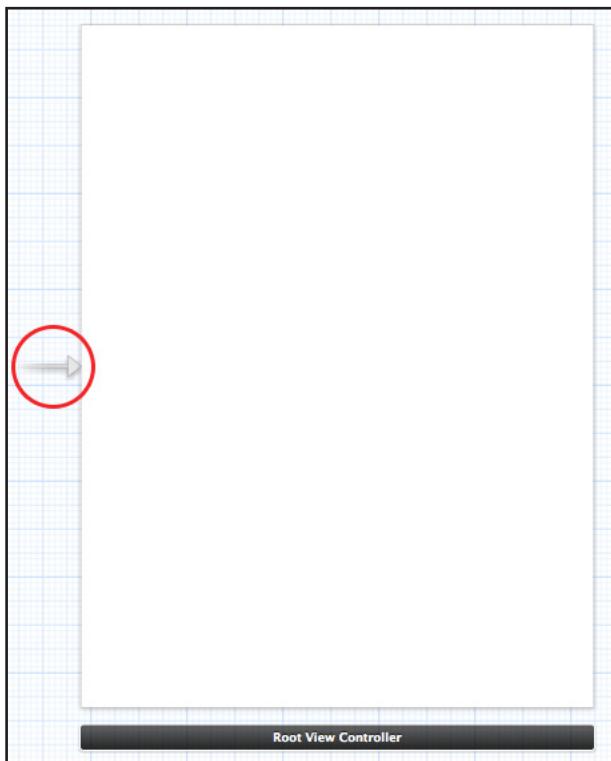
Now we need to repeat the same steps for the iPhone, so right-click your Photo-Album folder, select New File, choose the **iOS\User Interface\Storyboard** template, and click Next. Under Device Family make sure you choose iPhone, click Next and name your file **Storyboard_iPhone.storyboard**. Finally click Create and you should see your iPhone storyboard in the Project Navigator.

Great! We now have both our storyboards ready to be set up. Because both the iPhone and iPad versions of our interface will have the same layout and contents we'll have to repeat the same process for both. I know it's a bit tedious but luckily we only have to do this once and it will help us 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 our view controller, this indicates it's the initial view controller.



Select your newly created view controller and open the Attributes Inspector (the keyboard shortcut for this is OPTION + COMMAND + 4). In the Identifier field write RootViewController, and set the Status Bar to None under the Simulated Metrics section. After you've done this, navigate to the Identity Inspector (OPTION + COMMAND + 3) and for the class write RootViewController.

For our case we 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 our photo album), set the simulated status bar to None, and set the identifier to AlbumPageViewController.

If you have read chapters 4 & 5 on Storyboarding, you will notice we don't have a segue or relationship between the view controllers. Don't worry about this because

we can programmatically instantiate them using the identifier we just set. Adding the view controllers into a storyboard like this just helps us write less code and design our interface within a single file.

We will not be setting the class for our new view controller because we haven't created one yet.

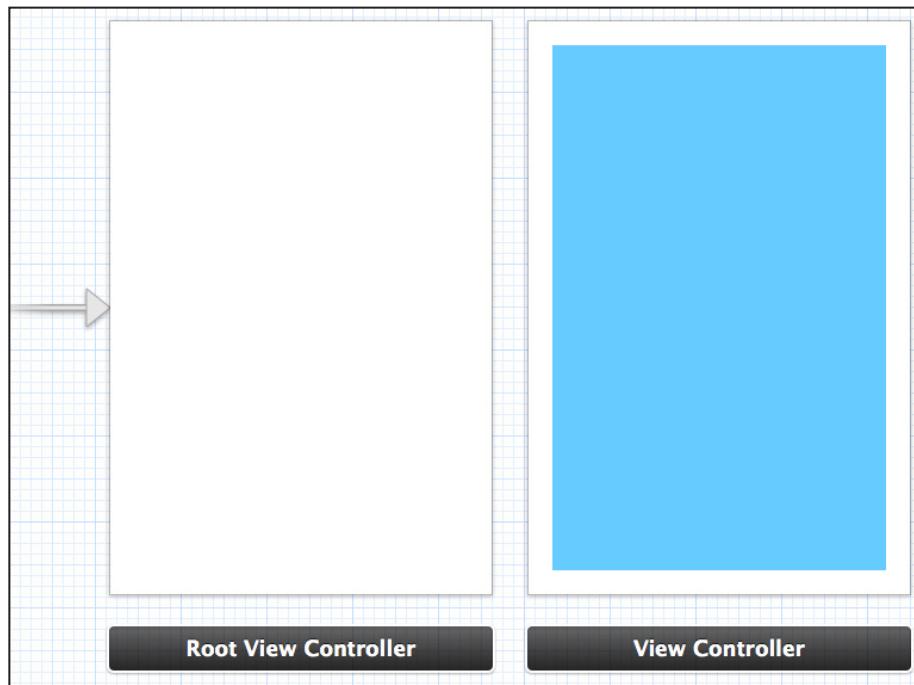
Drag a new view inside the newly created view controller and size it so that it fits within the blue interface guidelines, in the Attributes Inspector change the background color to something other than white (so we may differentiate it with the view controller's view).

Don't worry if this doesn't make much sense right now, I'll use this 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 Identifier 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 Identifier to AlbumPageViewController, add a view to the view controller you just created and change its background color.

Awesome, your storyboard files should now look like the following:

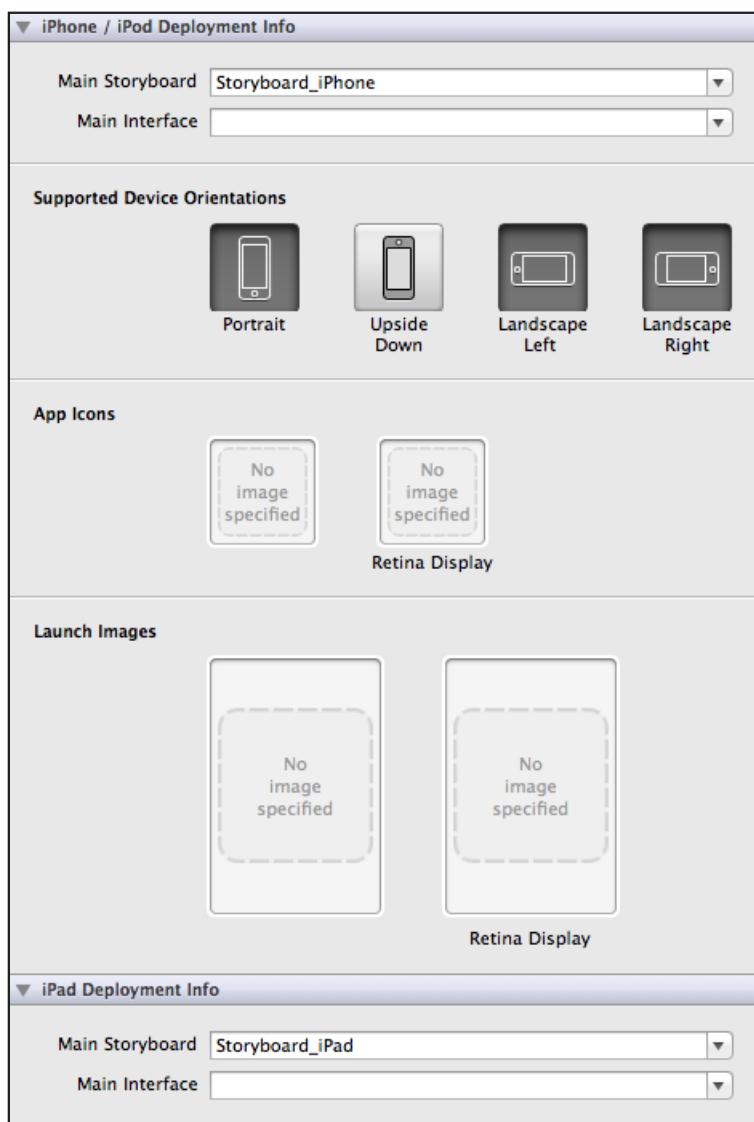


Don't worry if the colors aren't the same, the important thing is that your view controllers are setup correctly with the appropriate identifiers.

If you recall, earlier when we first created our project we didn't have a storyboard or interface file setup in the project settings. Let's go ahead set up our project now to use the storyboards we 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! Our storyboard files are now connected to our project and ready to be used. There are just a few small things we have to do before building and running our project.

In the Project Navigator find and open the **PhotoAlbum-Info.plist** file, add a new property for **Status Bar Is Initially Hidden** and set it to **YES**. This will be the actual line setting that hides it, the status bar we removed from our storyboard files was just a place holder for design purposes.

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 we are actually using our 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, our project is now set up and ready to use a UIPageViewController.

Open **RootViewController.m**, delete the commented out `loadView` method and move on to `viewDidLoad` where we will create our pageViewController. Inside `viewDidLoad` let's add some code:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    NSDictionary *pageViewOptions = [NSDictionary  
        dictionaryWithObjectsAndKeys:  
            UIPageViewControllerOptionSpineLocationKey,  
            [NSNumber numberWithInt:  
                UIPageViewControllerSpineLocationMin], nil];  
  
    self.pageViewController = [[UIPageViewController alloc]  
        initWithTransitionStyle:  
            UIPageViewControllerTransitionStylePageCurl  
        navigationOrientation:  
            UIPageViewControllerNavigationOrientationHorizontal  
        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, but it's still missing a few optional elements here and there but it should be familiar since we covered a lot of this back in the theory section.

Remember I told you we would 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 we can only put objects inside a dictionary, we must wrap it within an NSNumber.

Next up we create our page view controller and save that instance in our pageViewController property. In the initializer we pass in the transition style (which in our case is the page curl), a navigation orientation of forward (because we read our books from left to right) and the options dictionary (or nil in case you don't want to set any).

After this we need to pass the initial view controllers for our page view controller, in order to do this we create a UIViewController instance using the instantiateViewControllerWithIdentifier method provided to us by the UIStoryboard class. Notice how we used the AlbumPageViewController identifier which we setup in Interface Builder.

This line might seem new and interesting:

```
self.storyboard
```

This property is available to all UIViewController instances, that is 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 we create our view controller we proceed to set the view controllers for our page view controller. Inside this method we pass in an array of view controllers, the navigation direction, whether we want this action to be animated and a completion block.



Because we are creating the page view controller right after the app launches we don't want any animation to occur, when the user switches between a spine locations after changing the device orientation (meaning that you display two view controllers at a time) animating the transition might be a good idea.

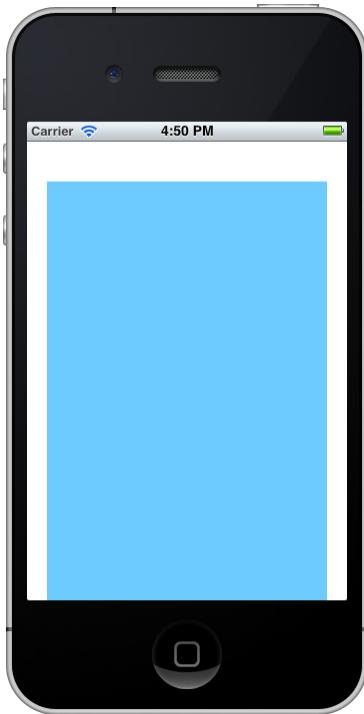
Finally we add the pageViewController as a child of the root view controller and add the page view controller's view as a subview of the root view controller's view (phew, we used the word view a lot in that paragraph!).

Go ahead and run your project now and you'll see something like this:



Yay! This is exactly what we wanted, now let's 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, our view controller is not centered properly! Fear not, with just one line we 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.

After adding this line of code we need to tell our application that we want the status bar to be hidden, this can be done in the PhotoAlbum-Info.plist file. The property we are looking to add is "Status Bar Is Initially Hidden" and then set it to YES, as shown below:

Main storyboard file base name (iPad)	String	Storyboard_iPad
Status bar is initially hidden	Boolean	YES
► Supported interface orientations	Array	(3 items)

Let's move on!

Expanding our App and the UIPageViewController

We want to create a custom subclass of UIViewController so we can handle laying out our album pages.

Control-click the PhotoAlbum folder, select New File, select the **iOS\Cocoa Touch\UIViewController subclass** 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 we won't be needing this.

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 the **Storyboard_iPhone.storyboard**.

Now go back to **RootViewController.m**. We need to change the regular UIViewController instance that we 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 our `albumPageViewController` variable an instance of UIViewController, change it to the following:

```
AlbumPageViewController *albumPageViewController = [self.storyboard  
    instantiateViewControllerWithIdentifier:@"AlbumPageViewController"];
```

Build and run the app so we can make sure that everything works fine, we get the same result as before but we are now using our AlbumPageViewController class as the page view controller's child view controllers.

If you notice at the moment there is no interaction going on within our app, we can't swipe to flip pages or even rotate the device. Let's add that functionality now!

We 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 let's implement the delegate and data source methods.

We'll begin with the only required delegate method, so copy this within the root-ViewController 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 our implementation file:

```
- (UIPageViewControllerSpineLocation)pageViewController:  
    (UIPageViewController *)pageViewController  
    spineLocationForInterfaceOrientation:  
        (UIInterfaceOrientation)orientation  
{  
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad &&  
        UIInterfaceOrientationIsLandscape(orientation))  
    {  
        AlbumPageViewController *pageOne =  
            [self.storyboard instantiateViewControllerWithIdentifier:  
                @"AlbumPageViewController"];  
        AlbumPageViewController *pageTwo =  
            [self.storyboard instantiateViewControllerWithIdentifier:  
                @"AlbumPageViewController"];  
  
        NSArray *viewControllers =  
            [NSArray arrayWithObjects:pageOne, pageTwo, nil];  
  
        [self.pageViewController  
            setViewControllers:viewControllers  
            direction:UIPageViewControllerNavigationDirectionForward  
            animated:YES completion:nil];  
  
        return UIPageViewControllerSpineLocationMid;  
    }  
  
    return UIPageViewControllerSpineLocationMin;  
}
```

All this does is return the mid spine location if we are in landscape mode on an iPad, otherwise we return the min location. Because the iPhone's screen is much smaller than the iPad we don't want to divide our page view controller into two pages, so we will always have the spine on the left side.

If we are on the iPad and are switching to landscape orientation we will display two view controllers (two pages side by side) so we need to set the page view controller's view controllers again (because it now needs two instead of just one).



We instantiate two `AlbumPageViewController` objects, store them in an array and call the method provided to us by `UIPageViewController`, we keep the navigation direction to forward and animate the transition.

Right now if we run this nothing will change, take a look at the `shouldAutorotateToInterfaceOrientation:` method and you'll see we 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 we support these orientations within it as well.

Go ahead and run your project now, you should see it rotates properly, another yay for us!

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 our `AlbumPageViewController` class. This isn't the final implementation of these methods but for now we will use them to test that our page view controller is properly setup with the correct navigation and spine locations.

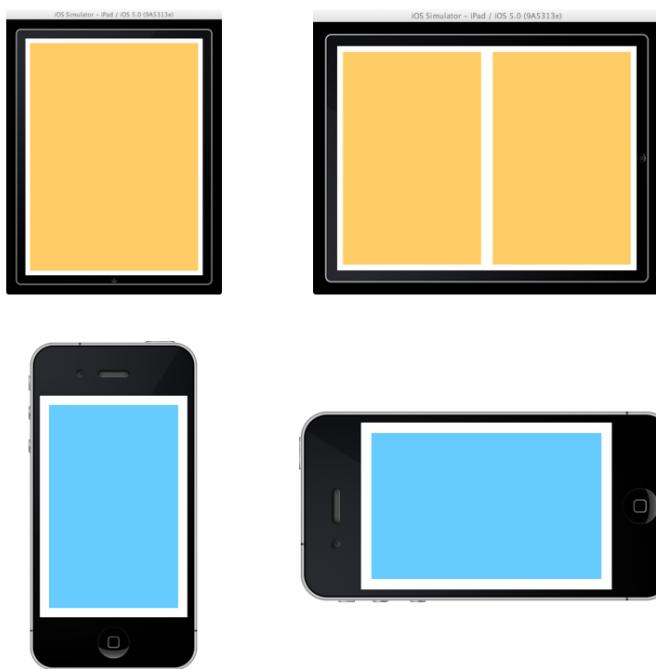
Before we go ahead and run our project we have to add the page view controller's gesture recognizers to the root view controller's gesture recognizers. Add this line inside the `viewDidLoad` method right after adding the page view controller's view as a subview:

```
self.view.gestureRecognizers =  
    self.pageViewController.view.gestureRecognizers;
```

Finally we need to set ourselves as the delegate and data source of the page view controller. Also inside `viewDidLoad` add this line right after we 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:



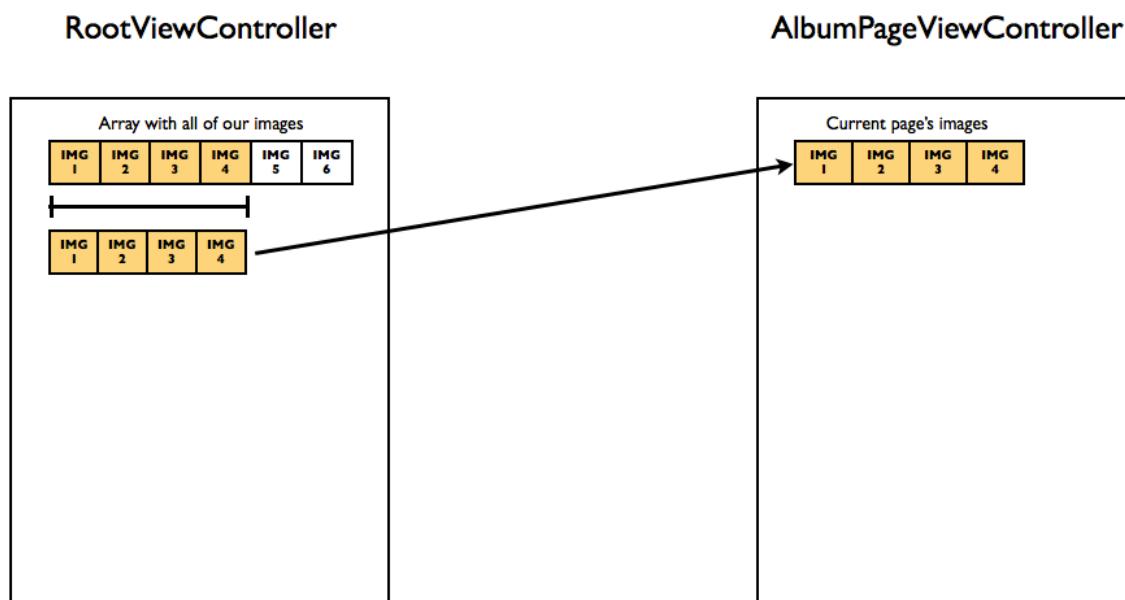
Just as we wanted we have the spine located on the left when we are using the iPhone, regardless of its orientation, and on the iPad we have a spine on the left when in portrait orientation or in the middle when in landscape orientation.

Go ahead and try swiping or tapping to flip to the next or previous page. Right now we aren't using any actual logic to drive our pages so we can infinitely flip through them.

We should work on that soon, but before doing so we need an actual model to drive the information being shown on our pages and so we can set a limit to the number of pages we display.

Building the Model

Here is an illustration to show you how things are going to work within our app:



Don't laugh, I know it's simple! Basically our root view controller will contain an array with all of the images for our application, whenever the user rotates the device or flips to another page we 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 we will show 4 pictures at a time, on the iPhone we will only show 2.

We will have to create the logic to determine how many images need to be passed along and which ones to choose from.

Let's add an NSArray property inside **RootViewController.h**:

```
@property (strong, nonatomic) NSArray *picturesArray;
```



Then in **RootViewController.m** synthesize it just below the pageViewController property.

```
@synthesize picturesArray = _picturesArray;
```

Make sure you also set it to nil inside the viewDidLoad method:

```
self.picturesArray = nil;
```

Let's do the exact same thing for the AlbumPageViewController class, inside **AlbumPageViewController.h** add the same property and then inside **AlbumPageViewController.m** synthesize it and set it to nil in viewDidLoad.

The AlbumPageViewController class will also have an integer to store its index in the view hierarchy, this allows us to control the hierarchy without having to write the methods that the template uses.

We are just doing things a bit different so we 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;
```

We are going to use an unsigned integer because our indexes will always be positive. Synthesize this property with the following line:

```
@synthesize index = _index;
```

Leave viewDidLoad as it is, since NSUInteger isn't an object we don't need to set it to nil or perform any memory cleanup.

Great! We 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.

Next go to **RootViewController.m** and update your 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 objectAtIndex:0];
NSArray *viewControllers = nil;

NSUInteger indexOfCurrentViewController =
    currentViewController.index;

// 3
if (indexOfCurrentViewController == 0 ||
    indexOfCurrentViewController % 2 == 0)
{
    UIViewController *nextViewController =
        [self pageViewController:self.pageViewController
            viewControllerAfterViewController:
                currentViewController];
    viewControllers =
        [NSArray arrayWithObjects:currentViewController,
            nextViewController, nil];
}
else
{
    UIViewController *previousViewController = [self
        pageViewController:self.pageViewController
        viewControllerBeforeViewController:
            currentViewController];
    viewControllers = [NSArray arrayWithObjects:
        previousViewController, currentViewController, nil];
}

// 4
[self.pageViewController setViewControllers:viewControllers
    direction:UIPageViewControllerNavigationDirectionForward
    animated:YES completion:NULL];

return UIPageViewControllerSpineLocationMid;
}

// 5
AlbumPageViewController *currentViewController = [self.
    pageViewController.viewControllers objectAtIndex:0];

NSArray *viewControllers = [NSArray arrayWithObject:
    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 we just check to see if we are on an iPad and in landscape orientation, if we 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 we just returned a new `AlbumPageViewController`. We are updating this because we want to get the index of the currently displayed `AlbumPageViewController` and do some stuff accordingly.

In order to do this we can use the `viewControllers` property of the page view controller which returns an array of the view controllers being displayed, we get the view controller at index 0 (the first one) and assign it to an `AlbumPageViewController` variable.

After this we create an `NSArray` named `viewControllers` that we will use to set the view controllers of our page view controller, finally we just read the index of the album page view controller we just stored and save it into a variable of type `NSUInteger`.

3. Inside this if statement we check to see whether the index of the album page view controller we are currently displaying is 0 or an even number, If so then we use the page view controller's data source method to return the album page view controller that goes after the one currently displayed. We store both the current and next album page view controllers in the `NSArray` variable we declared before.

If the statement isn't true then it means the index of the current album page view controller is an odd number, we do something very similar as before but we ask the data source to return the album page view controller before the one being currently displayed.

4. After properly configuring our view controllers we set them on the page view controller, once again we use the forward navigation with an animated transition and we return the mid spine location (which will result in two pages being displayed, like an open book).
5. If we are not on an iPad in landscape orientation then we 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 we are on an iPhone we will return the same view controller, if we are on an iPad (which means the user might have been using the app in landscape) we get the first view controller being displayed by our 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 our album so the first thing we do is cast and store the view controller being passed into the method (the view controller being currently displayed).

We have a simple if statement to check whether this is the first view controller, if it is we return nil (because there are no pages we can return before the first one) otherwise we 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 we are on the first page, we now check to see if we are on the last one. For testing purposes we are going to use the value 9 inside the if statement which will give us 10 pages for our album, when we setup our pictures we'll calculate this at runtime (depending on the amount of pictures we have).

If we're on the last page we return nil (because there are no pages after the last one) otherwise we instantiate and return a new `AlbumPageViewController` with the index that follows the one currently displayed.

That's almost all we need to do to limit the number of pages, the last thing we need to do is set the index of the album page view controller when our app launches. In the `viewDidLoad` method add this line right after we 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 we 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 we have an odd number of pages then the last one will not show when using an iPad in landscape mode. So if we have 9 pages then we will only show pages 7 and 8 but never 9 (we need pass two view controllers to the page view controller). Keep this in mind if you use this code in your projects, otherwise write something to suit your needs.

How cool is the page view controller? We have written very little code and achieved something that would have required serious customization in previous versions of iOS.

Our photo album is ready to show some photos, which means we need to add them to our project. We are going to store the photos in our 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 we're going to use for our 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 we have an array of strings with the names of our photos. We are going to load this array from the property list file and save it into our root view controller's picturesArray.

In order to properly size and scale the images we are going to determine the width and height at runtime and position them on our 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 objectForKey:@"PhotosArray"];
```

Now we have our array loaded with the picture names, great, but we need to pass them into our album page view controller so they can be shown on each page. In order to do this we have to make a few changes to the page view controller's data source methods as well as our 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 we 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 we begin by getting the previous view controller and checking if it's the first page (index 0), if it is then we don't pass a view controller before this one so we return nil. If we can show a page before the one we came from then we instantiate a new album page view controller object and decrease its index from the current one.

Simple stuff right? Now we get into the picture calculations, for this we declare an NSRange to determine which picture names we are going to get from the array and pass them into the album page view controller.

We have an if statement depending on whether we are on iPhone or iPad, on iPad we will display 4 pictures per page but on iPhone we will only show 2 because of the limited screen size.

Whether we are on iPhone or on iPad we use the album page view controller's index to determine at what index we will begin looking for the names of our pictures, once we do that we get either 4 strings for iPad or 2 for iPhone and pass them into the new album page view controller object.

Inside here we do make sure we don't get any invalid indexes in the array or try to access objects which 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;
```



```
NSInteger pageCount;

if ([[UIDevice currentDevice] userInterfaceIdiom] ==
    UIUserInterfaceIdiomPad)
{
    pageCount = (NSInteger)
        ceilf(self.picturesArray.count * 0.25);
}
else
{
    pageCount = (NSInteger)
        ceilf(self.picturesArray.count * 0.5);
}

pageCount--;

if (previousViewController.index == pageCount)
{
    return nil;
}

AlbumPageViewController *albumPageViewController =
    [self.storyboard instantiateViewControllerWithIdentifier:
        @"AlbumPageViewController"];
albumPageViewController.index = previousViewController.index + 1;

NSRange picturesRange;

if ([[UIDevice currentDevice] userInterfaceIdiom] ==
    UIUserInterfaceIdiomPad)
{
    NSInteger 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
{
    NSInteger 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 we check to see if we are on the last page already. Then we proceed to use the index of the new page view controller to determine which range of picture names we need to get from our array.

Before we leave the RootViewController we need to add some code to the viewDidLoad method. Both page view controller data source methods are called when we flip to a new page, but when our application first launches we 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;
```



```
if (self.picturesArray.count >= 2)
{
    picturesRange.length = 2;
}
else
{
    picturesRange.length = self.picturesArray.count;
}

albumPageViewController.picturesArray =
    [self.picturesArray subarrayWithRange:picturesRange];
}
```

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, we are getting the NSStrings with the names of the pictures properly passed into the album page view controllers, now we need to go to the album page view controller and properly position the pictures on the page!

Before we do that go back to your Storyboard_iPhone and Storyboard_iPad files and delete the colored UIViews we placed on our album page view controller. I also changed the view's background color to black and added 1 pixel thick UIViews as borders for our page.

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 we have our storyboards clean and customized, let's take a look at the final portion of code we 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 we 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
```

We create an NSMutableArray property named pictureViews which will store our



UIImageView objects, this will save us from having to re load images and re create any objects just because the interface rotation changed.

Now in the implementation make sure you synthesize it as well:

```
@synthesize pictureViews = _pictureViews;
```

Don't forget to nil out your properties in viewDidUnload, being clean and tidy with our code and memory is always a good habit (ARC certainly helps us with this now!)

```
- (void)viewDidUnload
{
    self.picturesArray = nil;
    self.pictureViews = nil;
    [super viewDidUnload];
}
```

After you synthesize your properties add this private method to your 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 &&
            InterfaceOrientationIsLandscape(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];
        }
    }
}
```



```
{  
    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 we want the pictures 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 we check whether we wanted the transition to be animated and if so, we 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 we determine the frame depending on the device we are on and use that to setup the pictures, then we check whether there are any pictures to show or not as we don't want to be accessing empty arrays or invalid indexes.

If we have just one picture we call on another private method (which we will see in a minute) that handles setting up the pictures, we don't have to divide the screen or anything for one picture, just scale and center it.

When there are two pictures we check whether the device is an iPhone in landscape or either an iPhone or iPad in portrait so we can divide the view horizontally or vertically respectively. Once that's done we call our `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 our `setPictureAtIndex:` method to handle positioning them.

Nothing too complicated, just some simple math to determine how many pictures we have, what orientation and device we are using the app on and where to position our pictures.

Now paste this method right above the `layoutPicturesAnimated:` method:

```
- (void)setPictureAtIndex:(NSUInteger)index inFrame:  
    (CGRect)frameForPicture  
{  
    UIImageView *picture = [self.pictureViews objectAtIndex: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 which contains the frame where the picture is to be positioned. We first get a UIImageView with the corresponding image and create a CGFloat variable to determine the ratio between the width and height of our picture.

Some calculations are done depending on whether the image width is larger than its height or vice versa. After that's determined we get the scale and proceed to make the image 20% smaller than the frame which contains it.

Finally the image gets centered and added as a subview of the album page view controller's view.

We 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];
    }
}

```

We create a UIImageView for every name of a picture we have in our picturesArray, we make sure to initialize our pictureViews array in case we haven't already.

And finally these two methods:

```

- (void)viewDidAppear:(BOOL)animated

```



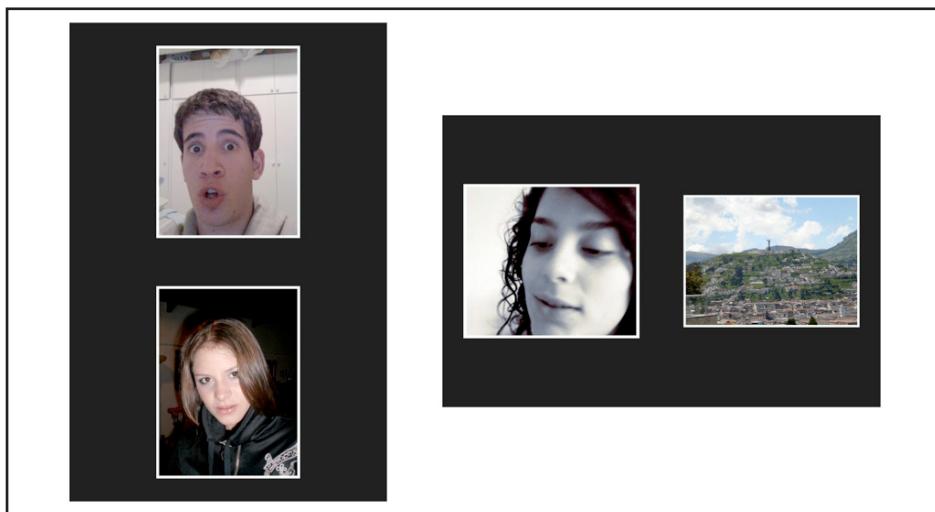
```
{  
    [super viewWillAppear:animated];  
  
    [self layoutPicturesAnimated:NO withDuration:0  
        forInterfaceOrientation:self.interfaceOrientation];  
}  
  
- (void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)  
    toInterfaceOrientation duration:(NSTimeInterval)duration  
{  
    [self layoutPicturesAnimated:YES withDuration:duration  
        forInterfaceOrientation:toInterfaceOrientation];  
}
```

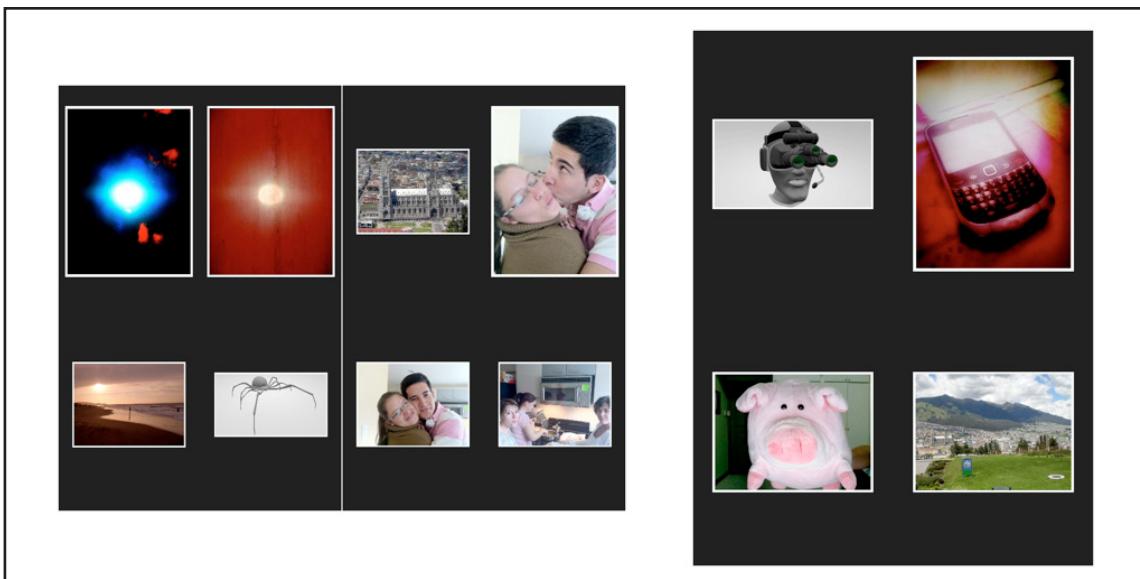
When our view first appears we want to position the pictures immediately, so we pass NO into the `layoutPicturesAnimated:` method. We also pass the current interface orientation so we construct the frame for the pictures to be placed in as well as a duration, which in our case is 0 because we want things to load immediately when we first load.

On the `willRotateToInterfaceOrientation:` method we layout our pictures again except this time we animate the transition, we use the new orientation we are going to rotate to and the same duration as the transition.

And with that...WE ARE DONE!

A big yay for us as we 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!

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 than it would have been rolling these features on your own. Plus, it makes games more social, increasing the viral and social elements of your games.

In iOS5, Game Center has 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 other types of casual turn based games. You can take a turn in your game, wait for your friend to take his turn, and then get notified when it's your turn again!

The new API is great to use, even in favor of other third party or custom built solutions, because of the built in social features included in Game Center. Players can send invitations from developer's games to others, even those who haven't installed the game, and the opportunity to download and install the game will be presented to the invited player.

That, combined with the large existing friend base of game center, the leaderboards and achievements in Game Center, is a great reason to consider using this new API!

In this tutorial we will build a simple UIKit based game called "Spinning Yarn", where you take turns writing a story together with a friend. We'll build the simple game scaffolding first, then we'll spend the bulk of the tutorial exploring the new Turn Based Gaming APIs.

Note that to get the most out of this tutorial, you will need two different devices for testing and two separate Game Center sandbox accounts. I know this is annoying for testing, but it's an unfortunate fact of life for Game Center development!



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 our game, we'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 we'll be covering in detail in this tutorial:

- **GKTurnBasedMatch:** Contains all the information and game state for a match. We'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.
- **GKTurnBasedMatchmakerViewController:** This will be our primary UI element for interacting with matches, switching between matches, and creating new matches. We'll use it as our command center.
- **GKTurnBasedEventHandlerDelegate:** We will make a helper class implement this protocol, so we can get notifications when the turn moves from player to player, when we are invited to a new match, or when the game ends.

Note that the Game Center app will automatically keep track of who we are playing with and other data about our game, so users can always switch to Game Center to see their game status.

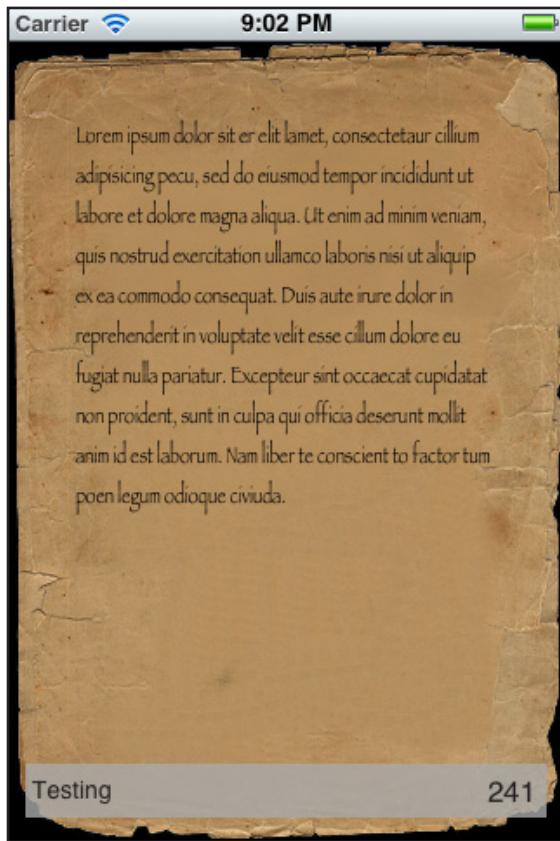
The SpinningYarn Game

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 we'll be making in this tutorial. I've made a starter project that contains the UI already made, so we 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!)

Feel free to look at **ViewController.xib** to see how the UI is arranged, and take a look at **ViewController.m** to see the code (it's very simple at this point). We'll convert the project into a fully functional game from here!

Setting up Game Center

Before you can start writing any Game Center code, you need to do two things:

1. Create and set an App ID
2. Register your app in iTunes Connect

Let's 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.

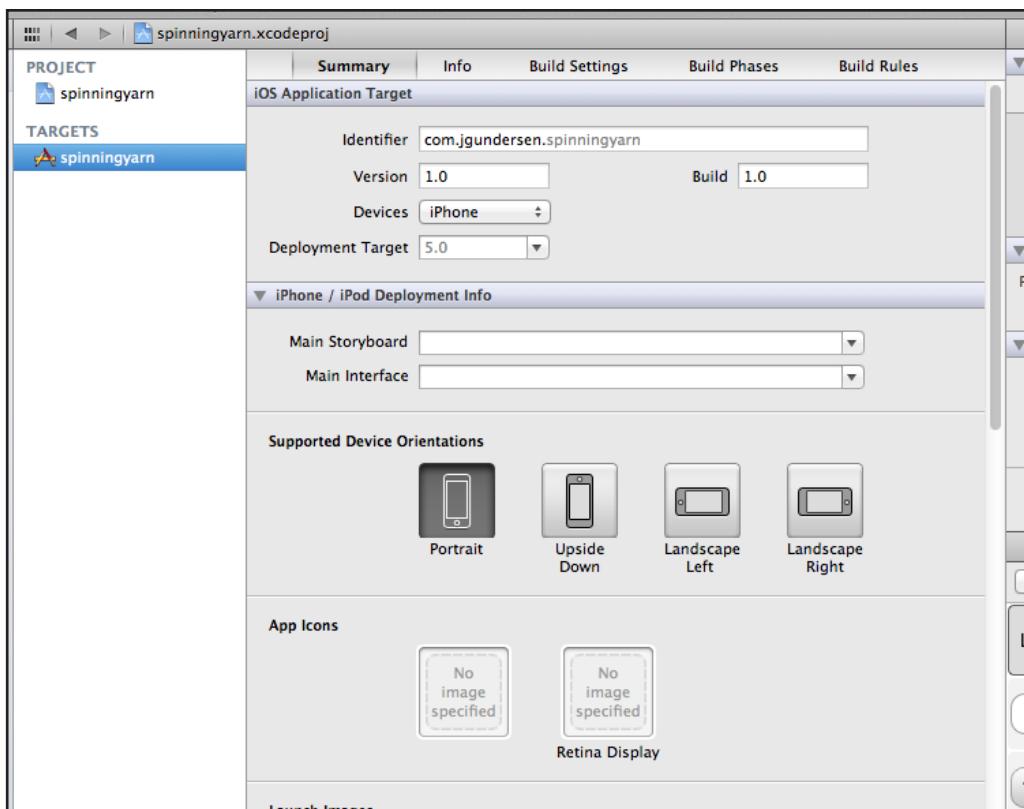
From there, select the App IDs tab, and create a new App ID for your app, similar to the following (except you'll be choosing different values):

The screenshot shows the 'Provisioning Portal : Third Rail, LLC' interface. The left sidebar has a menu with 'Home', 'Certificates', 'Devices', 'App IDs' (which is selected and highlighted in blue), 'Provisioning', and 'Distribution'. The main area has tabs 'Manage' and 'How To'. Below that is a section titled 'Create App ID' with a 'Description' field containing 'spinningyarn'. A note says 'You cannot use special characters as @, &, *, " in your description.' Below it is a 'Bundle Seed ID (App ID Prefix)' field with a dropdown set to 'Use Team ID'. A note says 'If you are creating a suite of applications that will share the same Keychain access, use the same bundle Seed ID for each of your application's App IDs.' Below that is a 'Bundle Identifier (App ID Suffix)' field with the value 'com.jgundersen.spinningyarn'. A note says 'Example: com.domainname.appname'.

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.

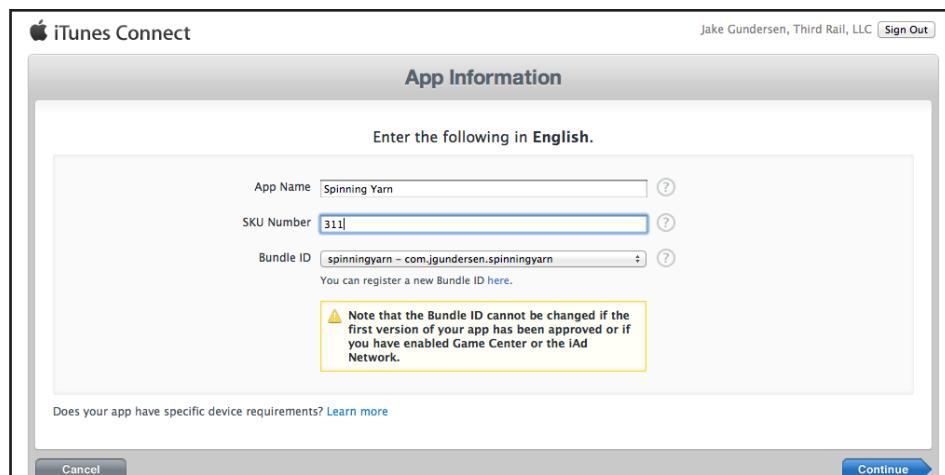
Register your app in iTunes Connect

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.

On the first screen, enter Spinning Yarn for the App Name, 311 for SKU Number, and select the bundle ID you created earlier, similar to the screenshot below:





Click continue, and follow the prompts to set up some basic information about your app. Note you will probably have to change the name, since I've already taken the Spinning Yarn name :]

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, and if all works well you should be in the "Prepare for Upload" stage and will see a screen like this:



Click the blue “Manage Game Center” button to the upper right, click the big blue “Enable” button, and click “Done”. That’s it – Game Center is enabled for your app, and you’re ready to write some code!

By the way – inside the “Manage Game Center” section, you might have noticed some options to set up Leaderboards or Achievements. We won’t be using those in this book, but if you ever need them that’s where they are!

Authenticate the Local Player: Strategy

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!” Otherwise, it will ask for the player’s username and password.

Authenticating the local user is easy – you just call `authenticateWithCompletionHandler`. You can optionally pass in a block of code that will be called once the user is authenticated.

But there’s a trick. There’s another way for the user to log in (or log out!). 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. You can find out about these by registering for an “authentication changed” notification.

So, our strategy to authenticate the player will be as follows:

- Create a singleton object to keep all the Game Center code in one spot.
- When the singleton object starts up, it will register for the “authentication changed” notification.
- The game will call a method on the singleton object to authenticate the user.
- 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, let’s try it out!



Authenticate the Local User: 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 gameCenterAvailable;
    BOOL userAuthenticated;
}

@property (assign, readonly) BOOL gameCenterAvailable;

+ (GCTurnBasedMatchHelper *)sharedInstance;
- (void)authenticateLocalUser;

@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.

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:`

```
@synthesize gameCenterAvailable;

#pragma mark Initialization

static GCTurnBasedMatchHelper *sharedHelper = nil;
+ (GCTurnBasedMatchHelper *) sharedInstance {
    if (!sharedHelper) {
        sharedHelper = [[GCTurnBasedMatchHelper alloc] init];
    }
    return sharedHelper;
}
```



This synthesizes the gameCenterAvailable property, then defines the method to create the singleton instance of this class.

Note there are many ways of writing singleton methods, but this is the simplest way when you don't have to worry about multiple threads trying to initialize the singleton at the same time.

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 that the turn-based gaming APIs we'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 we 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) {
            NSNotificationCenter *nc =
            [NSNotificationCenter defaultCenter];
            [nc addObserver:self
                selector:@selector(authenticationChanged)
                name:GKPlayerAuthenticationDidChangeNotificationName
                object:nil];
        }
    }
    return self;
}
```



```
- (void)authenticationChanged {

    if ([GKLocalPlayer localPlayer].isAuthenticated &&
        !userAuthenticated) {
        NSLog(@"Authentication changed: player authenticated.");
        userAuthenticated = TRUE;
    } else if (![GKLocalPlayer localPlayer].isAuthenticated &&
        userAuthenticated) {
        NSLog(@"Authentication changed: player not authenticated");
        userAuthenticated = FALSE;
    }

}
```

The init method checks to see if Game Center is available, and if so registers for the “authentication changed” notification. 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 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:

```
#pragma mark User functions

- (void)authenticateLocalUser {

    if (!gameCenterAvailable) return;

    NSLog(@"Authenticating local user...");
    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.

OK – GCTurnBasedMatchHelper now contains all of the code necessary to authenticate the user, so you just have to use it! Switch to **AppDelegate.m** and make the following changes:

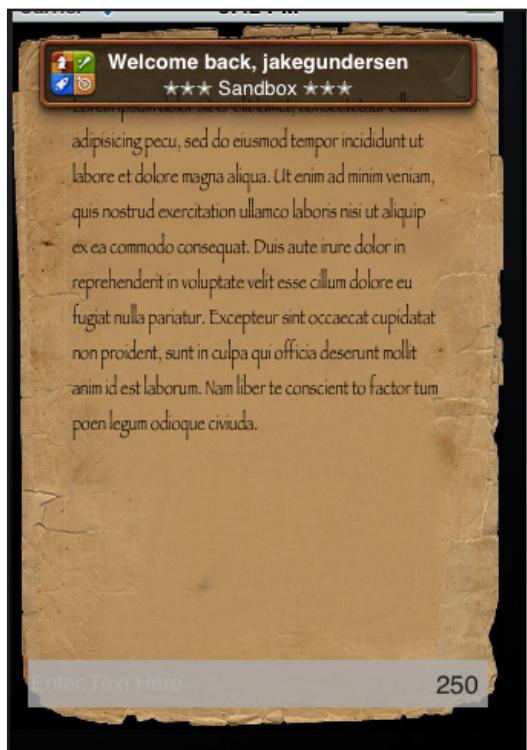
```
// At the top of the file
#import "GCTurnBasedMatchHelper.h"

// At the end of applicationDidFinishLaunching, right before
// the return YES
[[GCTurnBasedMatchHelper sharedInstance] authenticateLocalUser];
```

This creates the Singleton instance (which registers for the "authentication changed" callback as part of initialization), then calls the authenticateLocalUser method.

Usually you'd have to add the Game Kit framework to your project as well, but I've already added this for you in the **Build Phases\Link Binary with Libraries** section of your target settings.

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 we're successfully authenticating with Game Center and have a starter project ready, we're ready to start talking about the meat of this tutorial: Turn Based Gaming.

In a turn based game, you don't need to play at the same time as our 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, let's 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.

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

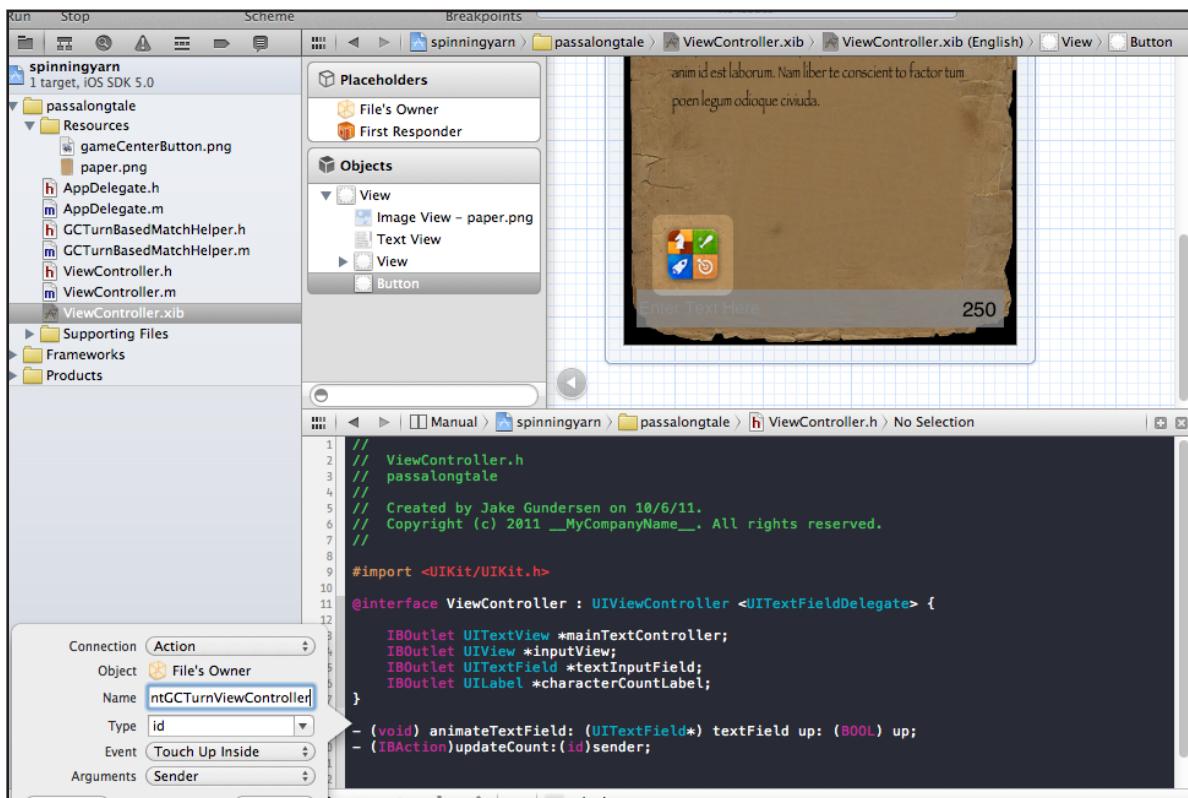


Last but not least, this singleton class has a delegate protocol, **GKTurnBasedEventHandlerDelegate** that provides notifications when the turn passes from one player to another, when we're invited by a friend to a match, or when the game ends.

Apple's Turn Based Match View Controller

The first thing we need to do to play a turn based match is create one! As discussed above, we can do that using Apple's provided `GKTurnBasedMatchmakerViewController`, so let's create a button we can present it to the screen.

Add the **gameCenterButton.png** file from the resources for this chapter into your project, and add a new button to the **ViewController.xib** as shown below:



Once you've added the button, set its type to **Custom** in the Attributes Inspector, and the Image to **gameCenterButton.png**. 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 is leaderboards, achievements, view controllers (turn based and live) and 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, we're not actually going to do the heavy lifting to present the GKTurnBasedMatchmakerViewController inside our ViewController class. Rather, we'll let the GCTurnBasedMatchHelper class do all that work for us. 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 and method into the header:

```
// New instance variable  
UIViewController *presentingViewController;  
  
// New method  
- (void)findMatchWithMinPlayers:(int)minPlayers  
    maxPlayers:(int)maxPlayers  
    viewController:(UIViewController *)viewController;
```

Here we create a new variable to store the view controller that will present the GKTurnBasedMatchmakerViewController, and a method that we'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];  
}
```

First we check the status of gameCenterAvailable. We can't do anything if game center isn't connected, so in that case we bail.

If we're connected, then we set up a GKMatchRequest. This object is the same as we would use for any multiplayer game center game. We are setting the minimum and maximum players for the request. It will control the GKTurnBasedViewController, not allowing us to include more than our max players or less than our min players.

We'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. 4 players is the maximum for a live multiplayer game, but the turn-based game can support up to 16!

We then create a new GKTurnBasedMatchmakerViewController, passing in the GKMatchRequest. We set the delegate of that object to self (GCTurnBasedMatchHelper). This will throw an error which we'll fix soon.

Then we set the showExistingMatches property to YES. This property controls what's presented to the user. If we set it to YES then we'll see all the matches we 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 we can add players until we reach the specified max. Each slot can be filled with an invitation to a specific player, or can be an automatch slot. If we set the showExistingMatches property to NO, then we'll be presented only with the create new game view.

Almost done - we 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 implement presentGCTurnViewController as follows:

```
- (IBAction)presentGCTurnViewController:(id)sender {  
    [[GCTurnBasedMatchHelper sharedInstance]  
        findMatchWithMinPlayers:2 maxPlayers:12 viewController:self];  
}
```

If you build and run now 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 we haven't implemented the delegate methods yet! So let's fix that next.

Implementing the Matchmaker View Controller Delegate

The first step is to open **GCTurnBasedMatchHelper.h** and modify the @interface as follows:

```
@interface GCTurnBasedMatchHelper : NSObject  
<GKTurnBasedMatchmakerViewControllerDelegate> {
```

Here we simply mark our 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:



```
#pragma mark GKTurnBasedMatchmakerViewControllerDelegate

-(void)turnBasedMatchmakerViewController:
    (GKTurnBasedMatchmakerViewController *)viewController
didFindMatch:(GKTurnBasedMatch *)match {
    [presentingViewController
        dismissModalViewControllerAnimated:YES];
    NSLog(@"did find match, %@", match);
}

-(void)turnBasedMatchmakerViewControllerWasCancelled:
    (GKTurnBasedMatchmakerViewController *)viewController {
    [presentingViewController
        dismissModalViewControllerAnimated:YES];
    NSLog(@"has cancelled");
}

-(void)turnBasedMatchmakerViewController:
    (GKTurnBasedMatchmakerViewController *)viewController
didFailWithError:(NSError *)error {
    [presentingViewController
        dismissModalViewControllerAnimated:YES];
    NSLog(@"Error finding match: %@", error.localizedDescription);
}

-(void)turnBasedMatchmakerViewController:
    (GKTurnBasedMatchmakerViewController *)viewController
playerQuitForMatch:(GKTurnBasedMatch *)match {
    NSLog(@"playerquitforMatch, %@, %@", match, match.currentParticipant);
}

@end
```

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 our 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 we've lost connectivity or for a variety of other reasons.

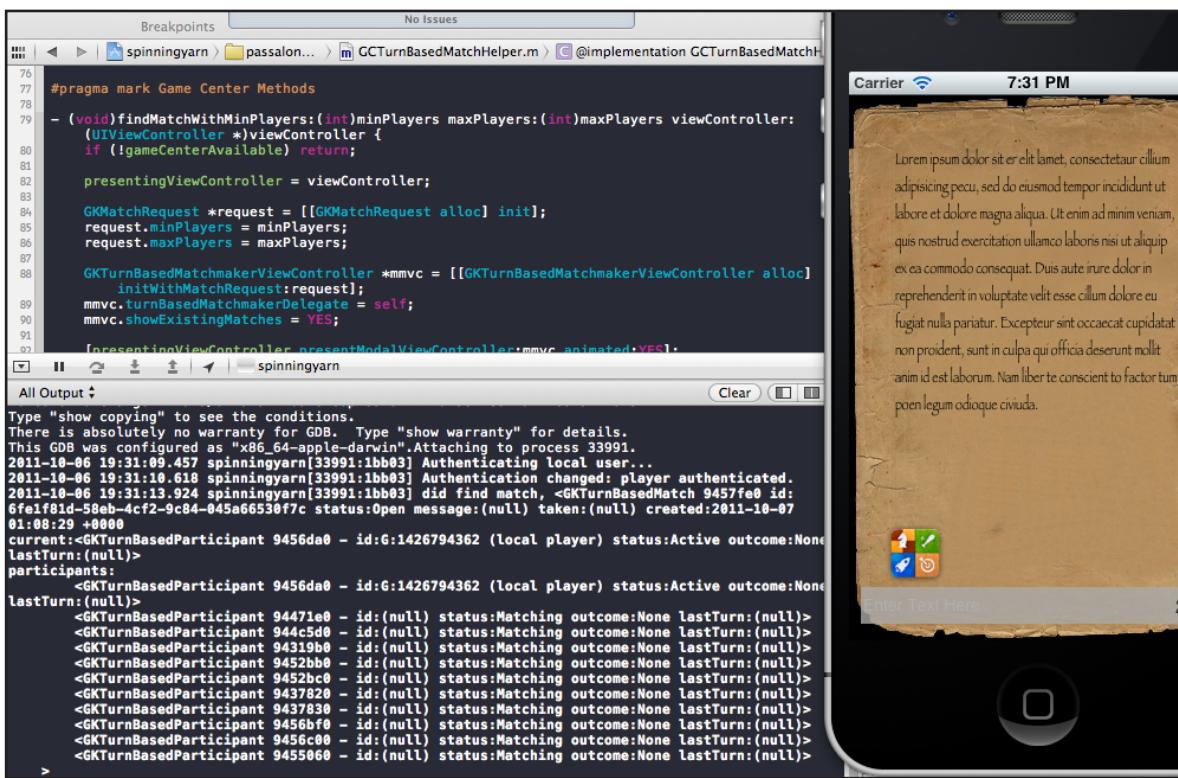
The final method (`playerQuitForMatch`) method is fired when a player swipes a match (while it's their turn) and quits it. Swiping a match will reveal a quit (if it's a match that is still active) or remove button. If a player quits a match while it's still their turn, they need to handle the match, 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.

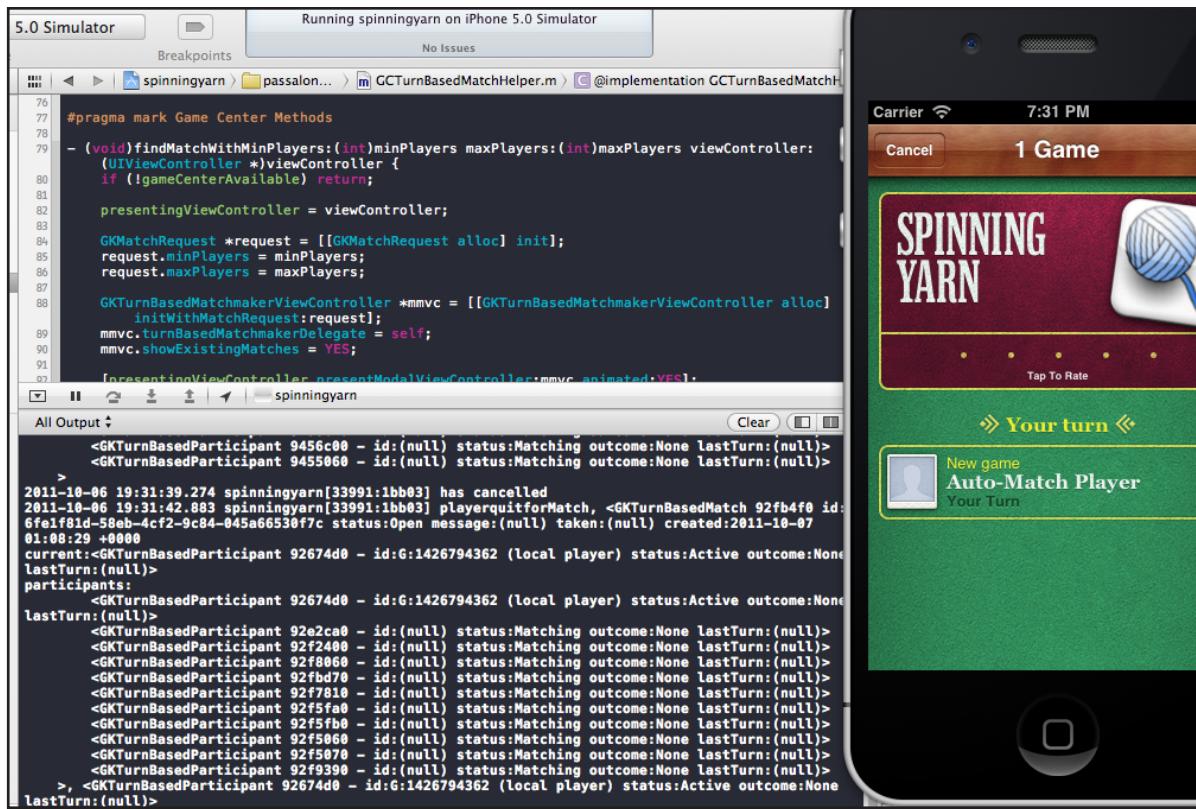
In each of these methods we're just dismissing the view controller and just logging out some information for now. An exception is the playerQuit method, where we don't dismiss the view controller, because the user might want to keep doing something else.

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 `GKTurnBasedParticipants` is yourself (the local player) and the second has an id of null (it hasn't found a partner for you yet).

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.



We're going to return to these methods and finish their implementation later. For now, we can leave them as they are.

Sending a Turn

We need to set up a method that sends a turn. When it's our turn, we want to be able to add a string of text (max 250 characters) to the end of the current story. When we send a turn, we'll add our text to the previous text, then we'll send that new string with our turn.

It's probably a good time to talk about the `GCTurnBasedMatch` object - remember this is passed to us in our `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 us a `GCTurnBasedMatch` to work with.

As you saw in the logs earlier, this object has a `participants` array that contains a `GCTurnBasedParticipant` 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 4096 bytes that we'll use to record the state of our match (the string of the story for this game), which will be passed from one player to the next.



The first thing we need to do is keep track of the current match, so we can modify the match data and have each player take their turn. So open **GCTurnBasedMatchHelper.h** and add the following after the @interface:

```
@property (retain) GCTurnBasedMatch * currentMatch;
```

And synthesize it in **GCTurnBasedMatchHelper.m**:

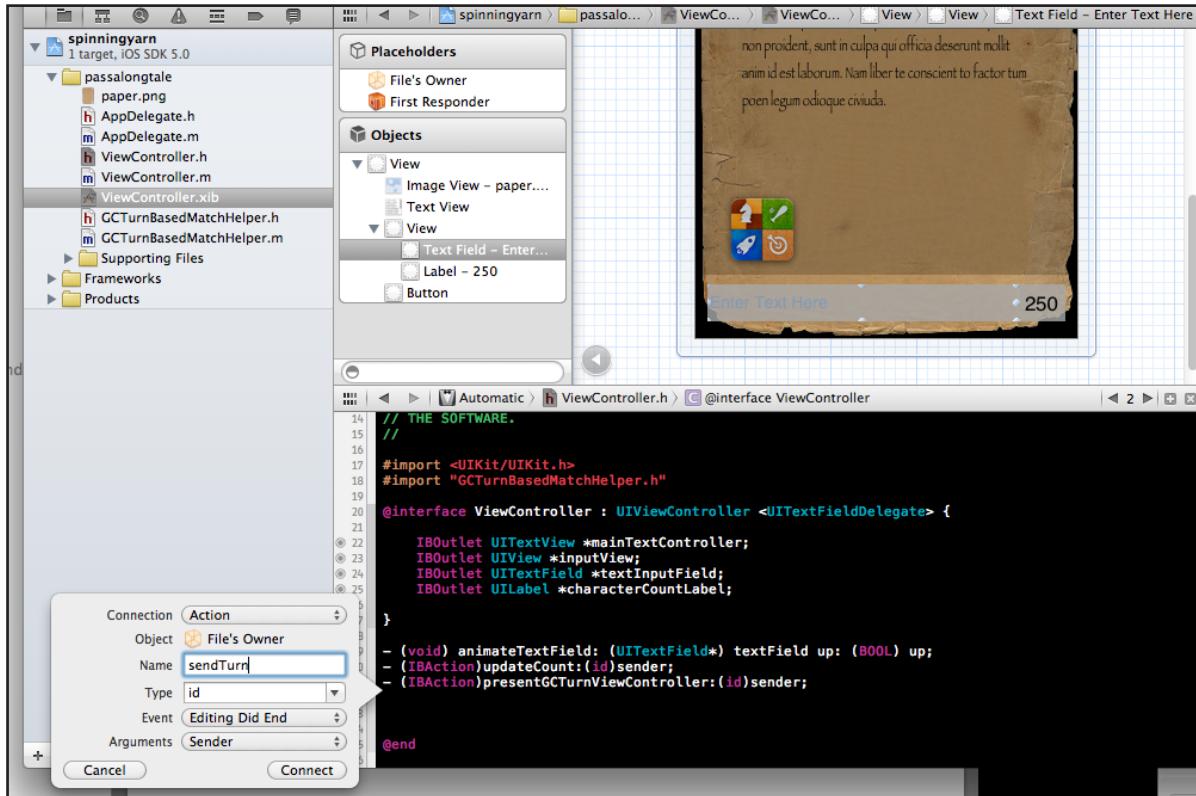
```
@synthesize currentMatch;
```

Let's modify our didFinidMatch callback to set the current match to the passed in match. Just add this to the bottom of the method:

```
self.currentMatch = match;
```

Now let's create a method to let the player take his turn (by adding a new bit of text to the story in this game). We'll call this method when the user enters some text in the text field and hits Done on the keyboard.

Open up **ViewController.xib**, and bring up the Assistant Editor so **ViewController.h** 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.



Switch to up **ViewController.m** and implement the method as follows:

```
- (IBAction)sendTurn:(id)sender {
    GKTurnBasedMatch *currentMatch =
        [[GCTurnBasedMatchHelper sharedInstance] currentMatch];
    NSString *newStoryString;
    if ([textInputField.text length] > 250) {
        newStoryString = [textInputField.text substringToIndex:249];
    } else {
        newStoryString = textField.text;
    }
    NSString *sendString = [NSString stringWithFormat:@"%@ %@", mainTextController.text, newStoryString];
    NSData *data =
        [sendString dataUsingEncoding:NSUTF8StringEncoding];
    mainTextController.text = sendString;

    NSUInteger currentIndex = [currentMatch.participants
        indexOfObject:currentMatch.currentParticipant];
    GKTurnBasedParticipant *nextParticipant;
    nextParticipant = [currentMatch.participants objectAtIndex:
        ((currentIndex + 1) % [currentMatch.participants count])];
    [currentMatch endTurnWithNextParticipant:nextParticipant
        matchData:data completionHandler:^(NSError *error) {
        if (error) {
            NSLog(@"%@", error);
        }
    }];
    NSLog(@"Send Turn, %@", %@", data, nextParticipant);
    textField.text = @"";
    characterCountLabel.text = @"250";
    characterCountLabel.textColor = [UIColor blackColor];
}
```

This method is doing a whole bunch of things. Let's step through each bit.

First we set up the currentMatch variable by retrieving the match from the GCTurnBasedMatchHelper singleton. We may be involved in many matches at a time, but we'll only be displaying one match at a time. We'll keep track of this one match in our currentMatch variable.

Next we need to check to see if the length of the string we input into the text field is too long. If the string is longer than 250 characters we cut it off with the substringToIndex call. If not, we just pass the string into our variable.

Next we create an NSData object by combining the string that's in the mainTextController with the string we just created. The dataUsingEncoding method converts the NSString representation to a UTF8 string and stores it in the data object.



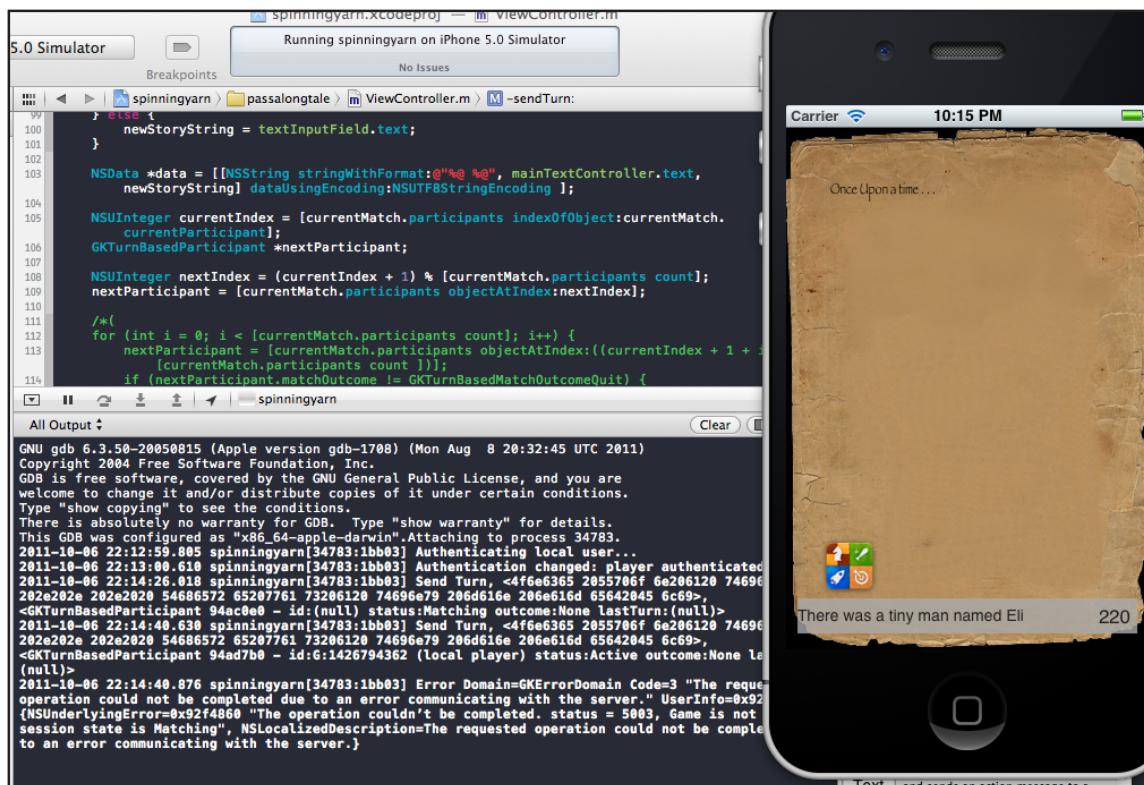
The next few lines set up the nextParticipant object. Each time we send a turn we need to send information about who the next person in the turn rotation is. We can set up our game to send the turn to any participant, including ourselves. We can follow a strict rotation where the order of players always follows a pattern. Or, we can set up other rules about who gets the next turn.

In our case we are retrieving the index of the current player, or the position of the currentMatch.currentParticipant property, in the currentMatch.participants array. Once we have the index of the current player, we just add one to get the next participant from the participants array.

Once we have the NSData and the nextParticipant set, we 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 our case we are just logging any error that occurs.

We also log out the nextParticipant and the data object. We can take a look at this to see if this is working as expected. The last action is to clear the contents of the textField and reset the characterCountLabel to 250 so when we 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:



You'll notice that in this log there is an error message. This is because I ran the method twice. The first time it sent fine. However, once I sent my turn to the server, it was no longer my turn.

The match is now in the status "matching" which means that it's looking for an automatched second player. When I try and send another turn for my player, this is the error message that comes back and my new turn isn't recorded.

Taking Turns

There are several ways that we can take a turn in our app. When we use automatch for a slot in our game, one of two things will happen. Either

1. We'll get back a new match, with our player as the first participant, or
2. We'll get back an existing match where our player has been placed in a second, third, fourth, etc slot in an existing match.

In both scenarios, `didFindMatch` will be called. We'll need to write some code to distinguish between the two cases, because we want different things to happen in each case:

1. If we get back a new match, we want to create a clean slate for our story. In our app, we'll use the string "Once upon a time . . ." every time we begin a new story.
2. If we 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, we want to show this in our text view.

We 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 their first turn. We're going to use the `lastTurn` property of the first participant in the `participants` array to determine if we are the first turn, or if another player has started the match. If we find that `lastTurn` is null, we'll assume that we're dealing with a new match, otherwise we'll assume that we already have `matchData` that we'll be dealing with.

So open up **GCTurnBasedMatchHelper.m** and replace the `didFindMatch` method as follows:

```
-(void)turnBasedMatchmakerViewController:  
    (GKTurnBasedMatchmakerViewController *)viewController
```



```
didFindMatch:(GKTurnBasedMatch *)match {
    [presentingViewController
        dismissModalViewControllerAnimated:YES];
    self.currentMatch = match;
    GKTurnBasedParticipant *firstParticipant =
        [match.participants objectAtIndex:0];
    if (firstParticipant.lastTurnDate) {
        NSLog(@"existing Match");
    } else {
        NSLog(@"new Match");
    }
}
```

This code should be easy to comprehend. We're getting the participant at the first index in our participants array, then testing to see if the lastTurnDate is populated. If it is, we know we're entering a match that has already started. If not, we know it's a new match and we are logging the case either way.

At this point, you'll need a second device or to switch between sandboxed game center accounts in order to get the existing match scenario. A single account/player can only take the first turn, after that you need a second player involved to test many of these scenarios.

So run the app on both 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 automatch 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 automatch 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 automatch while we're on the subject. Automatch 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 automatched until after the first and second player have both taken their turn.

Implementing Our Own Delegate Protocol

Now is a good time to introduce our new delegate protocol. We are going to build a protocol to manage the communication between the GCTurnBasedMatchHelper class



and our ViewController. This is better than hard-coding the ViewController into the GCTurnBasedHelper, because it will allow us 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 our ViewController class to handle what do for each case, because that's particular to the game logic.

Lets go ahead and build the delegate protocol now. Modify **GCTurnBasedMatchHelper.h** to look like the following (notice the new protocol, and delegate instance variable and property):

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

@interface GCTurnBasedMatchHelper : NSObject
<GKTurnBasedMatchmakerViewControllerDelegate> {
    BOOL gameCenterAvailable;
    BOOL userAuthenticated;
    UIViewController *presentingViewController;

    GKTurnBasedMatch *currentMatch;

    id <GCTurnBasedMatchHelperDelegate> delegate;
}

@property (nonatomic, retain)
    id <GCTurnBasedMatchHelperDelegate> delegate;
@property (assign, readonly) BOOL gameCenterAvailable;
@property (nonatomic, retain) GKTurnBasedMatch *currentMatch;

+ (GCTurnBasedMatchHelper *)sharedInstance;
- (void)authenticateLocalUser;
- (void)authenticationChanged;
- (void)findMatchWithMinPlayers:(int)minPlayers maxPlayers:(int)maxPlayers
    viewController:(UIViewController *)viewController;

@end
```



Also before we forget, switch to **GCTurnBasedMatchHelper.m** and synthesize the delegate as follows:

```
@synthesize delegate;
```

We've added the five protocol method declarations and a new instance variable, delegate. The delegate object will be sent the methods, and it will be up to that delegate (in our case the ViewController class) to implement them.

Let's quickly go through them now. Then later when we implement them we'll explain them in more detail.

1. **enterNewGame.** The first method we've already discussed the use for. When we are presented with a new game from the didFindMatch method, we want to display the "Once upon a time" starter text to the screen.
2. **layoutMatch.** The layoutMatch method is used when we want to view a match where it's another player's turn (just to check the state of the story for example). We want to prevent the player from sending a turn in this case, but we 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 our player's turn, but it's an existing match. This scenario exists when our player chooses an existing match from the GKTurnBasedMatchmakerViewController, or when a new turn notification comes in. We'll talk about notifications a little later in this tutorial.
4. **receiveEndGame.** The receiveEndGame method will be called when a match has ended on our player's turn, or when we receive a notification that a match has ended on another player's turn. For this simple game, we'll just end the game when we are getting close to the current NSData turn-based game size limit (4096 bytes).
5. **sendNotice.** The sendNotice method happens when we receive an event (update turn, end game) on a match that isn't one we're currently looking at. If we receive an end game notice on a match that we've got loaded into our currentMatch variable, we'll update the UI to reflect the current state of that match, but if we receive the same notice on a match other than the one we're looking at, we don't want to automatically throw the user into that match, taking them away from the match they are currently looking at. We'll decide how to handle this later on.

Let's go ahead and send the delegate methods in our didFindMatch method for our 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
        dismissModalViewControllerAnimated:YES];
    self.currentMatch = match;
    GKTurnBasedParticipant *firstParticipant =
        [match.participants objectAtIndex:0];
    if (firstParticipant.lastTurnDate) {
        [delegate takeTurn:match];
    } else {
        [delegate enterNewGame:match];
    }
}

```

Next, open up **ViewController.h** and mark it as implementing our new protocol:

```

@interface ViewController : UIViewController <UITextFieldDelegate,
GCTurnBasedMatchHelperDelegate> {

```

Then switch to **ViewController.m** and implement the `enterNewGame` and `takeTurn` methods:

```

#pragma mark - GCTurnBasedMatchHelperDelegate

-(void)enterNewGame:(GKTurnBasedMatch *)match {
    NSLog(@"Entering new game...");
    mainTextController.text = @"Once upon a time";
}

-(void)takeTurn:(GKTurnBasedMatch *)match {
    NSLog(@"Taking turn for existing game...");
    if ([match.matchData bytes]) {
        NSString *storySoFar =
            [NSString stringWithUTF8String:[match.matchData bytes]];
        mainTextController.text = storySoFar;
    }
}

```

Pretty simple, eh?

One last thing, we need to set the delegate property of the `GCTurnBasedMatchHelper` class to our `ViewController`. Let's do that in the `viewDidLoad` method, like so:

```
[GCTurnBasedMatchHelper sharedInstance].delegate = self;
```

Build and run on both devices, and you should be able to pass turns around!

Start a new game (either by invitation or automatch) 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 automatch create another automatch game on the other device with the same number of players. Again, the automatch 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 we 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.

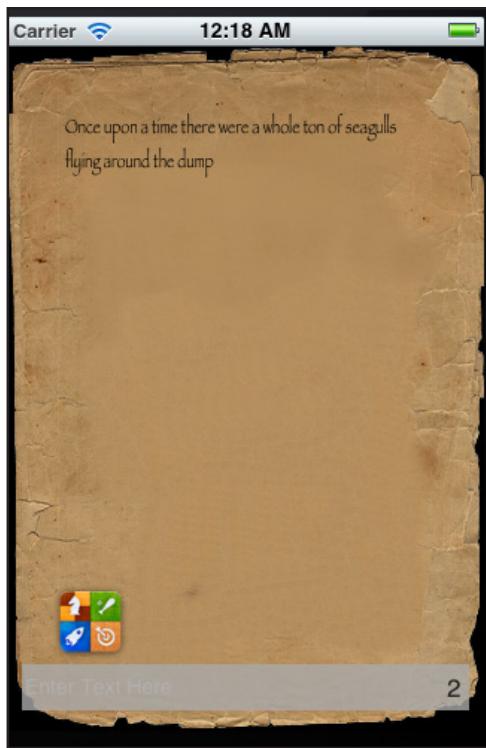
Too many matches? At this point I have about five matches that have wrong data and other problems with them. Some of your old games may act wrong because the data in them is incomplete or missing. I'd remove all of them and start fresh. A good way to clean out all this data is to call a method that programmatically deletes all the matches for a user.

Put this code into the completion block for authenticateWithCompletionHandler like so:

```
if ([GKLocalPlayer localPlayer].authenticated == NO) {
    [[GKLocalPlayer localPlayer]
     authenticateWithCompletionHandler:^(NSError *error) {
        [GKTurnBasedMatch loadMatchesWithCompletionHandler:
         ^(NSArray *matches, NSError *error){
            for (GKTurnBasedMatch *match in matches) {
                NSLog(@"%@", match.matchID);
                [match removeWithCompletionHandler:^(NSError *error){
                    NSLog(@"%@", error);}];
            }]];
        }];
    } else {
        NSLog(@"Already authenticated!");
    }
}
```

This code will only clear out the matches for one Game Center account, so you'll have to run it on both devices. Also, remember when you have this code in it will clear all your matches on startup, so make sure to comment it out when you don't need it.



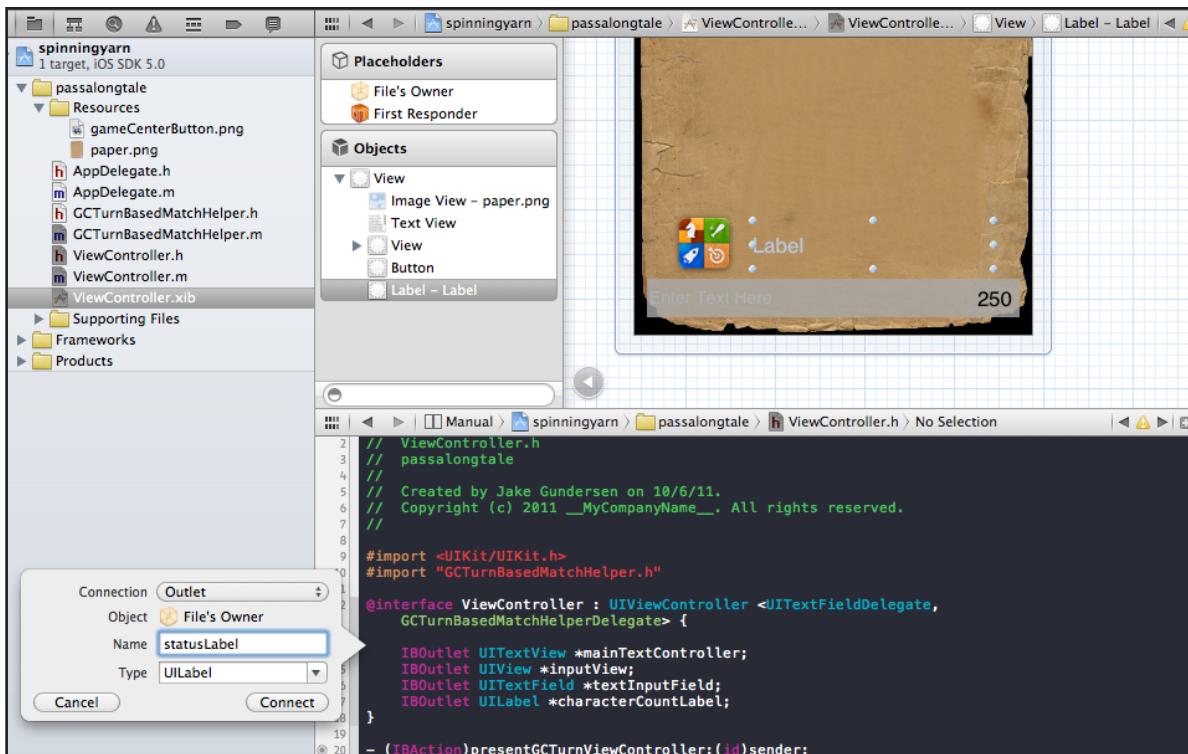


What To Do When It's Not Our Turn

Currently we have the ability to input text and run the game while it's not our turn. While the API prevents us from updating the game state outside our turn, our 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, we want to update a status label telling the player that the match is currently in another player's turn. Also, we should disable the text field.

Open up **ViewController.xib**, drag a label next to to the Game Center icon, and make it as wide as the screen. Bring up the Assistant Editor, make sure **ViewController.h** is visible, control-drag from the label down inside the `@interface` and connect it to an outlet named **statusLabel**.



We'll need to make some changes in both the GCTurnBasedMatchHelper class and the ViewController class in order keep track of whose turn it is. Let's start by editing our didFindMatch method in **GCTurnBasedMatchHelper.m**:

```

-(void)turnBasedMatchmakerViewController:
(GKTurnBasedMatchmakerViewController *)viewController
didFindMatch:(GKTurnBasedMatch *)match {
[presentingViewController
dismissModalViewControllerAnimated:YES];
self.currentMatch = match;
GKTurnBasedParticipant *firstParticipant =
[match.participants objectAtIndex:0];
if (firstParticipant.lastTurnDate == NULL) {
    // It's a new game!
    [delegate enterNewGame:match];
} else {
    if ([match.currentParticipant.playerID
isEqualToString:[GKLocalPlayer localPlayer].playerID]) {
        // It's your turn!
        [delegate takeTurn:match];
    } else {
        // It's not your turn, just display the game state.
        [delegate layoutMatch:match];
    }
}
}

```

Note we swapped the if/else statement around a bit here.

We're checking the current player's (from the match) playerID against the current player that's logged into game center. If they match, it's our player's turn. In that case we send the same method. However, if they don't match then it's not the current player's turn (this can happen when it's someone else's turn, or when it's no one's turn, because the match has ended).

If it's not our player's turn, then we're going to send a different method, the layoutMatch method. This just updates the UI to reflect the current state of the match. In our app, we'll be doing this a lot since we'll want the player to be able to watch as the story progresses.

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];
    }
    statusLabel.text = statusString;
    textField.enabled = NO;
    NSString *storySoFar = [NSString stringWithUTF8String:
        [match.matchData bytes]];
    mainTextController.text = storySoFar;
}
```

The first thing we do in this method is construct the string we put in `statusLabel`. We check the `GKTurnBasedMatchStatus` `match.status` and set the string based on whether the game is running or ended.

If we currently waiting on another player, we want to get the position of the player in the array. We'll add one so we don't have a Player 0. We construct the string for the player's turn and set the label to that string.

Next, we are disabling the `textField` so that our player won't have the ability to enter text. Finally, as we have before, we update the `mainTextController` to hold the body of the story.



Let's go back and make a few edits to our other two methods to incorporate the new label and the logic that will turn on the text field. Add this code:

```
- (void)enterNewGame:(GKTurnBasedMatch *)match {
    NSLog(@"Entering new game...");
    statusLabel.text = @"Player 1's Turn (that's you)";
    textField.enabled = YES;
    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];
    statusLabel.text = statusString;
    textField.enabled = YES;
    if ([match.matchData bytes]) {
        NSString *storySoFar = [NSString stringWithUTF8String:
            [match.matchData bytes]];
        mainTextController.text = storySoFar;
    }
}
```

This code is very similar, except we've added some code to enable the text input field and display the status.

A few other things we need to do, Change the properties on the statusLabel so that it's in word wrap mode and has 2 lines (and make it a bit taller).

Also in the viewDidLoad method, let's set the statusLabel to something like, "Press the game center button to get started" and disable the textField:

```
textField.enabled = NO;
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 note you still have to go back into Game Center and re-select a game to get updates, but we're one step closer to a functional game!





Finishing the Delegate Methods

We're finished with the `didFindMatch` method, but we 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 we'd want to handle the various errors in a more elegant way, but for our purpose here, logging the error is fine.

But we do have to change the `playerDidQuit` 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;  
  
    for (int i = 0; i < [match.participants count]; i++) {  
        part = [match.participants objectAtIndex:  
            (currentIndex + 1 + i) % match.participants.count];
```



```
    if (part.matchOutcome != GKTurnBasedMatchOutcomeQuit) {
        break;
    }
}
NSLog(@"playerquitforMatch, %@", %@", match, match.currentParticipant);
[match participantQuitInTurnWithOutcome:
    GKTurnBasedMatchOutcomeQuit nextParticipant:part
    matchData:match.matchData completionHandler:nil];
}
```

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, we 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 we 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 us and all that is dealt with automatically.

In this case we're iterating through the list of participants and looking for a participant who doesn't have a `matchOutcome` of `GKTurnBasedMatchOutcomeQuit`. We don't want to pass the match to a player who has quit. If we do, we'll get an error and that turn won't be recorded.

When we find the next participant in the array that doesn't have a quit status, we call `participantQuitInMatchWithOutcome:nextParticipant: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, we 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 we're fixing this, we should make some of the same changes to our `sendTurn` method. In a case like this one, we want to iterate through the participants and make sure the one we'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 id:4160 - 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
{NSError=0x1832a0 "The operation couldn't be completed. status = 5003, Session:
33f835fb-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 ([textInputField.text length] > 250) {
        newStoryString = [textInputField.text substringToIndex:249];
    } else {
        newStoryString = textInputField.text;
    }
    NSString *sendString = [NSString stringWithFormat:@"%@ %@", mainTextController.text, newStoryString];
    NSData *data = [sendString
        dataUsingEncoding:NSUTF8StringEncoding];
    mainTextController.text = sendString;

   NSUInteger currentIndex = [currentMatch.participants
        indexOfObject:currentMatch.currentParticipant];
    GKTurnBasedParticipant *nextParticipant;

    NSUInteger nextIndex = (currentIndex + 1) %
        [currentMatch.participants count];
    nextParticipant =
    [currentMatch.participants objectAtIndex:nextIndex];

    for (int i = 0; i < [currentMatch.participants count]; i++) {
        nextParticipant = [currentMatch.participants
            objectAtIndex:(((currentIndex + 1 + i) %
            [currentMatch.participants count])]];
        if (nextParticipant.matchOutcome !=
            GKTurnBasedMatchOutcomeQuit) {
            break;
        }
    }

    [currentMatch endTurnWithNextParticipant:nextParticipant
        matchData:data completionHandler:^(NSError *error) {
            if (error) {
                NSLog(@"%@", error);
                statusLabel.text =
                @"Oops, there was a problem. Try that again.";
            }
        }
    ]
}

```



```
        } else {
            statusLabel.text = @"Your turn is over.";
            textField.enabled = NO;
        }
    }];
    NSLog(@"Send Turn, %@", %@", data, nextParticipant);
    textField.text = @"";
    characterCountLabel.text = @"250";
}
```

First we add the code that runs through all the participants, the first time it finds one where they haven't quit (usually this will be the first iteration in the loop) it breaks out of the loop. This way we skip over participants who have quit.

The other thing we added here was two bits of unrelated code. First, we update the status if there was an error or if there wasn't. Second, if the turn was sent without problem, we disable the textField so that another turn cannot be sent immediately to that game.

If you build and run now, you should see the status update when you send a turn!



Event Handler Delegate

Our game is coming along well so far, but there's one major problem - we 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 we 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 our app when certain events happen, like when it's our player's turn.

There are three delegate callbacks that give our 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 our 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, we first need to set ourselves as the delegate of the GKTurnBasedEventHandler. This object is a singleton and the only time we deal with it directly is when we set ourselves as its delegate.

Like the view controller delegate protocol, we're going to be using our GCTurnBasedMatchHelper class to act as an intermediary for all the communication from these notifications. So, that's what we need to set up as the delegate. We have to set the delegate after we have logged in to game center or it may not work. Change the authenticateUser code to the following:

```
- (void)authenticateLocalUser {
    if (!gameCenterAvailable) return;

    void (^setGKEventHandlerDelegate)(NSError *) = ^ (NSError *error)
    {
        GKTurnBasedEventHandler *ev =
            [GKTurnBasedEventHandler sharedTurnBasedEventHandler];
        ev.delegate = self;
    };

    NSLog(@"Authenticating local user...");
    if ([GKLocalPlayer localPlayer].authenticated == NO) {
        [[GKLocalPlayer localPlayer]
            authenticateWithCompletionHandler:
            setGKEventHandlerDelegate];
    } else {
        NSLog(@"Already authenticated!");
        setGKEventHandlerDelegate(nil);
    }
}
```

You can see here that I'm setting up a block (we do this so we can pass it into the completionHandler of the authenticate method). The block just gets a pointer to the singleton and uses the pointer to set the delegate to self. Simple.



Then we pass that in as the completionHandler parameter in the case where we need to authenticate. If we don't need to authenticate then we just call the block.

The block takes an NSError parameter if it's run by the completionHandler. If we call it directly we can just enter nil for the error (because the error would have been coming from any problem with the authenticate method, we didn't run it).

Alright, we're set up to receive callbacks from the GKTurnBasedEventHandlerDelegate, well almost. We also need to set our object as a GKTurnBasedEventHandlerDelegate and implement the methods. Do that now:

In **GCTurnBasedMatchHelper.h**, change the @interface line to:

```
@interface GCTurnBasedMatchHelper : NSObject  
    <GKTurnBasedMatchmakerViewControllerDelegate,  
     GKTurnBasedEventHandlerDelegate> {
```

And add the following code to **GCTurnBasedMatchHelper.m**:

```
#pragma mark GKTurnBasedEventHandlerDelegate  
  
-(void)handleInviteFromGameCenter:(NSArray *)playersToInvite {  
    NSLog(@"new invite");  
}  
  
-(void)handleTurnEventForMatch:(GKTurnBasedMatch *)match {  
    NSLog(@"Turn has happened");  
}  
  
-(void)handleMatchEnded:(GKTurnBasedMatch *)match {  
    NSLog(@"Game has ended");  
}
```

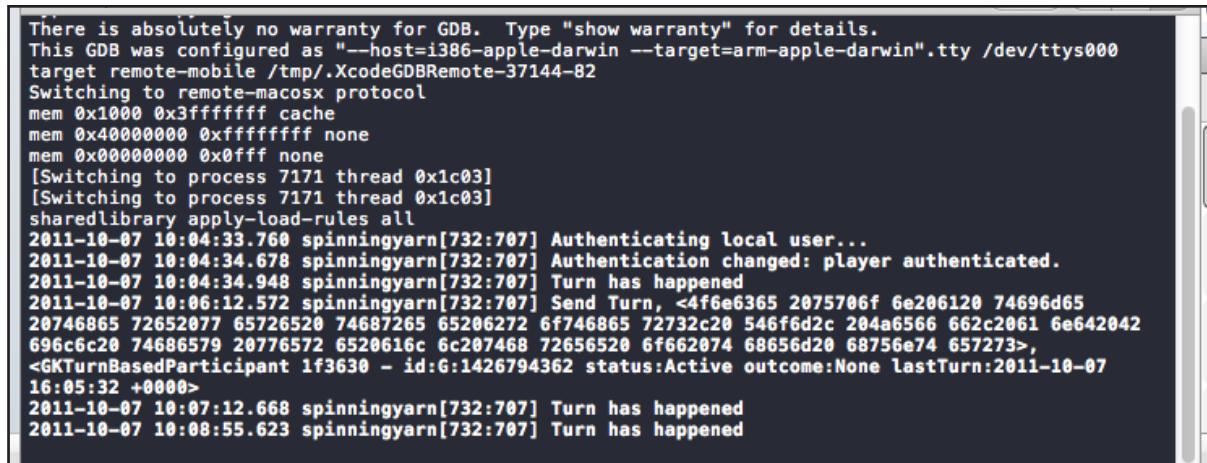
Build and run now. You should be able to test the handleTurn event by letting the other player take a turn, and you should see the log message. w00t!

The handleMatchEnded will require us 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.

Here's an output to the console for the handleTurn event:





```

There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i386-apple-darwin --target=arm-apple-darwin".tty /dev/ttys000
target remote-mobile /tmp/.XcodeGDBRemote-37144-82
Switching to remote-macosx protocol
mem 0x1000 0x3fffffff cache
mem 0x40000000 0xffffffff none
mem 0x00000000 0xffff none
[Switching to process 7171 thread 0x1c03]
[Switching to process 7171 thread 0x1c03]
sharedlibrary apply-load-rules all
2011-10-07 10:04:33.760 spinningyarn[732:707] Authenticating local user...
2011-10-07 10:04:34.678 spinningyarn[732:707] Authentication changed: player authenticated.
2011-10-07 10:04:34.948 spinningyarn[732:707] Turn has happened
2011-10-07 10:06:12.572 spinningyarn[732:707] Send Turn, <4f6e6365 2075706f 6e206120 74696d65
20746865 72652077 6526520 74687265 65206272 6f746865 72732c20 546f6d2c 204a6566 662c2061 6e642042
696c6c20 74686579 20776572 6520616c 6c207468 72656520 6f662074 68656d20 68756e74 657273>,
<GKTurnBasedParticipant 1f3630 - id:6:1426794362 status:Active outcome:None lastTurn:2011-10-07
16:05:32 +0000>
2011-10-07 10:07:12.668 spinningyarn[732:707] Turn has happened
2011-10-07 10:08:55.623 spinningyarn[732:707] Turn has happened

```

Handling Invitations

Let's write `handleInviteFromGameCenter` next. You may assume erroneously at first, as I did, that this method fires whenever we receive a named invite to a game. This is not what the method is for at all!

There is no method 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 the game. So, when you switch from game center to the game, there's information about who 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 we get a new invite from game center, we need to instantiate the `GKTurnBasedMatchmakerViewController` with a `GKMatchRequest`. This method will give us an array of players that are supposed to be in the match. We'll use this object to set up the `GKMatchRequest`. Here's what that code should look like:

```

-(void)handleInviteFromGameCenter:(NSArray *)playersToInvite {
    [presentingViewController
        dismissModalViewControllerAnimated:YES];
    GKMatchRequest *request =
        [[[GKMatchRequest alloc] init] autorelease];
    request.playersToInvite = playersToInvite;
    request.maxPlayers = 12;
    request.minPlayers = 2;
    GKTurnBasedMatchmakerViewController *viewController =
        [[GKTurnBasedMatchmakerViewController alloc]
            initWithMatchRequest:request];
    viewController.showExistingMatches = NO;
}

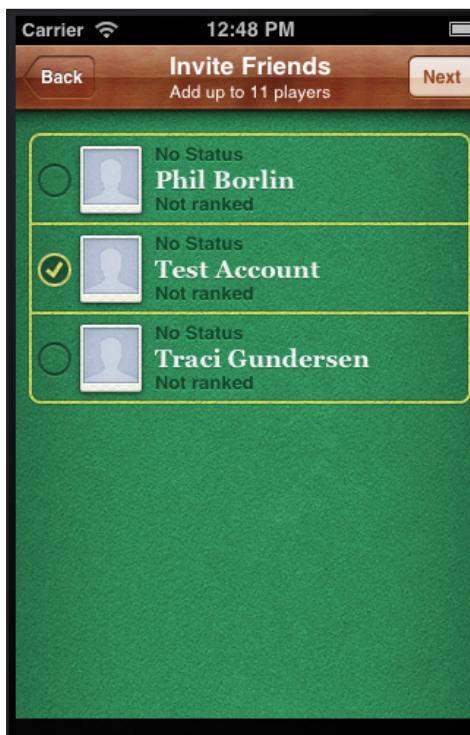
```



```
viewController.turnBasedMatchmakerDelegate = self;
[presentingViewController
    presentViewController:viewController animated:YES];
}
```

The first thing we do is get rid of any current modal view controller that is present. After that we set up the GKMatchRequest and then the GKTurnBasedMatchmakerViewController. Note that the showExistingMatches is set to NO. We only want to see the new game view for this match. We set up the delegate and then present our view controller.

If you build and run now you should be able to start a match from the Game Center (it won't call this method if you do an invite from inside our game) and send the invite. One thing to note, the simulator doesn't receive any notifications from the GKTurnBasedEventHandlerDelegate, so you'll have to test this on a device.



Handling the Turn Event

When the 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. We need to distinguish between these scenarios and handle each in turn.



Here's the code:

```
-(void)handleTurnEventForMatch:(GKTurnBasedMatch *)match {
    NSLog(@"Turn has happened");
    if ([match.matchID isEqualToString:currentMatch.matchID]) {
        if ([match.currentParticipant.playerID
            isEqualToString:[GKLocalPlayer localPlayer].playerID]) {
            // it's the current match and it's our turn now
            self.currentMatch = match;
            [delegate takeTurn:match];
        } else {
            // it's the current match, but it's someone else's turn
            self.currentMatch = match;
            [delegate layoutMatch:match];
        }
    } else {
        if ([match.currentParticipant.playerID
            isEqualToString:[GKLocalPlayer localPlayer].playerID]) {
            // it's not the current match and it's our turn now
            [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
        }
    }
}
```

We end up with four scenarios. We'll send delegate methods for three of them, the fourth we'll ignore, but we've set it up here in case you wish to handle it in another game.

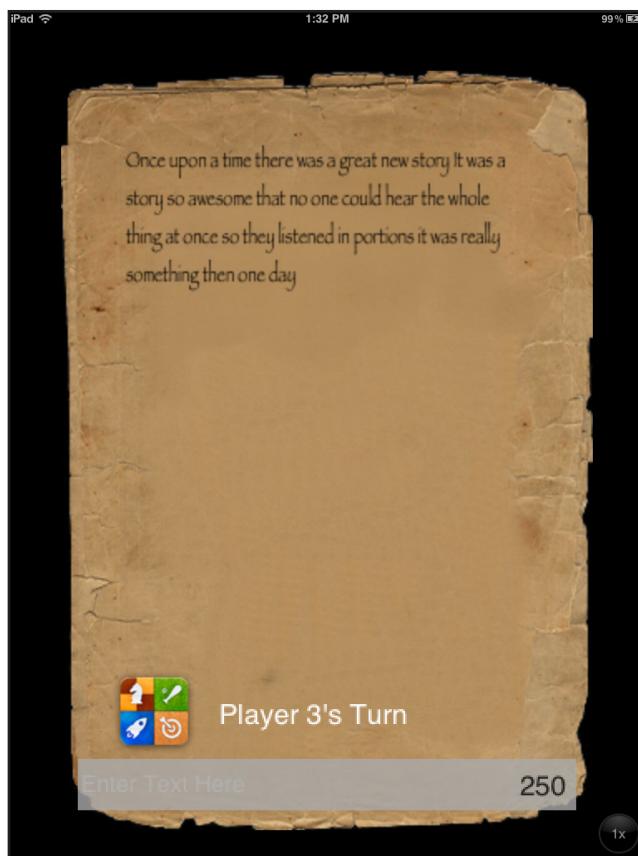
The first two are that the current match is the same as the match we're in. In that case we'll send `takeTurn`, if it's our turn, and `layoutMatch` if it's not. We're still setting the `currentMatch` to our passed in `match`, because even though the `matchID` is the same, the incoming match has updated state data (`match.matchData`) and the `currentParticipant` has changed.

If we are sent a notice for a match that we're not looking at, and it has become our turn, we'll send the `sendNotice` delegate method. We'll use this method to display an alert letting the user know.

If the turns change on matches that we're not looking at, and it's not our turn, we'll do nothing. The users can go looking at those matches by loading them using the `GKTurnBasedMatchmakerViewController`, we don't need to interrupt them every time things change on every match. In other kinds of games, we may want to do just that, so that section is available to you.



With the handleTurn method in place, you can now run the game. 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!

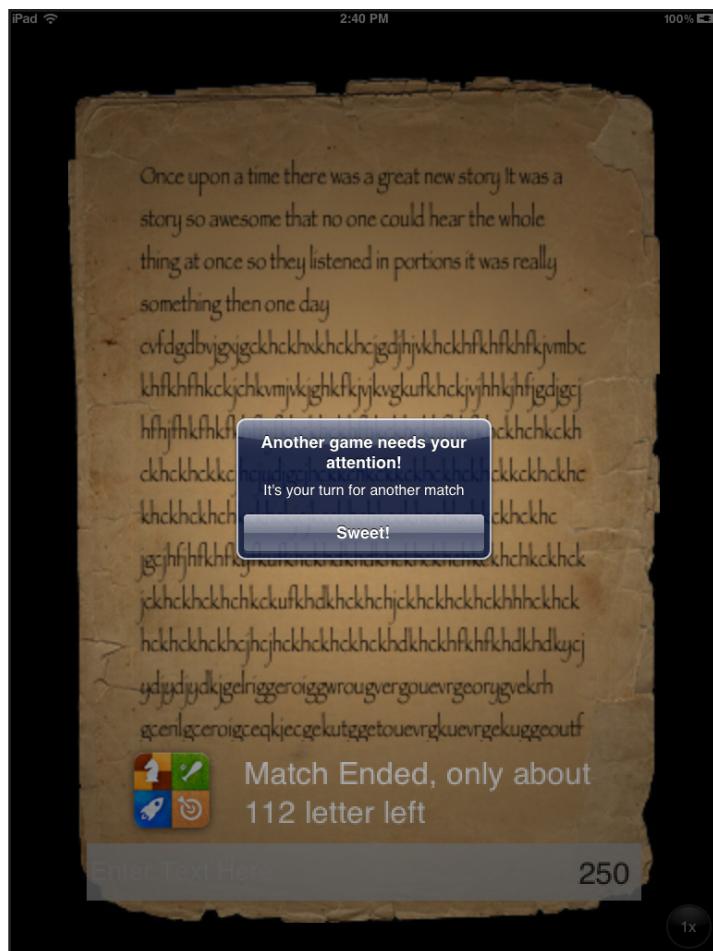


Depending on the state of your game, you might get an error saying the sendNotice callback isn't implemented yet. That's our next step anyway, so let's go ahead and implement the sendNotice method in **ViewController.m**.

```
- (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];  
    [av release];  
}
```

This is pretty straightforward. We're just letting the player know that another match needs their attention. We'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:





Ending the Game

We've really only got one thing left to accomplish, and that's ending the game. But, we probably also want to give our users some advance notice that they only have a certain number of turns left.

We'll do this with a new method that checks the length of the `NSData`. We'll end the game when it gets to above 3800 characters, but let's start letting our players know when the game gets to about 3000 characters.

This new method will be called each time a match is loaded, so we'll put it in our `takeTurn` method and our `layoutMatch` method. If the match is getting close to being over, this method will add some information to our `statusLabel`, saying, there's only about 200 characters left.

Add this method to `ViewController.m` right after `dealloc`:

```
-(void)checkForEnding:(NSData *)matchData {
```



```

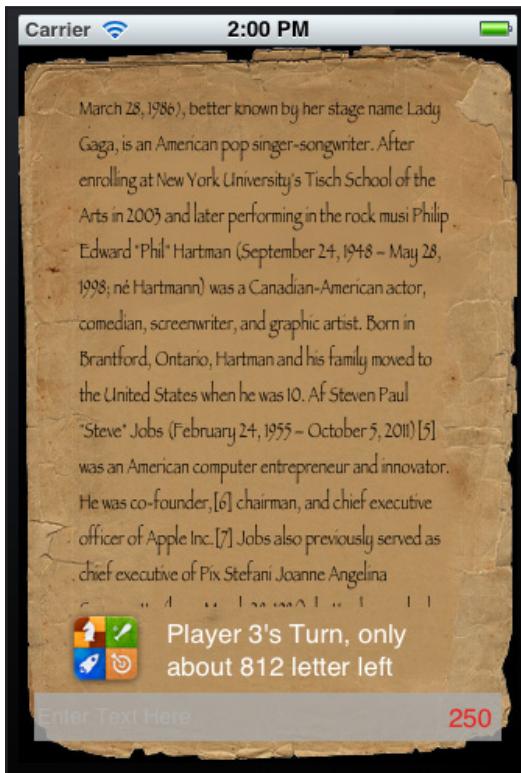
if ([matchData length] > 3000) {
    statusLabel.text = [NSString stringWithFormat:
        @"%@, only about %d letter left", statusLabel.text,
        4000 - [matchData length]];
}
}

```

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 we'll do is end the game whenever the total character count exceeds 3800. The reason is that we don't want the possibility that the NSData string will be larger than 4096, because Game Center won't take the endTurn if the matchData is too large.

Modify sendTurn one last time to the following:

```

- (IBAction)sendTurn:(id)sender {
    GKTurnBasedMatch *currentMatch =
        [[GCTurnBasedMatchHelper sharedInstance] currentMatch];
    NSString *newStoryString;
}

```



```
if ([textInputField.text length] > 250) {
    newStoryString = [textInputField.text substringToIndex:249];
} else {
    newStoryString = textField.text;
}

NSString *sendString = [NSString stringWithFormat:@"%@ %@", mainTextController.text, newStoryString];
NSData *data = [sendString dataUsingEncoding:NSUTF8StringEncoding ];
mainTextController.text = sendString;

NSUInteger currentIndex = [currentMatch.participants
    indexOfObject:currentMatch.currentParticipant];
GKTurnBasedParticipant *nextParticipant;

NSUInteger nextIndex = (currentIndex + 1) %
    [currentMatch.participants count];
nextParticipant = [currentMatch.participants objectAtIndexAtIndex:nextIndex];

for (int i = 0; i < [currentMatch.participants count]; i++) {
    nextParticipant = [currentMatch.participants
        objectAtIndex:(currentIndex + 1 + i) %
        [currentMatch.participants count]]];
    if (nextParticipant.matchOutcome != GKTurnBasedMatchOutcomeQuit) {
        NSLog(@"isn't quit %@", nextParticipant);
        break;
    } else {
        NSLog(@"nex part %@", nextParticipant);
    }
}

if ([data length] > 3800) {
    for (GKTurnBasedParticipant *part in currentMatch.participants) {
        part.matchOutcome = GKTurnBasedMatchOutcomeTied;
    }
    [currentMatch endMatchInTurnWithMatchData:data
        completionHandler:^(NSError *error) {
            if (error) {
                NSLog(@"%@", error);
            }
        }];
    statusLabel.text = @"Game has ended";
} else {

    [currentMatch endTurnWithNextParticipant:nextParticipant
        matchData:data completionHandler:^(NSError *error) {
            if (error) {
                NSLog(@"%@", error);
                statusLabel.text =
                    @"Oops, there was a problem. Try that again.";
            }
        }];
}
```



```

        } else {
            statusLabel.text = @"Your turn is over.";
            textField.enabled = NO;
        }
    }];
}

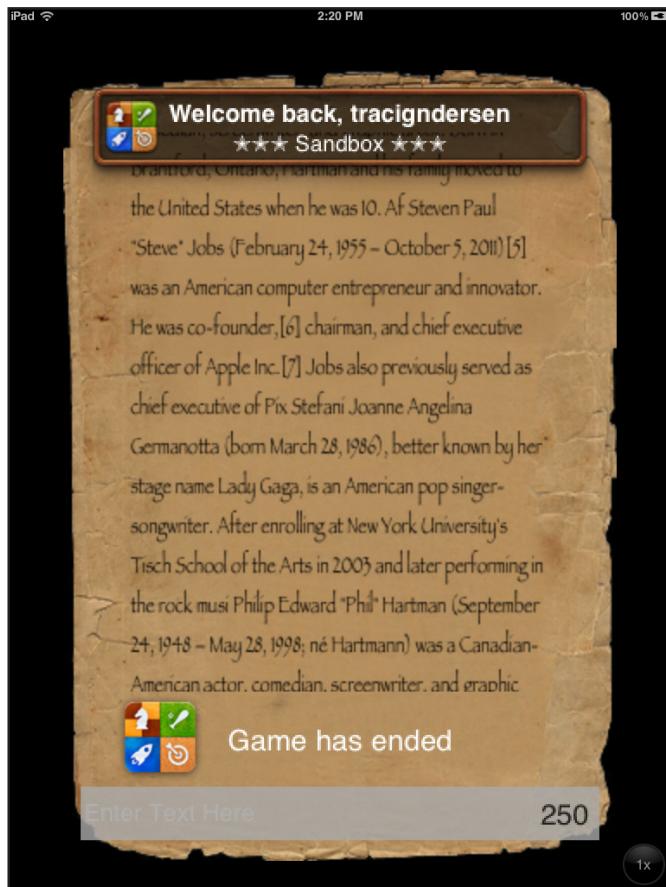
NSLog(@"Send Turn, %@", %@", data, nextParticipant);
textField.text = @"";
characterCountLabel.text = @"250";
characterCountLabel.textColor = [UIColor blackColor];
}

```

We're just wrapping the current endTurn call in an if statement. If we are above 3800 characters, instead of calling endTurn, we'll call endMatch. Once we call end game that notification will be sent to all the other players in the handleMatchEnded notification.

We'll deal with that in a second, but let's give our 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, end in sight. Now we just need to handle the handleMatchEnded in **GCTurnBasedMatchHelper.m**:

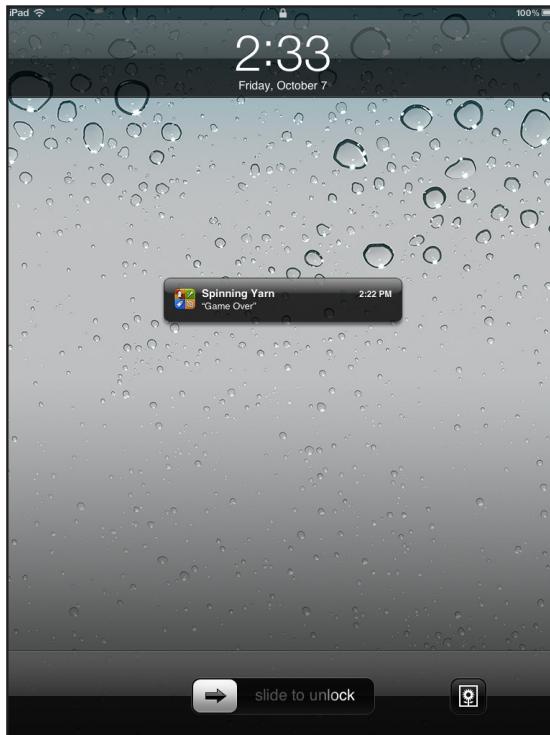
```
-(void)handleMatchEnded:(GKTurnBasedMatch *)match {
    NSLog(@"Game has ended");
    if ([match.matchID isEqualToString:currentMatch.matchID]) {
        [delegate recieveEndGame:match];
    } else {
        [delegate sendNotice:@"Another Game Ended!" forMatch:match];
    }
}
```

If we're looking at another match we'll just send the notice, that spawns the UI-Alert, otherwise we'll send recieveEndGame. Let's implement that as well in **ViewController.m**:

```
-(void)recieveEndGame:(GKTurnBasedMatch *)match {
    [self layoutMatch:match];
}
```

We could do something extra in this case, but because we're setting the statusLabel in layoutMatch this should suffice. However, should we decide later to add something to an endGame notice, like email our story or invite the same group to play again, we could do that here.

Here's what it looks like if you get a game ended notice while the device is locked:



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 boring Turn Based Gaming UI with your own interface programmatically!



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 status about the various games the player is in and make your game more engaging.

In this tutorial, we are going to modify our SpinningYarn app to use a custom user interface for turn-based gaming. We'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.

This tutorial continues the SpinningYarn project where we left it off last time, so open up the project or grab the finished project from the last chapter's resources before continuing.

Getting Started

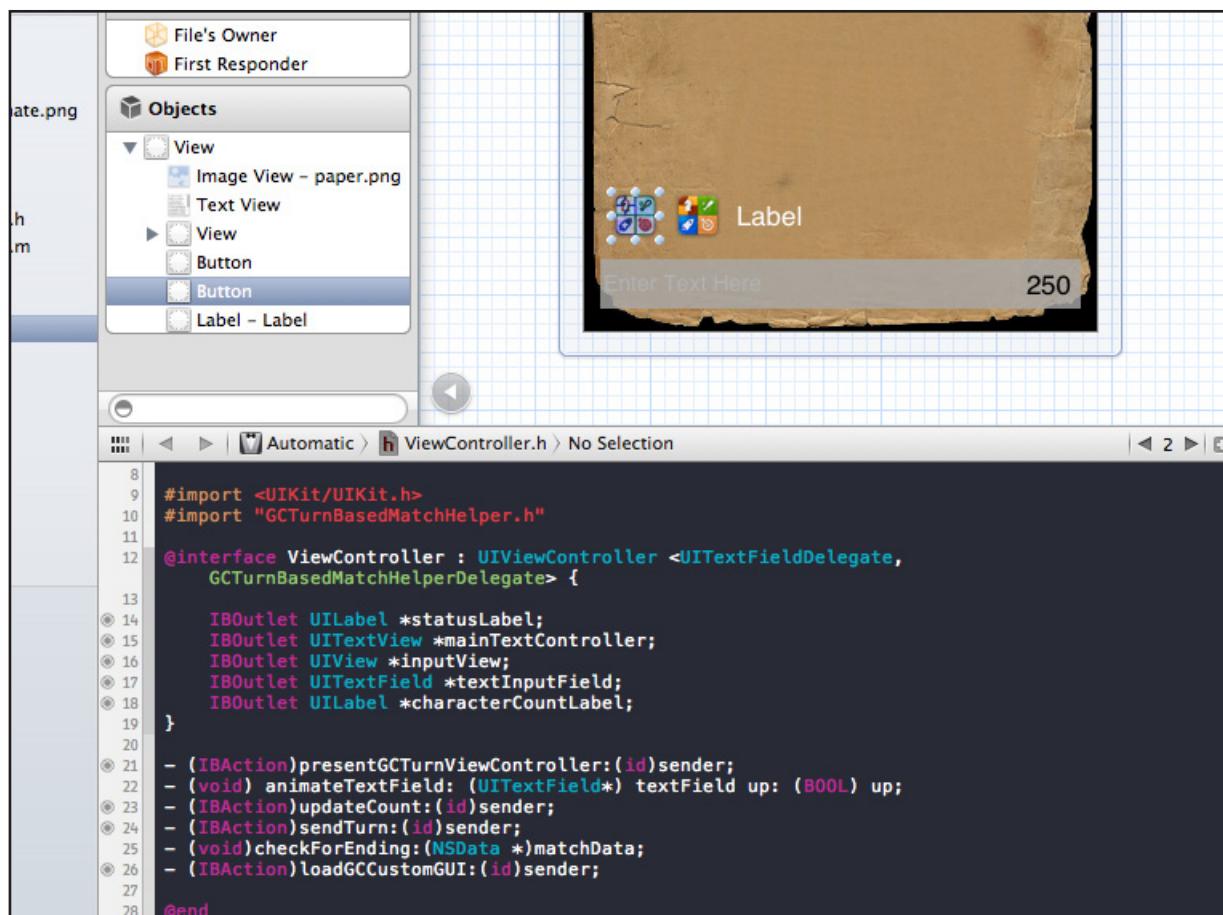
The first thing we need to do is create a new table view controller which we'll be using to display the games we're currently in. So create a new file with the **iOS\ Cocoa Touch\UIViewController subclass** template, enter **GCCustomGUI** for the class name, **UITableViewController** for subclass of, make sure **With XIB for user interface** is checked (but the other is not), and finish creating the file.

The default implementation the template generates for us gives us a blank table view. Let's make a couple small visual tweaks to the table view. Open up **GCCustomGUI.xib** and change the table view style to **Grouped**, the separator to **None**, and the selection to **No Selection**.

For now, let's just hook up this blank table view to a button on the main view controller so we can make sure it shows up OK.



From the resources for this chapter, drag **gameCenterButtonAlternate.png** into your project. Then open up **ViewController.xib** and add a new button with this new image to the left of our current button, as you can see below:



Hint: to set up the new button, set its type to **Custom** in the Attributes Inspector, set the Image to **gameCenterButtonAlternate.png**, and with the button selected choose **Editor\Size to Fit Content** to resize the button to the same size as the image.

Once you've done this, bring up the Assistant Editor and make sure **ViewController.h** is visible. Control-drag from the new button below the @interface, set the Connection type to **Action**, the name to **loadGCCustomGUI**, and click **Connect**.

Fantastic. Let's implement the method we just linked to the button. First add the GameKit header and a forward class predeclaration in **ViewController.h**:

```
#import <GameKit/GameKit.h>
@class GCCustomGUI;
```

Also declare an instance variable for the custom GUI:



```
GCCustomGUI * newGUI;
```

Next switch to **ViewController.m** and add another import to the top of the file:

```
#import "GCCustomGUI.h"
```

Then implement `loadGCCustomGUI` as follows:

```
- (IBAction)loadGCCustomGUI:(id)sender {  
  
    newGUI = [[GCCustomGUI alloc] initWithNibName:@"GCCustomGUI"  
             bundle:nil];  
    UINavigationController *nav =  
        [[UINavigationController alloc] initWithRootViewController:  
            newGUI];  
    [self presentModalViewController:nav animated:YES];  
  
}
```

This code should be familiar to all who have worked with UIKit. We're just creating the new view controller, putting it inside a navigation controller, and presenting it.

Compile and run, and you'll see that now we have a blank table view controller when you tap the new button:



This is a good start for our new GUI, except you'll notice that after you load the new table view, you're stuck and can't dismiss it! So let's add a way to dismiss this controller once we've presented it.



Open **GCCustomGUI.m**, and import GameKit at the top of the file since we will be using that shortly:

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

Then add the following code to the end of viewDidLoad:

```
self.tableView.rowHeight = 220;
self.tableView.editing = NO;

UIBarButtonItem *plus = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
    target:self
    action:@selector(addNewMatch)];
self.navigationItem.rightBarButtonItem = plus;

UIBarButtonItem *cancel = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemCancel
    target:self
    action:@selector(cancel)];
self.navigationItem.leftBarButtonItem = cancel;
```

The first thing we're doing is creating tall table view cells. We'll be displaying a bunch of buttons and info about our matches later on, so we want plenty of space for them.

Next we turn off the editing capability of the table view. We'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.

Next we create two bar buttons for the navigation bar: a plus button (which we'll use to create a new match) and a cancel button (which we'll use to get back to the main screen).

Next we need to add these methods so that this code will run. After viewDidLoad, add the following code:

```
-(void)cancel {
    [self.parentViewController
        dismissModalViewControllerAnimated:YES];
}

-(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!");
        // we'll load the match here
    }
};

}
```

The cancel method is easy, it just dismisses us from the view.

The addNewMatch method might look a little familiar, because it's very close to what we did in the GCTurnBasedMatchHelper's findNewMatchWithMinPlayers:maxPlayers:viewController: method.

Just like we did previously, we create a new GKMatchRequest and specify the number of players we want for the match. But then we do something different - instead of passing the GKMatch to the built-in GKTurnBasedMatchmakerViewController like we did previously, now we call findMatchForRequest to find a match programatically.

Note: according to the current documentation, setting up a match using this method will only allow an auto-match. If you want to invite particular players, you must use the GKTurnBasedMatchmakerViewController.

We then create a block that will be called when a match is found (or an error occurs). Either way, we just log out what happens for now, and later on we'll load the match upon success.

Let's try it out! Compile and run the project, and you should be able to pull up the new navigation controller, dismiss it with the cancel button, and even 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! We want to load all the matches we'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 we're playing. To do this, we're going to call a method named `loadMatchesWithCompletionHandler:` on `GKTurnBasedMatch`. This requests a list of every match our player is involved in, basically the same list of matches that is usually displayed by the built-in `GKTurnBasedMatchmakerViewController`.

Open up **GCCustomGUI.m**, and let's start by creating an array to keep track of our matches. We're gonna use a new technique in iOS 5 that lets us put this into our implementation file, because this data isn't something we'll need access to outside this class. Change the `@implementation` line to:

```
@implementation GCCustomGUI {  
    NSArray *allMyMatches;  
}
```

Pretty slick, eh? :]



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 above viewDidLoad). We're breaking it out into a separate method because we'll call it from other objects (later on) in order to keep our 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];
    }
}
```



```
    }];
```

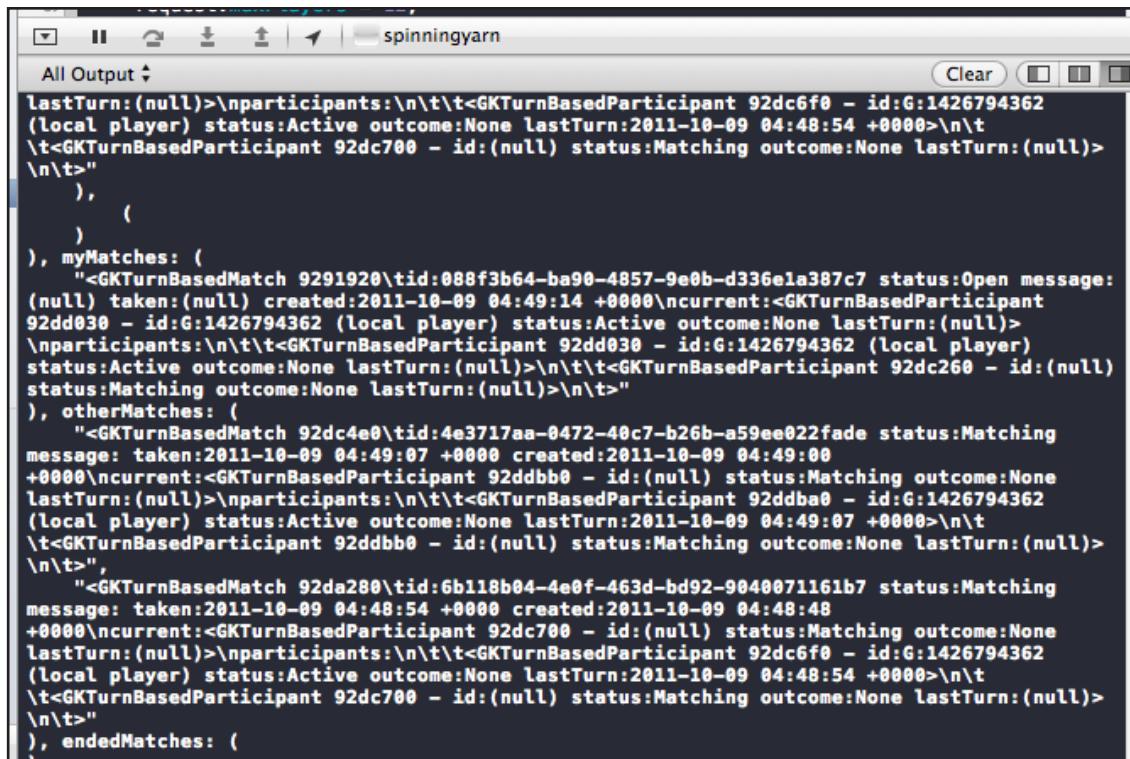
```
}
```

There's a lot of code here, so let's go over it section by section according to the comments.

1. Here we call `loadMatchesWithCompletionHandler` to get all the matches our 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, we first test if there was an error. If not, then we have a list of matches.
3. We want to segment these into three types that will be displayed in three sections of our table view. The first is matches where it's currently our 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. We then launch into a for loop where we iterate over each match. Inside the loop, we start by looking through all of the participants to find the entry for the current player. This way we know what the current player's `matchOutcome` is.
5. Next we can determine which list to put the match in based on whether the game has ended, and whether it's our turn or not.
6. When we're done we put all three into the `allMyMatches` array. We log out the results as well, so that we can follow along and make sure that we're getting the data and structure we think we're getting.

Compile 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!





```

spinningyarn
All Output ▾
Clear (X) (X) (X)
lastTurn:(null)>\nparticipants:\n\t\t<GKTurnBasedParticipant 92dc6f0 - id:G:1426794362
(local player) status:Active outcome:None lastTurn:2011-10-09 04:48:54 +0000\n\t
\t<GKTurnBasedParticipant 92dc700 - id:(null) status:Matching outcome:None lastTurn:(null)>
\n\t>
),
(
)
), myMatches: (
    "<GKTurnBasedMatch 9291920\ntid:088f3b64-ba90-4857-9e0b-d336e1a387c7 status:Open message:
(null) taken:(null) created:2011-10-09 04:49:14 +0000\ncurrent:<GKTurnBasedParticipant
92dd030 - id:G:1426794362 (local player) status:Active outcome:None lastTurn:(null)>
\nparticipants:\n\t\t<GKTurnBasedParticipant 92dd030 - id:G:1426794362 (local player)
status:Active outcome:None lastTurn:(null)>\n\t\t\t<GKTurnBasedParticipant 92dc260 - id:(null)
status:Matching outcome:None lastTurn:(null)>\n\t"
), otherMatches: (
    "<GKTurnBasedMatch 92dc4e0\ntid:4e3717aa-0472-40c7-b26b-a59ee022fade status:Matching
message: taken:2011-10-09 04:49:07 +0000 created:2011-10-09 04:49:00
+0000\ncurrent:<GKTurnBasedParticipant 92ddb0 - id:(null) status:Matching outcome:None
lastTurn:(null)>\nparticipants:\n\t\t<GKTurnBasedParticipant 92dba0 - id:G:1426794362
(local player) status:Active outcome:None lastTurn:2011-10-09 04:49:07 +0000\n\t
\t<GKTurnBasedParticipant 92ddb0 - id:(null) status:Matching outcome:None lastTurn:(null)>
\n\t"
),
"<GKTurnBasedMatch 92da280\ntid:6b118b04-4e0f-463d-bd92-9040071161b7 status:Matching
message: taken:2011-10-09 04:48:54 +0000 created:2011-10-09 04:48:48
+0000\ncurrent:<GKTurnBasedParticipant 92dc700 - id:(null) status:Matching outcome:None
lastTurn:(null)>\nparticipants:\n\t\t<GKTurnBasedParticipant 92dc6f0 - id:G:1426794362
(local player) status:Active outcome:None lastTurn:2011-10-09 04:48:54 +0000\n\t
\t<GKTurnBasedParticipant 92dc700 - id:(null) status:Matching outcome:None lastTurn:(null)>
\n\t"
), endedMatches: (

```

Setting up the Table View

Next we need to implement the table view data source methods so we can display this data to the screen.

Let's start by setting up our sections and their titles. Open up **GCCustomGUI.m** and implement `numberOfSectionsInTableView` and `titleForHeaderInSection` as follows:

```

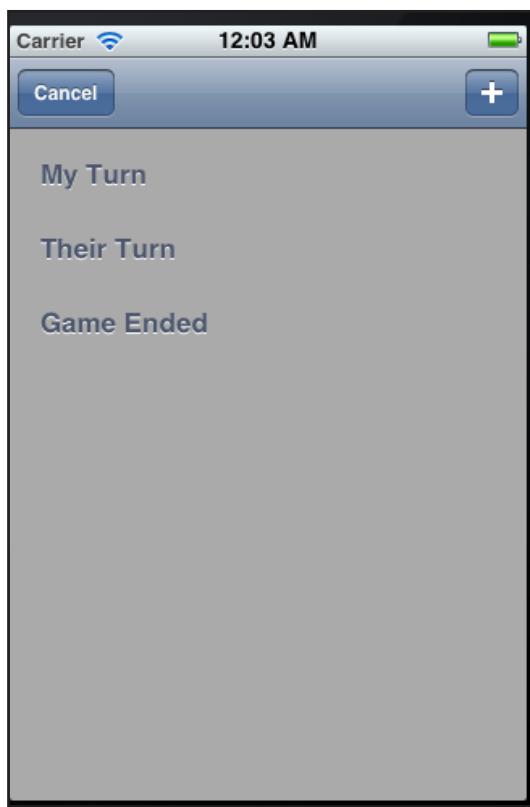
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 3;
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    if (section == 0) {
        return @"My Turn";
    } else if (section == 1) {
        return @"Their Turn";
    } else {
        return @"Game Ended";
    }
}

```



This is pretty simple stuff - we just display a header for each of our arrays. Compile and run to make sure this appears OK.

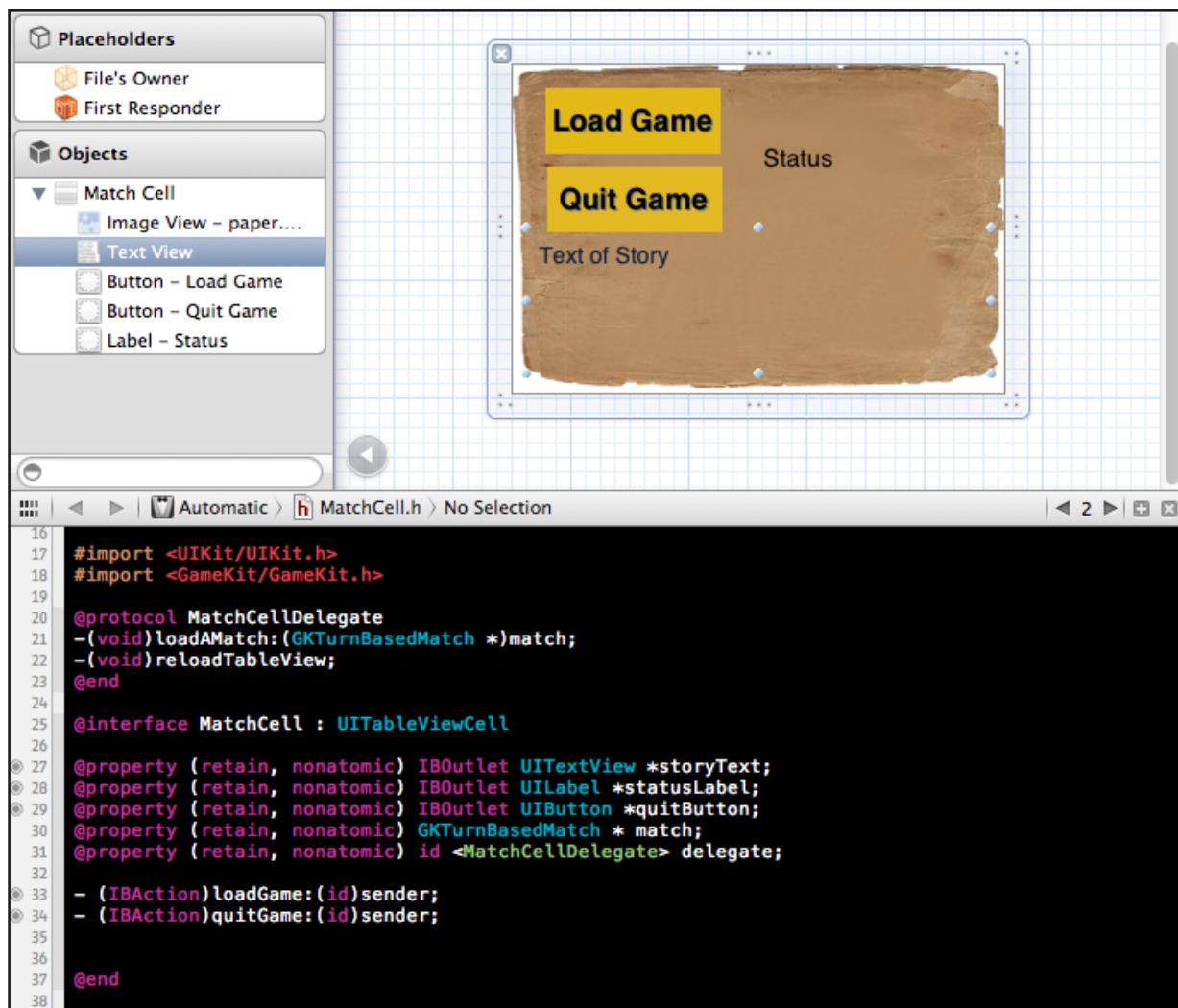


Next, we'll make the table display some custom cells to display the match information in each row.

To save you some time creating the custom cell layout in Interface Builder yourself, I've created the basic layout of the custom cell for you. You will find three files for MatchCell in the resources for this project (MatchCell.h, MatchCell.m, and MatchCell.xib). Drag these files into your project.

Note: If you would like to create this yourself instead of using the one I precreated, here's a screenshot and some info.





Note: This cell has two buttons connected to similarly named methods, load game and quit game. It also has a Label called textLabel that will tell us status information about that match, it also has a textView that will contain an excerpt of the current story. I have connected an outlet to the quitButton because the text will change (If we have already quit that match and it's in a quit list.)

The background uses the same paper image that we are using for our main background it has a delegate protocol and property in order to be able to communicate with other view controllers (without going through the tableview).

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), and the XIB contains the layout for a custom cell we'll be using to display match information.

The next step is to connect some of the GUI elements to outlets. So open up **MatchCell.xib**, bring up the assistant editor and make sure **MatchCell.h** is visible, and do the following:

- Control-drag from the **text view** to below the @interface. Set the connection type to **Outlet**, name it **storyText**, and click Connect.
- Control-drag from the **label** to below the @interface. Set the connection type to **Outlet**, name it **statusLabel**, and click Connect.
- Control-drag from the **Load Game** button to below the @interface. Set the connection type to **Action**, name it **loadGame**, and click Connect.
- Control-drag from the **Quit Game** button to below the @interface. Set the connection type to **Action**, name it **quitGame**, and click Connect.
- Control-drag from the **Quit Game** button to below the @interface (again). This time set the connection type to **Outlet**, name it **quitButton**, and click Connect.

Next, open up **MatchCell.h** and import some headers we'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;
```

Finally, switch to **MatchCell.m** and synthesize the property:

```
@synthesize match;
```

Now that we've got some hooks into our cell, we can import the MatchCell class into our GCCustomGUI class and start using it!

Open up **GCCustomGUI.m** and import MatchCell.h at the top of the file:

```
#import "MatchCell.h"
```

Then edit the following two methods:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return [[allMyMatches objectAtIndex:section] count];
}

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



```

    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    // 1
    if (cell == nil) {
        NSArray *topLevelObjects = [[NSBundle mainBundle]
            loadNibNamed:@"MatchCell" owner:self options:nil];
        cell = [topLevelObjects objectAtIndex:0];
    }

    // 2
    GKTurnBasedMatch *match =
        [[allMyMatches objectAtIndex:indexPath.section]
            objectAtIndex:indexPath.row];
    MatchCell *c = (MatchCell *)cell;
    c.match = match;
    // 3
    if ([match.matchData length] > 0) {
        // 4
        NSString *storyString =
            [NSString stringWithUTF8String:[match.matchData bytes]];
        c.storyText.text = storyString;
        // 5
        int days = -floor([match.creationDate timeIntervalSinceNow] /
            (60 * 60 * 24));
        cStatusLabel.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, we just return the count for the appropriate array.

In `cellForRowAtIndexPath` we'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. Here we use the `loadNibName:owner:options` method to load the `MatchCell` XIB on demand and create the objects inside. We get the first object from the array, which is the table view cell inside.
2. Here we get the match that corresponds to the section and row position for this cell (see how easy that is because our array structures match the cell `indexPath` structure!) Then we assign the match to the cell's `match` property. We're going to use this later, to quit matches and load matches.



3. Next we 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), we don't display anything.
4. If there is match data, we set the text of the `storyText` text view to the string data in the `NSData` property of the match. We can then see the first few lines of each story, which gives us a much better sense for where we are in the story.
5. We also construct a `statusLabel` string. We want to tell the user how long it's been since the match was started. We use the `match.creationDate` and a method called `timeIntervalSinceNow`. That method tells us how many seconds have passed between a date value and now. We 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. We 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 we use this information to create a status string.

That's it! Compile and run, and you should have custom cells full of match information:



Our 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.

We need a way to dismiss the the entire navigation controller from the table view controller. There are a number of ways to do this, but we're going to use an approach similar to what we did in our GCTurnBasedMatchHelper class. We're going to create an instance variable that will reference our ViewController so we 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, retain) ViewController * vc;
```

Then switch to **GCCustomGUI.m** and synthesize the new property:

```
@synthesize vc;
```

Next, find the addNewMatch method, and add the following code after the "we'll load the match here" comment:

```
[vc dismissModalViewControllerAnimated:YES];  
[[GCTurnBasedMatchHelper sharedInstance]  
 turnBasedMatchmakerViewController:nil didFindMatch:match];
```

Here's where that reference becomes useful. We first dismiss the modelViewController (which is the class we're in), and then we call turnBasedMatchmakerViewController:didFindMatch on the GCTurnBasedMatchHelper.

The reason we 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, we want to make sure the singleton stays up to date, because when we send a turn, it's going to ask the singleton for the currentMatch in order to send the next turn.

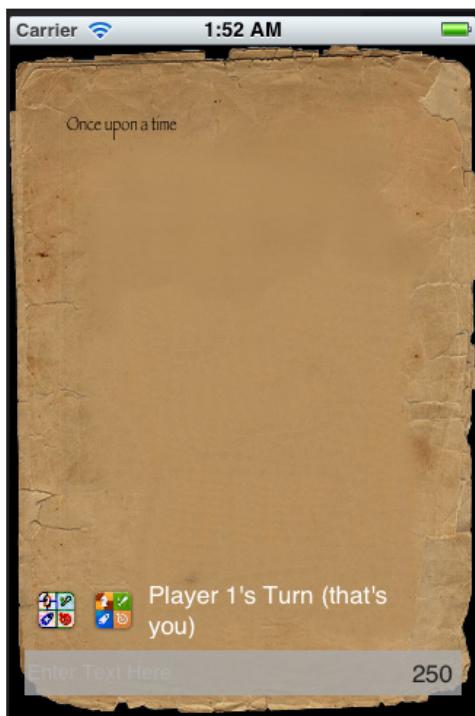
By pretending that we are just like the provided GKTurnBasedMatchmakerViewController, we save ourselves trying to replicate all the logic that we built up in the first part of this tutorial. We're just sending nil in for the viewController here because we are taking care of dismissing ourselves with the first line.



We need to set the vc property before this is going to work. Switch to **ViewController.m**, find the `loadGCCustomGUI` method, and add this line before the `[self presentModalViewController:nav animated:YES];` line:

```
newGUI.vc = self;
```

There, we 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 our "Load Game" button in our table view cell, so we can load up an existing game.

To keep things clean, we are going to create a delegate protocol in our `MatchCell` so that we 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
```



Also add a property for the delegate:

```
@property (retain, nonatomic) id <MatchCellDelegate> delegate;
```

Then switch to **MatchCell.m** and synthesize the delegate:

```
@synthesize delegate;
```

Also, implement the loadGame method to call the method on the delegate:

```
- (IBAction)loadGame:(id)sender {
    [delegate loadAMatch:match];
}
```

Next we need to set the delegate when we create the MatchCell. Open up **GCCustomGUI.h** and mark the class as implementing MatchCellDelegate and import the MatchCell header:

```
#import "MatchCell.h"

@interface GCCustomGUI : UITableViewController <MatchCellDelegate>
```

Then switch to **GCCustomGUI.m**, and add the following inside tableView:cellForRowAtIndexPath, after the line c.match = match:

```
c.delegate = self;
```

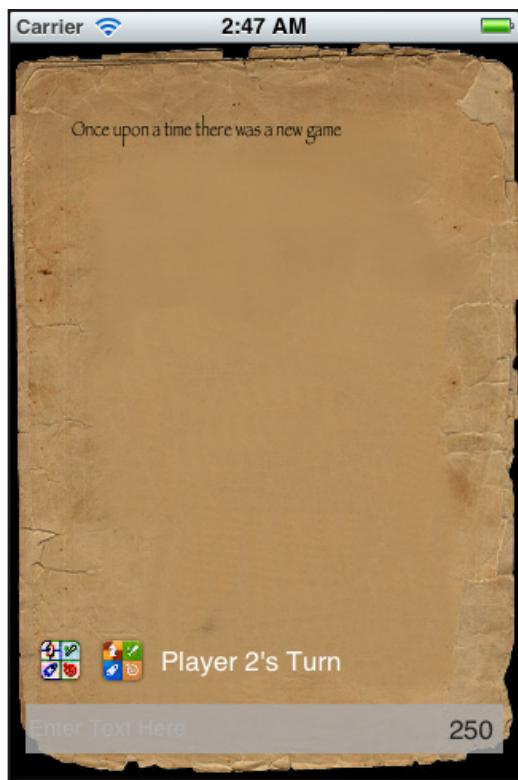
The last step is to implement the delegate method as follows:

```
-(void)loadAMatch:(GKTurnBasedMatch *)match {
    [vc dismissModalViewControllerAnimated:YES];
    [[GCTurnBasedMatchHelper sharedInstance]
        turnBasedMatchmakerViewController:nil didFindMatch:match];
}
```

Here we dismiss the view controller and notify the GCTurnBasedMatchHelper we found a match, like we 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 we dive into code, we 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 we might need to update when we quit a match.

The `GKTurnBasedParticipant` has a `status` property and an `outcome` property that may or may not be set by the method we use to quit. Also, if our match has all but one player who has quit, we need to change the `status` property of the `GKTurnBasedMatch`. We 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 we can use to quit from a match, the first two are `participantQuitInTurnWithOutcome:nextParticipant:` and `participantQuitOutOfTurnWithOutcome:..`. Both of these methods will set the participant status to quit and the outcome to whatever we pass in as an argument to the method.

The other method available to us is the `removeWithCompletionHandler:` method. This method will remove us from the match, but it doesn't set any of these flags. This method should only be used to remove a match once we have already quit.

None of the methods set the match's status. If we are the second to last player to quit the match, we'll need to be responsible and end the match. So we'll need to do some checking in order to see if this is necessary.

If we quit when it's our turn, we can use the logic already in the GCTurnBasedMatchHelper, but if not our turn, we'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 ([match.currentParticipant.playerID isEqualToString:
        [GKLocalPlayer localPlayer].playerID]) {           //1
        [[GCTurnBasedMatchHelper sharedInstance]
            turnBasedMatchmakerViewController:nil playerQuitForMatch:match];
    } else {                                         //2
        [match participantQuitOutOfTurnWithOutcome:
            GKTurnBasedMatchOutcomeQuit
            completionHandler:^(NSError *error) {
                if (error) {
                    NSLog(@"%@", error.localizedDescription);
                }
            }];
    }                                               //3
    int partsStillActive = 0;
    for (GKTurnBasedParticipant *participant in match.participants) {
        if (participant.matchOutcome == GKTurnBasedMatchOutcomeNone) {
            partsStillActive++;
        }
    }                                               //4
    if (partsStillActive < 2) {
        for (GKTurnBasedParticipant *participant in match.participants) {
            if (participant.matchOutcome == GKTurnBasedMatchOutcomeNone) {
                participant.matchOutcome = GKTurnBasedMatchOutcomeTied;
            }
        }
        [match endMatchInTurnWithMatchData:match.matchData
            completionHandler:^(NSError *error) {
                if (error) {
                    NSLog(@"%@", error);
                }
            }];
    }                                               //5
    [delegate reloadTableView];
}
```



Let's go through each of the code sections one at a time. The first marked method checks to see if we are quitting a match where it's our turn. If it is, we need to do some things (like pass on the baton to the next player), so we'll use the method we already created in the GCTurnBasedMatchHelper.

The second section fires if it's not our turn. In this case all we really need to do is call the quit method and give ourselves a match outcome. We're giving ourselves the quit outcome. The completion handler just prints out any error messages that may come back.

The next section iterates through all the participants and checks for the outcome. If it isn't set to None, that means that the participant is no longer playing in the match. We'll count up how many participants still have an outcome of None. If this is less than two than we don't have enough active players to continue in any meaningful way.

In the next section we will deal with that scenario. If there are less than two players left, we'll iterate through them (there will only be one, but whatever) and assign them the outcome of tied. Then we call the same endMatchInTurnWithMatchData:completionHandler: we called in our sendTurn method earlier. This ends the game and all the participants will get a notification.

Finally, we need to update our GUI. That last method we call is the same that we built to load our GUI in the first place, I told you there was a reason to break it out into its own method. We 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
```

Compile 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 we need to do is change the button for the matches that we've quit. We don't want to be quitting the same match multiple times, but we do want to be able to remove matches we've quit from so that we don't have a huge list of matches that we no longer care about.

The first thing to do is go back to our `cellForRowAtIndexPath` method in **GCCustomGUI.m**, and add the following right before the call to `return cell`:

```
if (indexPath.section == 2) {
    [c.quitButton setTitle:@"Remove" forState:UIControlStateNormal];
    [c.quitButton setTitle:@"Remove" forState:UIControlStateNormal];
}
```

If we're in this third section, then we know that all these games are matches that have ended. We want the button to change its title to Remove when it's in the third section.

Alright, but we need to go back and deal with this button press. Switch to **MatchCell.m** and modify `quitGame` so it calls a different method if we're in the Remove state:

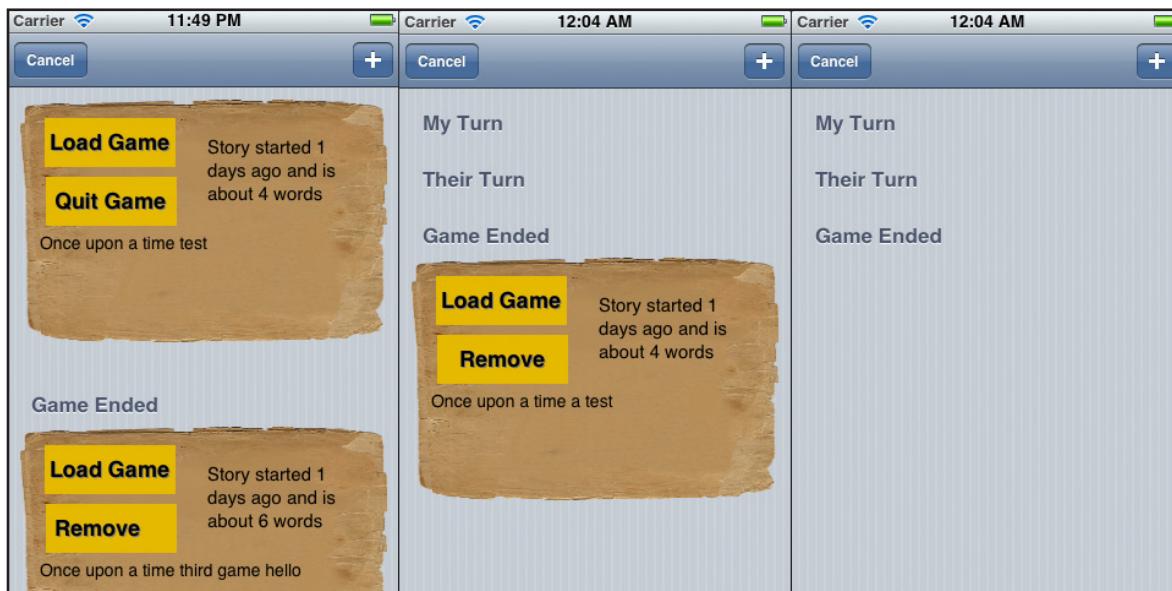


```
- (IBAction)quitGame:(id)sender {
    UIButton *send = (UIButton *)sender;
    if ([send.titleLabel.text isEqualToString:@"Remove"]) {
        NSLog(@"remove, %@", send.titleLabel.text);
        [match removeWithCompletionHandler:^(NSError *error) {
            [delegate reloadTableView];
        }];
    } else {
        // rest of code...
    }
}
```

We're checking the button title to see if we need to remove or quit. If it's titled remove, then we can use the removeWithCompletionHandler method to get rid of the match. The remove method prevents the match from being retrieved when we 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 our quitGame method should be included in the else bracket. We want to do this first block or the block of code we were just looking at, not both. We're putting the reload method in the completion handler, because if it's called too early we'll still get the removed match back in our 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 we 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, our GUI should reload to reflect the most recent state. Secondly, we should add some buttons to the alertview so that we can either dismiss it, load the match that just received and update, or load our GUI view.

The first thing we need is a way to check if our GUI is loaded or not.

Switch to **GCCustomGUI.h** and add the following declaration:

```
- (BOOL)isVisible;
```

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 our 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 we call this by itself, it will load the view into memory if it's already loaded. By using both we can avoid the extra overhead. We can thank StackOverflow for this goodness.

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 ([newGUI isVisible]) {
        [newGUI 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];
        [av release];
    }
}
```



This first part just checks if the GUI is up and if it is, we call `reloadTableView` on it.

Next, we are setting a new value, alternate match. We are going to need a reference to this other match if the user presses the 'Let's see it!' button. Set up that variable now in **GCTurnBasedMatchHelper.h**:

```
@property (nonatomic, retain) GKTurnBasedMatch *alternateMatch;
```

Then synthesize this new property in **GCTurnBasedMatchHelper.m**:

```
@synthesize alternateMatch;
```

The next part we're adding a few extra items to the `otherButtonTitles`, 'Let's see it!' and 'All my Matches'.

The other change we've made is setting the delegate to `self`, so we need to declare our ViewController as a class that implements `UIAlertViewDelegate` protocol. Change the class declaration in **ViewController.h** to the following:

```
@interface ViewController : UIViewController <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 loadGCCustomGUI:nil];
            break;
        default:
            break;
    }
}
```



```
    }  
}
```

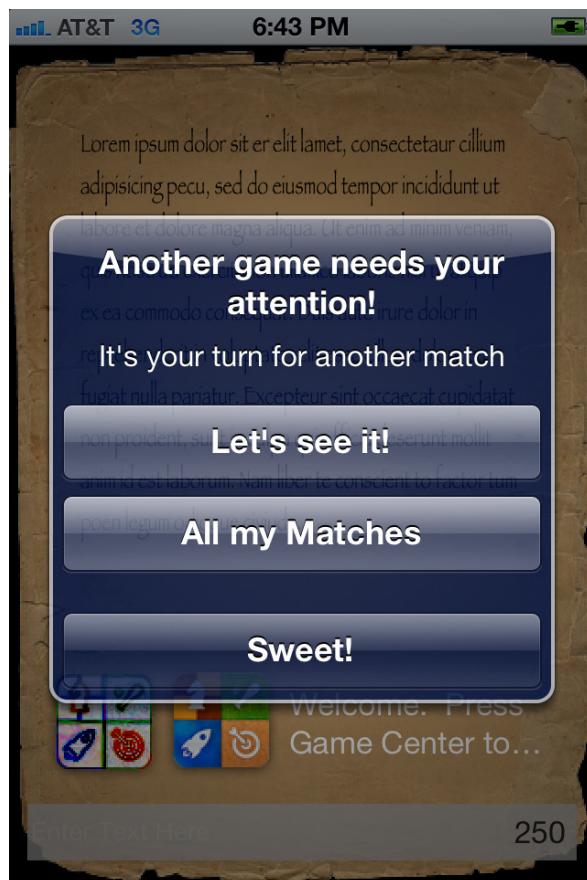
This method gets passed the buttonIndex value which in our case is a value of 0 if the "Sweet!" (Cancel) is pressed, a value of 1 if the "Let's See It!" is pressed, and 2 if the "All my Matches" is pressed.

We handle the cancel button the same as we did before, it simply dismisses the alertview. We don't have do handle it in the switch.

If we decide we want to load the notified match, we'll first set the current match to the alternate match. Then we pass the appropriate method to ViewController object. If it's our turn we use the takeTurn method, if not we use layoutMatch.

If the user pressed "All my Matches", we call the loadGCCustomGUI method, this will act just like when we push our button.

If you build and run now, play a few turns, you'll be given options to load your games:



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.

We 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!



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 it's own parameters and can be queried in code to provide information about the filter, it's 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. We'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 we 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 hold the image data. You can create an instance from a UIImage, from an image file, or from 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.

We'll be using each of these classes as we create our project.

Getting Started

Open up Xcode and create a new project with the **iOS\Application\Single View Application** template. Enter **CI Test** for the Product Name, select **iPhone** for the device family, and make sure that **Use Automatic Reference Counting** is 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. 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.

From the resources for this chapter, add **image.png** to your project. I like to put all the images and sound files into a **Resources** group. Your project won't have that grouping. So, if you want to (it's not necessary), control-click the **image.png** file once it's added and choose **New Group from Selection**. Then click on the folder name to change it to **Resources**.

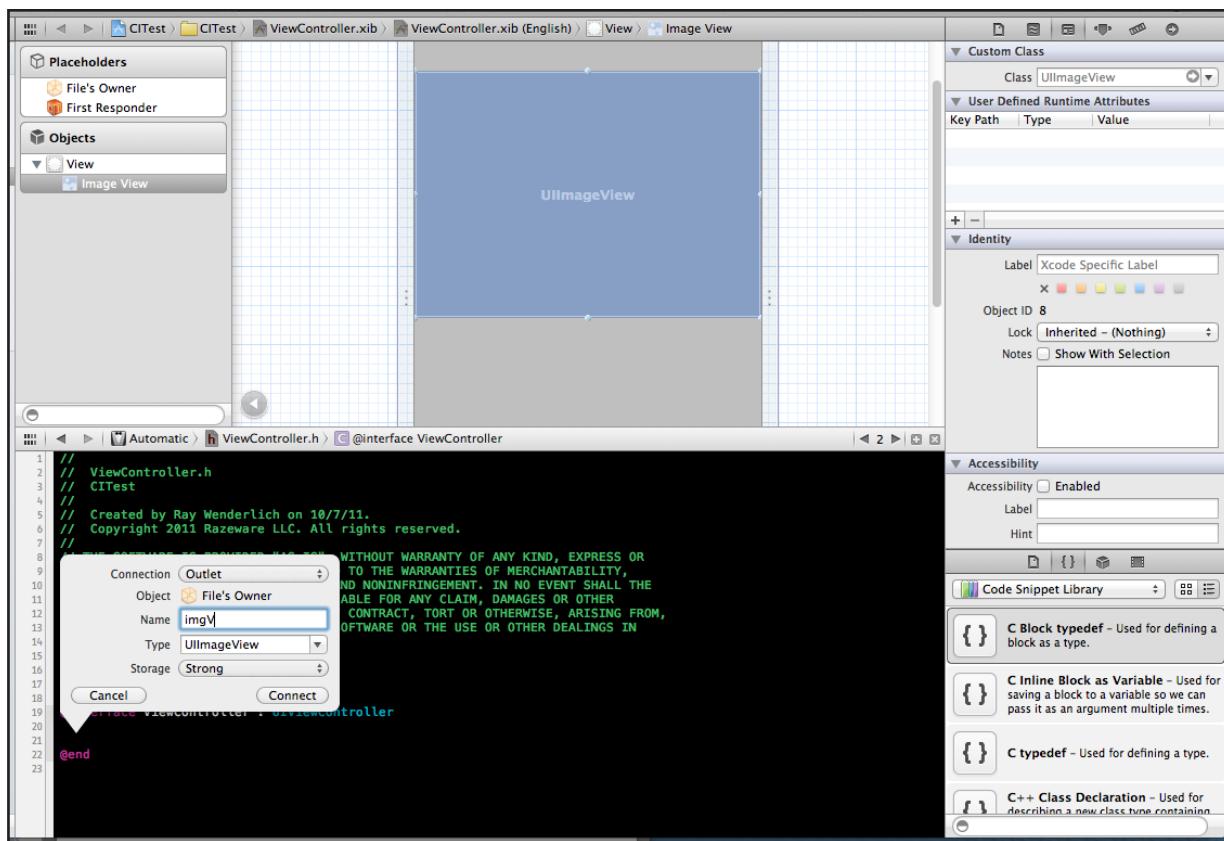
Now we're going to hide the status bar. We need to add a key to our plist, add a line of code, and change a setting in the xib to accomplish this.

First the plist. Open **SupportingFiles\CI Test-Info.plist**, control-click anywhere in the white space, and select **Add Row**. Change the value to **Status bar is initially hidden** and set the value to **YES**.

Next the .xib file. Open **ViewController.xib** and highlight the view. Bring up the attributes inspector, and change the **Status Bar** value to **None**. With the statusbar removed, we can add the 20 pixels that the statusbar takes up back in. In the size inspector, change the height of the view from 460 to 480.

While we're in the .xib file, let's add a UIImageView to our view object. Drag and drop one from the objects panel. 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 **imgV**, and click Connect.

Finally, we'll add some code to `AppDelegate.m`. In the `applicationDidFinishLaunchingWithOptions` method add the following line before the return YES line:

```
[UIApplication sharedApplication] setStatusBarHidden:YES];
```

Run your project, and you should see a plain gray screen with no status bar. The initial setup is complete - now onto Core Image!

Basic Image Filtering

We're going to get started by simply running our image through a `CIFilter` and displaying it on the screen.

Every time we want to apply a `CIFilter` to an image we 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.
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 `CIIImage` - 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`:

```
NSString *filePath =
[[NSBundle mainBundle] pathForResource:@"image" ofType:@"png"];
NSURL *fileNameAndPath = [NSURL fileURLWithPath:filePath];

CIIImage *beginImage =
[CIIImage imageWithContentsOfURL:fileNameAndPath];
CIContext *context = [CIContext contextWithOptions:nil];

CIFilter *filter = [CIFilter filterWithName:@"CISepiaTone"
keysAndValues: kCIIInputImageKey, beginImage,
@"inputIntensity", [NSNumber numberWithFloat:0.8], nil];
CIIImage *outputImage = [filter outputImage];

CGImageRef cgimg =
[context createCGImage:outputImage fromRect:[outputImage extent]];
UIImage *newImg = [UIImage imageWithCGImage:cgimg];

[imgV setImage:newImg];

CGImageRelease(cgimg);
```

The first two lines create an `NSURL` object that holds the path to our image file.

Next we create our `CIIImage` with the `imageWithContentsOfURL` method and create the `CIContext`. 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 we pass in `nil` for that argument.

Next we'll create our `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 `CISepiaTone` filter takes only two values, the `KCIIInputImageKey` (a `CIIImage`) and the `@"inputIntensity"`, a float value, wrapped in an `NSNumber`, between 0 and 1. Here we give that value 0.8. Most of the filters have default values that will be used



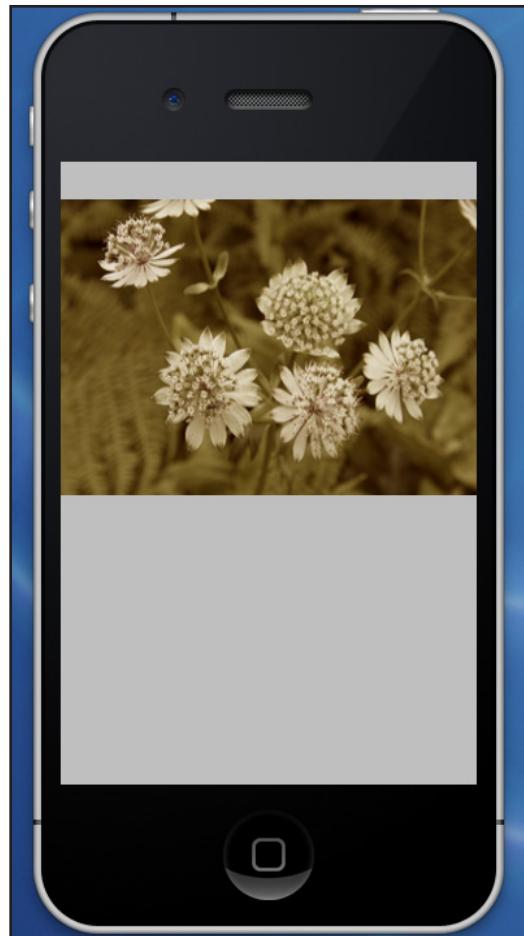
if no values are supplied. One exception is the `CIImage`, this must be provided as there is no default.

Getting a `CIImage` back out of a filter is easy - just use the `outputImage` property.

Once we have an output `CIImage`, we will need to convert it into a `CGImage` (which we can then convert to a `UIImage` or draw the screen directly). This is where the context comes in. Calling the `createCGImage:fromRect:` on the context with the supplied `CIImage` will produce a `CGImageRef`. This can then be used to create a `UIImage` by calling the `imageWithCGImage` constructor.

Once we've converted it to a `UIImage`, we just display it in the image view we added earlier.

Compile and run the project, and you'll see our image filtered by the sepia tone filter. Congratulations, you have successfully used `CIImage` and `CIFilters`!



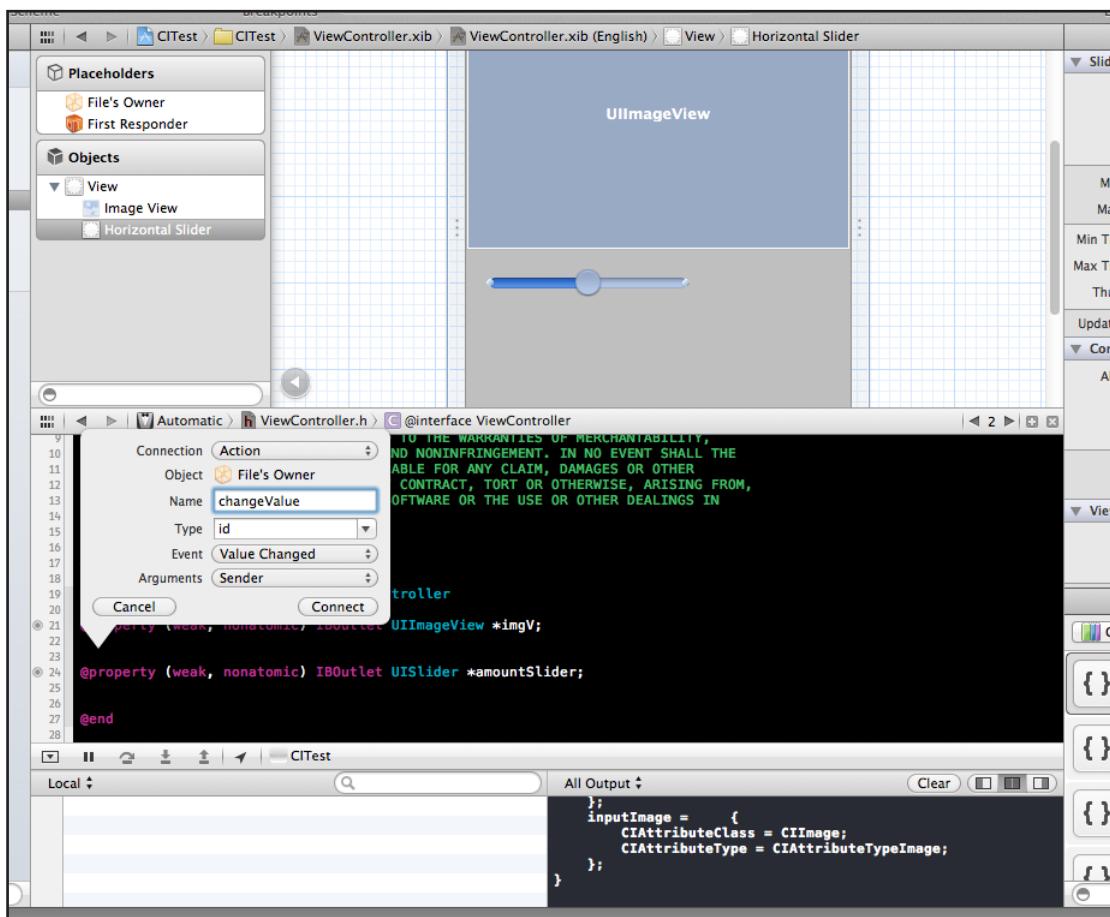
Changing Filter Values

This is great, but this is just the beginning of what you can do with Core Image filters. Lets add a slider and set it up so we can adjust the image settings in real time.

Open up **ViewController.xib** and drag a slider in below the image view. 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 **changeValue**, 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.

The screen layout should look sort of like this:



We need to recreate parts of the process every time we move the slider. However, we don't want to redo the whole process, that would be very inefficient and would



take too long. We'll need to change a few things in our class so that we hold on to some of the objects we create in our viewDidLoad method.

The biggest thing we want to do is reuse the CIContext whenever we need to use it. If we recreate it each time, our program will run very slow. The other things we can hold onto are the CIFilter and the CIImage that holds our beginning image. We'll need a new CIImage for every output, but the image we start with will stay constant.

We 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",  
    [NSNumber numberWithFloat:0.8], nil];
```

Now we'll implement the changeValue method. What we'll be doing in this method is altering the value of the @"inputIntensity" key in our CIFilter dictionary. Once we've altered this value we'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 changeValue method with the following:

```
-(IBAction)changeValue:(UISlider *)sender {  
    float slideValue = [sender value];  
  
    [filter setValue:[NSNumber numberWithFloat:slideValue]  
        forKey:@"inputIntensity"];  
    CIImage *outputImage = [filter outputImage];
```



```
CGImageRef cgimg = [context createCGImage:outputImage  
    fromRect:[outputImage extent]];  
  
UIImage *newImg = [UIImage imageWithCGImage:cgimg];  
[imgV setImage:newImg];  
  
CGImageRelease(cgimg);  
}
```

You'll notice that we've changed the variable type from `(id)sender` to `(UISlider *)sender` in the method definition. We know we'll only be using this method to retrieve values from our `UISlider`, so we can go ahead and make this change. If we'd left it as `(id)`, we'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 we've made here.

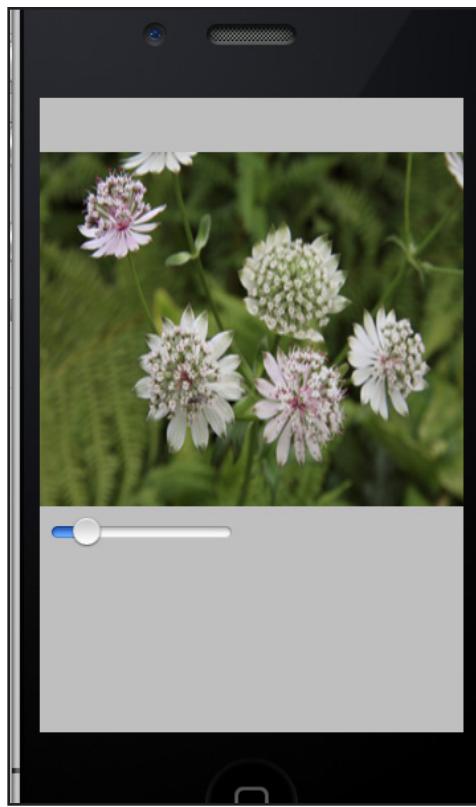
We retrieve the float value from the slider. Go back to Interface Builder and make sure the slider is set to min 0, max 0, and default 0.5. These happen to be the right values for this `CIFilter`.

The `CIFilter` has methods that will allow us to set the values for the different keys in its dictionary. Here, we're just setting the @"`inputIntensity`" to an `NSNumber` object with a float value of whatever we get from our slider.

The rest of the code should look familiar, as it follows the same logic as our `viewDidLoad` method. We're going to be using this code over and over again. From now on, we'll use the `changeSlider` method to render the output of a `CIFilter` to our `UIImageView`.

Compile and run, and you should have a functioning live slider that will alter the sepia value for our image in real time!



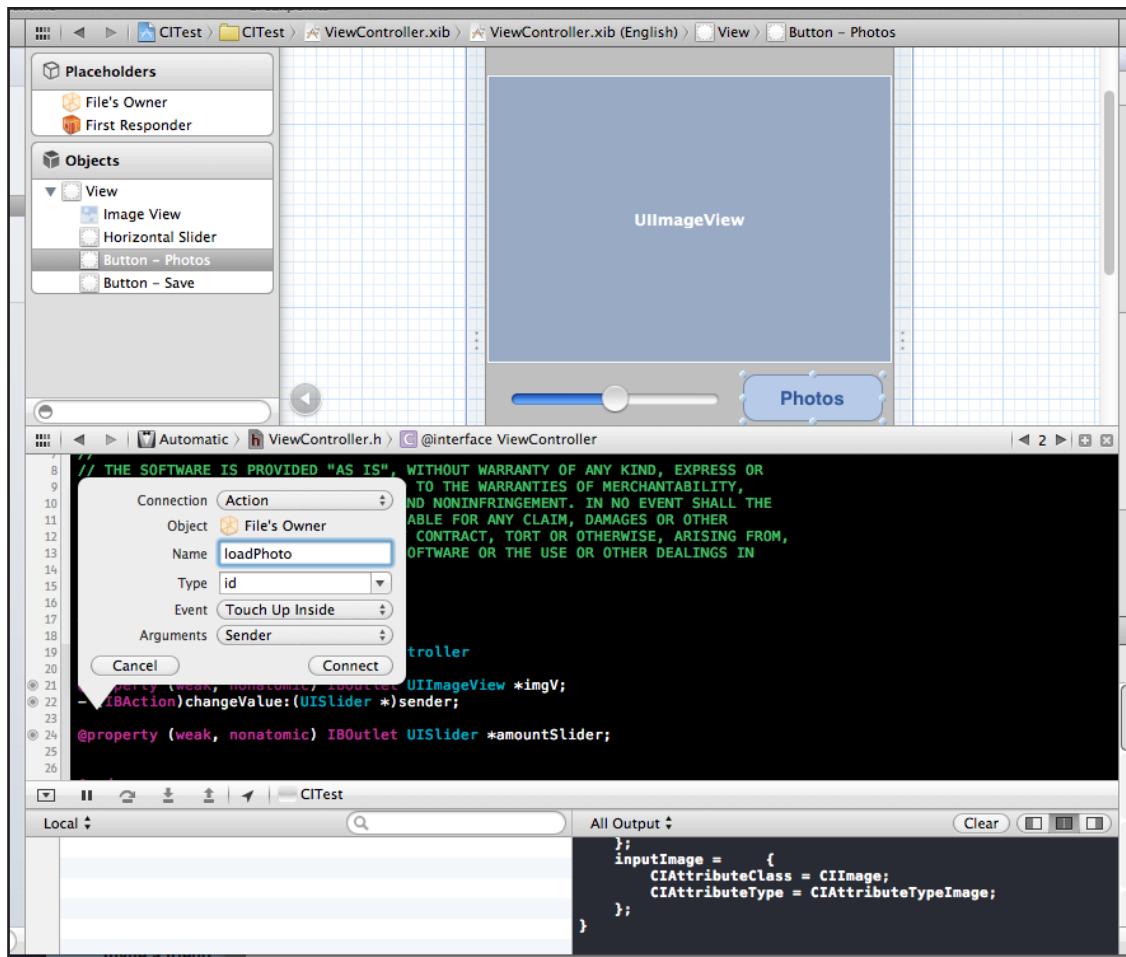


Getting Photos from the Photo Album

Now that we can change the values of the filter on the fly, things are starting to get real interesting! But what if we don't care for this image of flowers? Let's set up a `UIImagePickerController` so we can get pictures from out of the photo album and into our program so we can play with them.

We need to create a button that will bring up the photo album view, so open up `ViewController.xib` and drag in a button to the right of the slider.

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];
}

```

The first line of code instantiates a new UIImagePickerController. We then set the delegate of the image picker to self (our ViewController).

We get a warning here. We need to setup our ViewController as an UIImagePickerControllerDelegate and UINavigationControllerDelegate and then implement the methods in that delegate's protocol. In **ViewController.h**, change the interface line like so:

```

@interface ViewController : UIViewController
<UIImagePickerControllerDelegate, UINavigationControllerDelegate>

```

Now implement the following two methods:

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info {
    [self dismissModalViewControllerAnimated:YES];
    NSLog(@"%@", info);
}

- (void)imagePickerControllerDidCancel:
(UIImagePickerController *)picker {
    [self dismissModalViewControllerAnimated:YES];
}
```

In both cases, we dismiss the UIImagePickerController. That's the delegate's job, if we don't do it there, then we 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.

Compile and run and tap the button, and it will bring up the image picker with the photos in your photo album. If you are running this in the simulator, you probably won't get any photos. On the simulator or on a device without a camera, 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 to save that image. Next time you run our app, you'll have it!

Here's what you should see in the console after you've selected an image (something like this):

```
[This GDB was configured as x86_64-apple-darwin11.2.0]
[Switching to process 22762 thread 0x1300b]
[Switching to process 22762 thread 0xf803]
2011-10-04 23:28:44.453 CITest[22762:f803] {
    UIImagePickerControllerMediaType = "public.image";
    UIImagePickerControllerOriginalImage = "<UIImage: 0xce55330>";
    UIImagePickerControllerReferenceURL = "assets-library://asset/asset.JPG?id=633CD9D7-
ED71-410E-906E-56F84CE444C3&ext=JPG";
}
```

Note that it has an entry in the dictionary for the "original image" selected by the user. This is what we want to pull out and filter!

Now that we've got a way to select an image, how do we set our CIImage beganImage to use that image?

Simple, just change the delegate method to look like this:

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info {
```



```
[self dismissModalViewControllerAnimated:YES];
UIImage *gotImage =
[info objectForKey:UIImagePickerControllerOriginalImage];
beginImage = [CIImage imageWithCGImage:gotImage.CGImage];
[filter setValue:beginImage forKey:kCIColorInputImageKey];
[self changeValue:amountSlider];
}
```

We need to create a new CIImage from our selected photo. We 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 the UIImagePickerController Delegate Protocol Reference.

We need to convert this into a CIImage, but we don't have a method to convert a UIImage into a CIImage. However, we do have [CIImage imageWithCGImage:] method. We can get a CIImage from our UIImage by calling UIImage.CGImage, so we do exactly that!

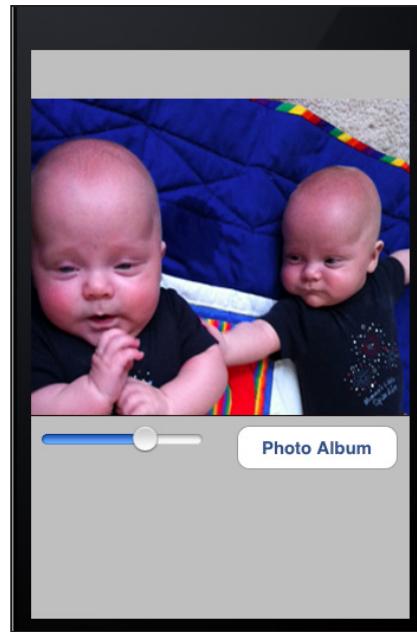
We then set the key in the filter dictionary so that the input image is our new CIImage we just created.

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 we need to do that again, so we can just call the changeValue method. Even though the slider value hasn't changed, we can still use that method's code to get the job done. We could break that code into its own method, and if we were going to be working with more complexity, we would to avoid confusion. But, in this case our purpose here is served using the changeValue method. We pass in the amountSlider so that it has the correct value to use.

Compile and run, and now you'll be able to update the image from your photo album!





What if we create the perfect sepia image, how do we hold on to it? We could take a screenshot, but we're not that ghetto. Let's learn how to save our photos back to the photo album.

Saving to Photo Album

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 we switch from one app to another. If the photo isn't finished being saved, it won't be there when we go looking for it later!

The solution to this is to use the CPU CIRendering context. The default is the GPU, so we need to make a few changes in order to have the CIContext use the CPU. The other advantage of using the CPU is that the CPU is not limited to a texture size (the GPU is limited, use `-inputImageMaximumSize` and `-outputImageMaximumSize` to find out on the current device). The GPU gives better performance, so for photos it's often better to use the CPU (single image is easier than a stream of images), for video the GPU is the better choice.

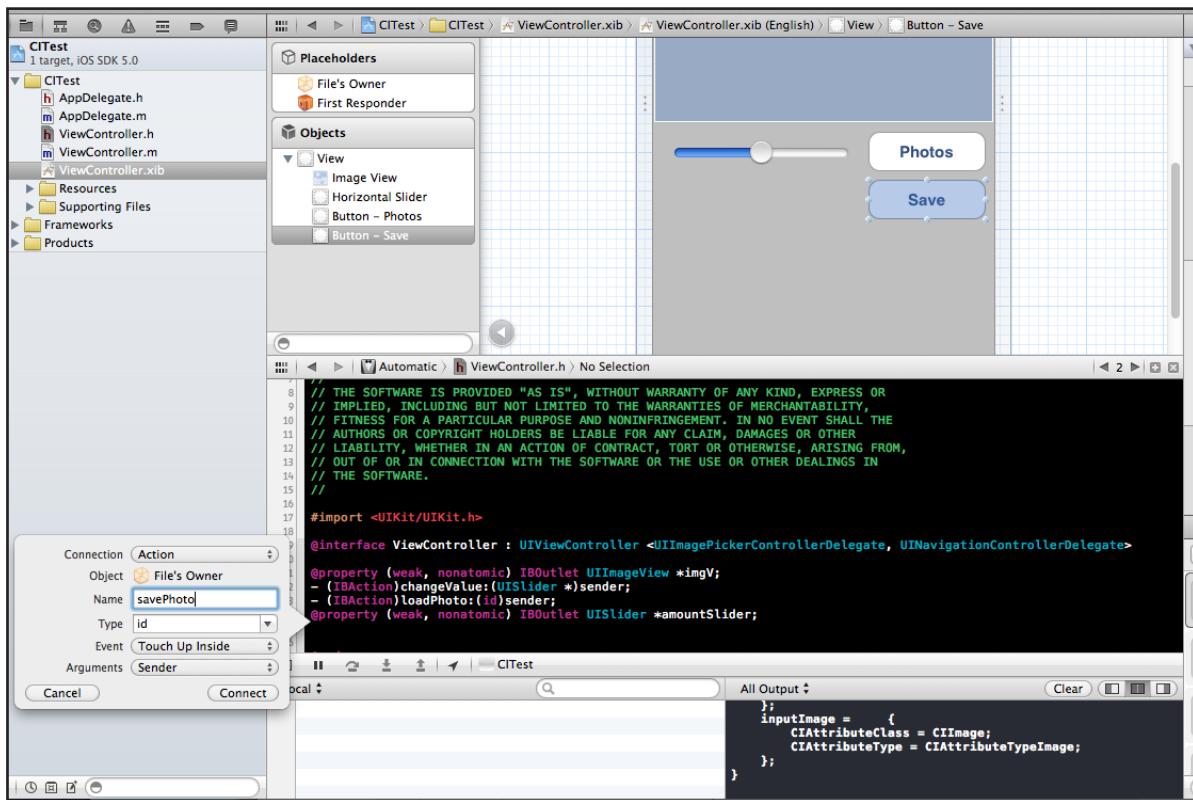
To use the CPU in our context we need to add options to our CIContext initialization. Remember that we passed in nil? Well we're gonna change that:

```
context = [CIContext contextWithOptions:  
          [NSDictionary dictionaryWithObject:[NSNumber numberWithBool:YES]  
             forKey:kCIContextUseSoftwareRenderer]];
```



Congratulations, now you're using the CPU to perform the CI calculations. Note however that software rendering does not work on the simulator, so you'll need to test on a device from now on.

Let's add a new button to our app that will let us save the photo we are currently modifying with all the changes we've made. Open **ViewController.xib**, add a new button, and connect it to a new **savePhoto** method, like you did last time.



Then switch to **ViewController.m** and implement the photo as follows:

```
- (IBAction)savePhoto:(id)sender {
    CIImage *saveToSave = [filter outputImage];
    CGImageRef cgImg = [context createCGImage:saveToSave
        fromRect:[saveToSave extent]];
    ALAssetsLibrary* library = [[ALAssetsLibrary alloc] init];
    [library writeImageToSavedPhotosAlbum:cgImg
        metadata:[saveToSave properties]
        completionBlock:^(NSURL *assetURL, NSError *error) {
            CGImageRelease(cgImg);
        }];
}
```

The AssetsLibrary framework needs to be added to get rid of the errors we are getting on this code. Go back to the project container, 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>
```

In this code block we:

1. Get the `CIImage` output from the filter.
2. Generate the `CGImageRef`.
3. Save the `CGImageRef` to the photo library.
4. Release the `CGImage`. That last step happens in a callback block so that it only fires after we're done using it.

Compile 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. When we save our file to our photo library we want to preserve these attributes.

The metadata associated with a `CIImage` can be accessed by the `properties` method. In the previous code section we are passing the `CIImage` metadata in to the metadata for the generated library's `save` function, so we're good to go!

What Other Filters are Available?

The `CIFilter` API has 130 filters on the Mac OS plus the ability to create custom filters. On the iOS platform, it has 48 or more. The reason for this is that filters are being added. 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, we can use the `[CIFilter filterNames InCategory:kCICategoryBuiltIn]` method. This method will return an array of filter names. In addition, each filter has an `attributes` method that will return a diction-



ary 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 our 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];
```

Compile and run, and you should see some console output that looks something like this:



```
CISepiaTone,
CISoftLightBlendMode,
CISourceAtopCompositing,
CISourceInCompositing,
CISourceOutCompositing,
CISourceOverCompositing,
CIStraightenFilter,
CIStripesGenerator,
CITemperatureAndTint,
CIToneCurve,
CIVibrance,
CIVignette,
CIWhitePointAdjust
)
2011-09-28 00:53:13.710 CIPrject[4143:707] {
CIAttributeFilterCategories = (
    CICategoryCompositeOperation,
    CICategoryVideo,
    CICategoryStillImage,
    CICategoryInterlaced,
    CICategoryNonSquarePixels,
    CICategoryHighDynamicRange,
    CICategoryBuiltIn
);
CIAttributeFilterDisplayName = Addition;
CIAttributeFilterName = CIAdditionCompositing;
inputBackgroundImage = {
    CIAttributeClass = CIImage;
    CIAttributeType = CIAttributeTypeImage;
};
inputImage = {
    CIAttributeClass = CIImage;
    CIAttributeType = CIAttributeTypeImage;
};
}
2011-09-28 00:53:13.713 CIPrject[4143:707] {
CIAttributeFilterCategories = (
    CICategoryGeometryAdjustment,
    CICategoryVideo,
    CICategoryStillImage,
```

In the resources file for this project, I've also attached a text file with the results of this output in case you want to look through the list of filters that are available.

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, where we'll cover compositing filters, masking images, additional filter examples, and even face detection!



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.

We looked at just one example filter in the chapter - the sepia tone filter. 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

You don't have to run just one filter per image like we've done so far - filters can be chained together!

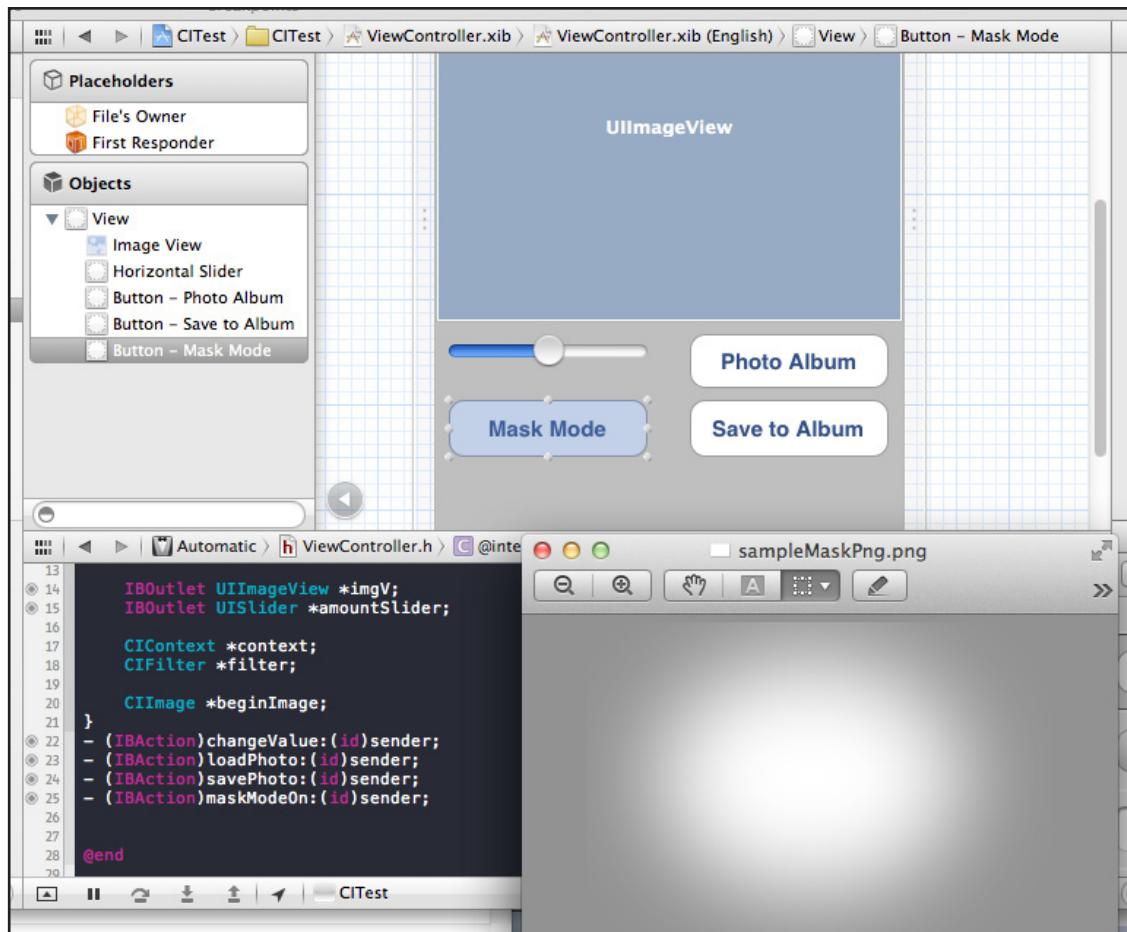
The output of one filter can be passed in as the input of another filter. The ability to composite filters creates virtually any possible desired effect. What's more, the process is lazy. The system doesn't process the filters one at a time, but rather waits until the entire chain is constructed and then creates a composite filter to apply to the image. This results in speedy performance.

We're going to use this chaining to create a simple app that will allow us to 'paint' a transparent circle on one image, revealing another image underneath.

We'll be using the following two filters: `CISourceAtopComposition` and `CISepiaTone`.

First we need to set up a button that will activate this new mode. We'll call it mask mode. Create the button and connect it to a new **maskMode** method as shown below:





The resources for this chapter include an image we'll be using for a mask - **sample-MaskPng.png**, so add it into your project. If you open it up, you'll see it's just a white sphere with a radial gradient that fades to transparent.

We're going to use a filter called `CISourceAtopCompositing`. It uses the white value of our sphere object to composite with the underlying layer. Where there's transparent space, the underlying image will be masked.

Implement the **maskMode** method as follows:

```

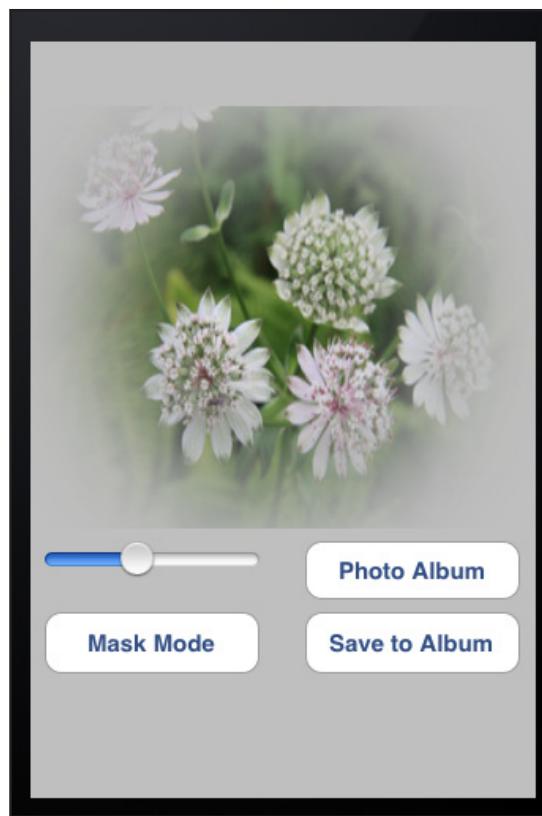
- (IBAction)maskMode:(id)sender {
    CIIImage *maskImage = [CIIImage imageWithCGImage:[UIImage
        imageNamed:@"sampleMaskPng.png"].CGImage];
    CIFilter *maskFilter = [CIFilter filterWithName:@"CISource
        AtopCompositing" keysAndValues:kCIInputImageKey, beginImage,
        @"inputBackgroundImage", maskImage, nil];
    CGImageRef cgImg = [context createCGImage:maskFilter.outputImage
        fromRect:[maskFilter.outputImage extent]];
    [imgV setImage:[UIImage imageWithCGImage:cgImg]];
    CGImageRelease(cgImg);
}
  
```



This code creates a `CIImage` from our `sampleMaskPng.png` file. We're using a different method here to load the image. It's probably less efficient to do it this way, because we are first creating the `UIImage` object, then getting the `CGImage` from the `UIImage`, which we 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`).

We create the new filter. This filter has only two input keys, `kCIInputImageKey`, and `@"inputBackgroundImage"`. Then we are using the same process to display the image. We don't use our `[self changeValue]` method here because that would load the SepiaTone Filter.

If you build and run and hit the new button, you should see the following:



Now if we slide our slider, we get the `CISepiaTone` filter and if we press the mask mode button we get the `CISourceCompositingAtop` filter. Let's change the mask mode method, so that we can use both at the same time.

We'll need to create a boolean or some other method to keep track of whether the mask mode is on or off, and we'll need to create a filter chain. I'm just going to 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 we slide, it's there that we need to check for whether the mask mode is on or off. Also, we need to add the output of our CISepiaTone filter to the input of the CISourceCompositingAtop filter.

First we need to create an instance variable for the mask mode button. Just use a control drag to create an IBOutlet called `maskModeButton` in the header. Then we need to change our methods to incorporate some new code. Here's what the first of those two methods should look like.

```
- (IBAction)maskMode:(id)sender {  
  
    CIImage *maskImage = [CIImage imageWithCGImage:  
        [UIImage imageNamed:@"sampleMaskPng.png"].CGImage];  
    CIFilter *maskFilter = [CIFilter filterWithName:  
        @"CISourceAtopCompositing" keysAndValues:kCIInputImageKey,  
        [filter outputImage], @"inputBackgroundImage", maskImage, nil];  
  
    CIImage *outputImage = [maskFilter outputImage];  
  
    if ([maskModeButton.titleLabel.text  
        isEqualToString:@"Mask Mode Off"]) {  
        [maskModeButton setTitle:@"Mask Mode"  
            forState:UIControlStateNormal];  
        outputImage = [filter outputImage];  
    } else {  
        [maskModeButton setTitle:@"Mask Mode Off"  
            forState:UIControlStateNormal];  
    }  
  
    CGImageRef cgImg = [context createCGImage:outputImage  
        fromRect:[outputImage extent]];  
    [imgV setImage:[UIImage imageWithCGImage:cgImg]];  
    CGImageRelease(cgImg);  
}
```

What we're trying to do is turn the mask filter, `CISourceAtopCompositing`, on and off. We start the method by setting up that filter and giving it our `CISepiaTone` filter's output image as its input image.

First we are using the label on our `maskModeButton` to determine what state we are currently in "Mask Mode" or "Mask Mode Off". This may look backwards at first, but consider the following, the text displayed on the button is the mode that pressing the button will change us to. So, if the button currently says, "Mask Mode Off", then pressing the button turns it off, which means that we are currently in mask mode.

So, our first block checks the label on the button. If the button currently says Mask Mode Off, then we are entering that state. We want to turn it off. So, in that case



we change the value of the `outputImage` (which currently holds both our initial image and our mask) to our initial filter's output (no mask). Then we change the label on the button.

If the label doesn't say "Mask Mode Off", then we are in the process of turning the mask mode on. In that case, all we need to do change the label and proceed using the filter as it exists, with the mask on. The whole process is a little circuitous.

Next we want to fix the `changeValue` method so it applies the mask filter if it's been turned on. Here's what that method looks like:

```
- (IBAction)changeValue:(UISlider *)sender {
    float slideValue = sender.value;

    [filter setValue:[NSNumber numberWithFloat:slideValue]
        forKey:@"inputIntensity"];
    CIImage *outputImage = [filter outputImage];

    if ([maskModeButton.titleLabel.text isEqualToString:
        @"Mask Mode Off"]) {
        CIImage *maskImage = [CIImage imageWithCGImage:[UIImage
            imageNamed:@"sampleMaskPng.png"].CGImage];
        CIFilter *maskFilter = [CIFilter filterWithName:
            @"CISSourceAtopCompositing" keysAndValues:kCIIInputImageKey,
            outputImage, @"inputBackgroundImage", maskImage, nil];
        outputImage = maskFilter.outputImage;
    }

    CGImageRef cgimg = [context createCGImage:outputImage
        fromRect:[outputImage extent]];
    UIImage *newImg = [UIImage imageWithCGImage:cgimg];
    [imgV setImage:newImg];
    CGImageRelease(cgimg);
}
```

Here we are employing the same strategy. If we see that the label on the button indicates that pressing the button will turn the masking mode off (meaning it's currently on), then we need to apply the `CISSourceAtopCompositing` filter in addition to the `CISepiaTone` filter we've already applied.

Again, all these filters do when we instantiate them is set themselves up. They are only applied to the image, efficiently and all at once, when we render them to a `CGImage`. That means that a bunch of filters can all be used and still run pretty quickly.

If you build and run now, you should have an app that allows you to adjust the sepia value while in mask mode!

Combining a Filtered Image With Another

What we'd like to do next is be able to have two layers (our current sepia layer and a new layer) that consist of a chosen image that will appear beneath our current masked image.

From the resources for this chapter, add **bryce.png** to your project. We're going to use this image as our bottom layer image.

Add the following code your class (add the method declaration to your header as well):

```
-(CIIImage *)addBackgroundLayer:(CIIImage *)inputImage {
    CIIImage *bg = [CIIImage imageWithCGImage:[UIImage
        imageNamed:@"bryce.png"].CGImage];
    CIFilter *sourceOver = [CIFilter filterWithName:@"CISource
        OverCompositing" keysAndValues:kCIInputImageKey, inputImage,
        @"inputBackgroundImage", bg, nil];
    return sourceOver.outputImage;
}
```

This code should make sense at this point. This method is going to take an input image we 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, we won't see bryce at all.

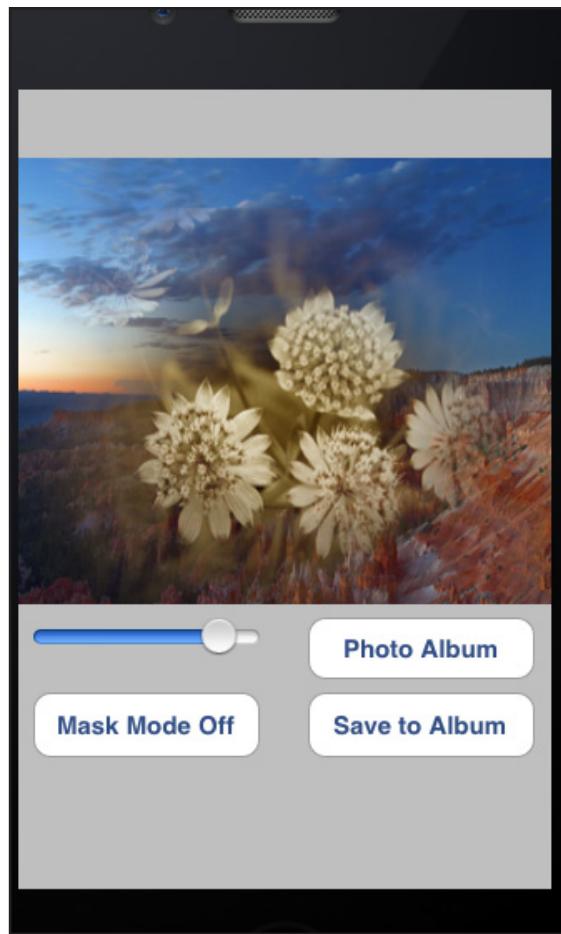
However, when we use our masked, sepia'd output image as the input image for this filter, we'll see the bryce in those transparent areas.

Add the following line to both the `changeValue` and the `maskModeOn` methods right before the line that creates our `cgImg` (the `CGImageRef`):

```
outputImage = [self addBackgroundLayer:outputImage];
```

Go ahead and run it now and you'll see the following:



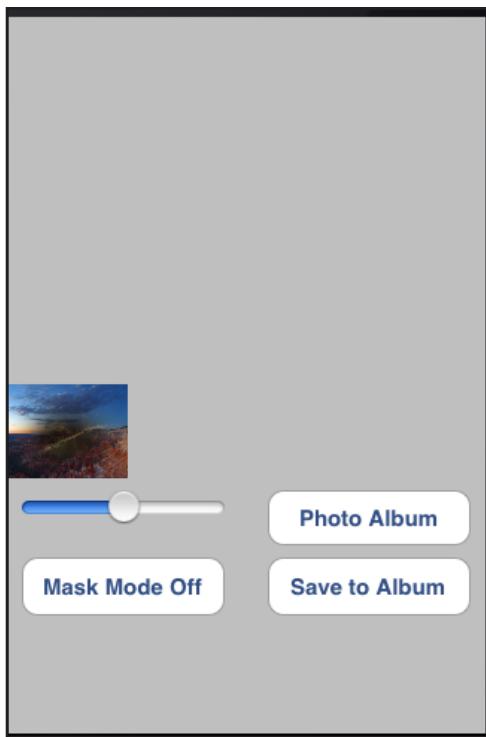


Pretty cool for just a few lines of code, eh?

Resizing our Input Image

One problem we'll run into now is that if we are using images of different sizes, the `CIImage` filter chain we've set up is combining an image of say 400 pixels wide by 300 pixels tall with our 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 our `imgV` container. We need to resize the incoming image from the photos album so it's the same size at the image of bryce.

We'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.

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 this: `UIImage *gotImage = [info objectForKey:@"UIImagePickerControllerOriginalImage"];`

```
gotImage = [gotImage scaleToSize:[beginImage extent].size];
```

This will fix the problem. 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!

We can create an `CIIImage` with either raw pixel data or with a `CGImage`. If we draw a `CGImage` using Core Graphics calls, we can pass that in as a `CIIImage` for one of our filters. This capability opens up all kinds of possible interaction.

Let's write the draw call first:

```
- (CGImageRef)drawMyCircleMask:(CGPoint)location {
    NSUInteger width = 320;
    NSUInteger height = 213;
    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, 1, .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. When we draw with core graphics we always need a context. Sometimes that can be the context that represents what we'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. We're calling `CGContextSetRGBFillColor` to set our color to white (1, 1, 1) with a transparency value of 70% (.7). We're setting that for our bitmap context. Next we call the `CGContextFillEllipseInRect` method that actually draws our circle. We're gonna be using a circle as a mask.

We give it a `x` and a `y` origin, and then the width and height. We are using the touch point that we passed in for this location. We want the circle to be centered



around our touch, so we need to subtract half the width and height (25) from the origin points.

We then use what we've drawn in our context to create a `CGImageRef`. This will be returned to calling object.

Next we 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:imgV];
    if (loc.y <= 213 && loc.y >= 0) {
        loc = CGPointMake(loc.x, imgV.frame.size.height - loc.y);
        CGImageRef cgimg = [self drawMyCircleMask:loc];
        UIImage *img = [UIImage imageWithCGImage:cgimg];
        imgV.image = img;
    }
}
```

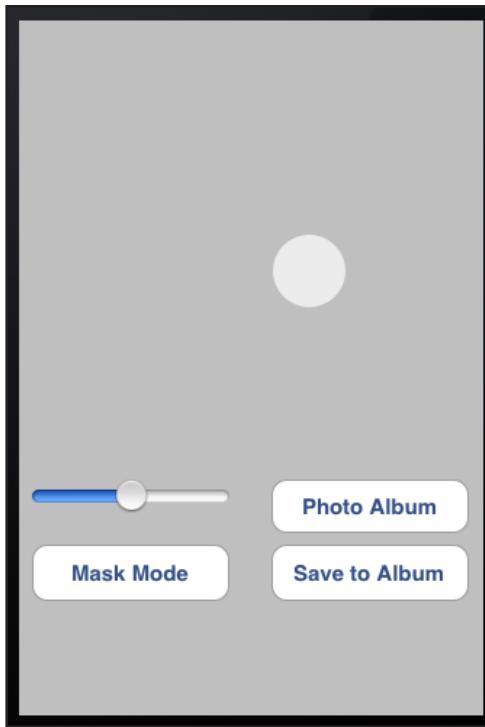
We'll replicate this method for `touchesMoved` as well. But, for right now this will suffice. We first get the location of any of the touches in our set. We're only interested in touches that reside within the bounds of the `UIImageView` here. So we're checking to see if the touch was inside it's frame. We don't need to worry about the x bounds, because it stretches the width of the screen.

Next, we convert the coordinate space from `UIView` to Core Graphics by subtracting the y coordinate from the height of the `imgV`. Then we create a `CGImageRef` from the method we just created, passing in our `loc` `CGPoint`.

What we're going to do ultimately is pass this `CGImage` into a `CIIImage` and `CIFilter`, but for now, let's just convert it to a `UIImage` and display it on screen so we 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`:





Now that we have everything we need in place, we should be able to use this new `CGImage` as our `CIIImage` in our masking filter. We want the `CIIImage` that represents the mask to be available throughout the program. So we need to add it to our list of private instance variables:

```
CIImage *maskImage;
```

Also, we should probably initialize it in the `viewDidLoad` method:

```
maskImage = [CIImage imageWithCGImage:  
[UIImage imageNamed:@"sampleMaskPng.png"].CGImage];
```

Since we initialize it in the `viewDidLoad`, we no longer need to initialize it in our other methods. Take that line out of both the `changeValue` and `maskModeOn` methods.

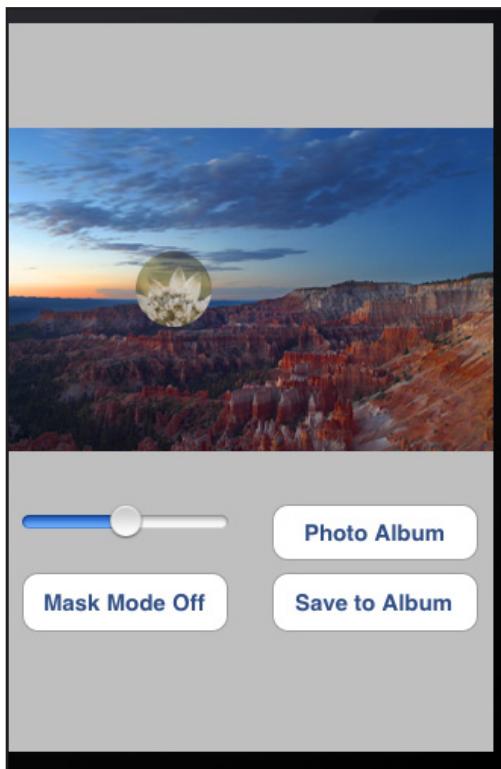
Now, when we run the `touchesBegan` method, instead of setting the `imgV` image to our drawn `CGImage`, we'll be replacing the `CIIImage` in the filter with it. Here's the code for that:

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {  
    CGPoint loc = [[touches anyObject] locationInView:imgV];  
    if (loc.y <= 213 && loc.y >= 0) {  
        loc = CGPointMake(loc.x, imgV.frame.size.height - loc.y);  
        CGImageRef cgimg = [self drawMyCircleMask:loc];  
        maskImage = [CIImage imageWithCGImage:cgimg];  
        [self changeValue:amountSlider];  
    }  
}
```

{

We want to release the previous CIImage and create our new maskImage. Finally, we call the changeValue method to update and display our new composite filtered image.

Duplicate the touchesBegan method and rename it touchesMoved. Build and run now, you'll see that initially when you turn on mask mode, it still uses the static image, but once you touch the screen it changes to our circle mask, and the circle will follow your finger around. Getting cool!



But, we really want more of a paint type mask than just a circle that moves. So, that's the last thing we'll do. We'll want to modify our draw method to keep the current context around and have an additive model. We'll need to add our cgcontext variable to our class and initialize it in our viewDidLoad method, then just add circles to it every time we detect a touch.

Let's do that now. First add the cgcontext variable to the private instance variables list:

```
CGContextRef cgcontext;
```

We'll break up our 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 existing one with these two:

```

-(void)setupCGContext {
    NSUInteger width = 320;
    NSUInteger height = 213;
    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, 213));
    }

    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;
}

```

Also call setupCGContext in viewDidLoad (you might have to predeclare setupCGContext in the header or a private category to get this to work):

```
[self setupCGContext];
```

Note we added a second parameter to the drawMyCircleMask method. We're going to reset the drawing each time we call touchesBegan. If the passed in parameter is set to YES, then we'll call CGContextClearRect(cgcontext, CGRectMake(0, 0, 320, 213)), 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:imgV];
    if (loc.y <= 213 && loc.y >= 0) {
        loc = CGPointMake(loc.x, imgV.frame.size.height - loc.y);
        CGImageRef cgimg = [self drawMyCircleMask:loc reset:YES];
    }
}

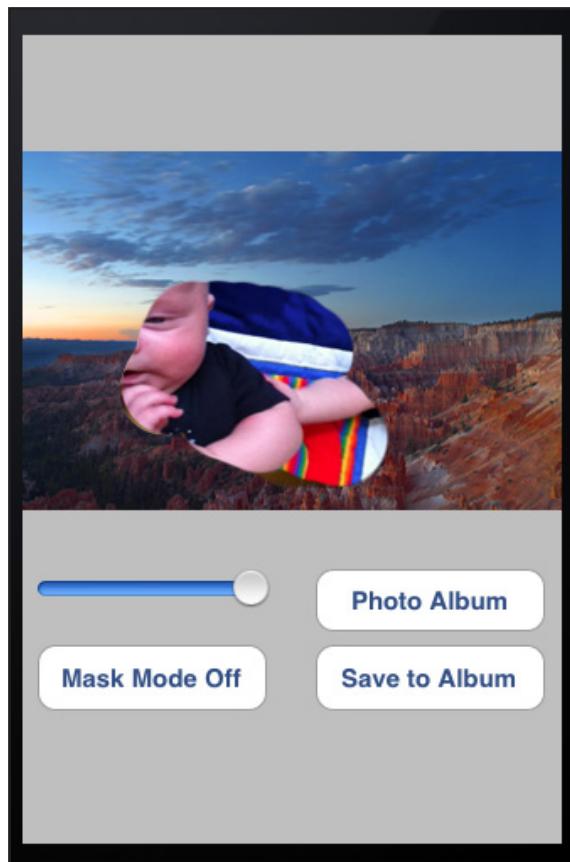
```



```
        maskImage = [CIImage imageWithCGImage:cgi];
        [self changeValue:amountSlider];
    }

-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    CGPoint loc = [[touches anyObject] locationInView:imgV];
    if (loc.y <= 213 && loc.y >= 0) {
        loc = CGPointMake(loc.x, imgV.frame.size.height - loc.y);
        CGImageRef cgi = [self drawMyCircleMask:loc reset:NO];
        maskImage = [CIImage imageWithCGImage:cgi];
        [self changeValue:amountSlider];
    }
}
```

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 exists in the 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. We'll need to import an image with a face or faces 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>

Add this method:

```
- (void)hasFace:(CIImage *)image {
    CIDetector *faceDetector = [CIDetector
        detectorOfType:CIDetectorTypeFace context:nil
        options:[NSDictionary dictionaryWithObjectsAndKeys:
            CIDetectorAccuracyHigh, CIDetectorAccuracy, nil]];
    NSArray *features = [faceDetector featuresInImage:image];
    NSLog(@"%@", features);
    for (CIFaceFeature *f in features) {
        NSLog(@"%@", f.leftEyePosition.x, f.leftEyePosition.y);
    }
}
```

We instantiate the face detector and give it a type. CIDetectorTypeFace is the only type currently. We also pass it a dictionary of options. We can use CIDectectorAccuracyHigh or CIDectectorAccuracyLow, which is faster. Here we're using high.

This function has one purpose, to create the detector and record the information in the features array. We're passing in our desired CIImage as the argument that will be used for the featuresInImage method on the detector.

In this case we're also logging the result. We won't need this later, but to test whether the face detector is working we're using the log statements here.

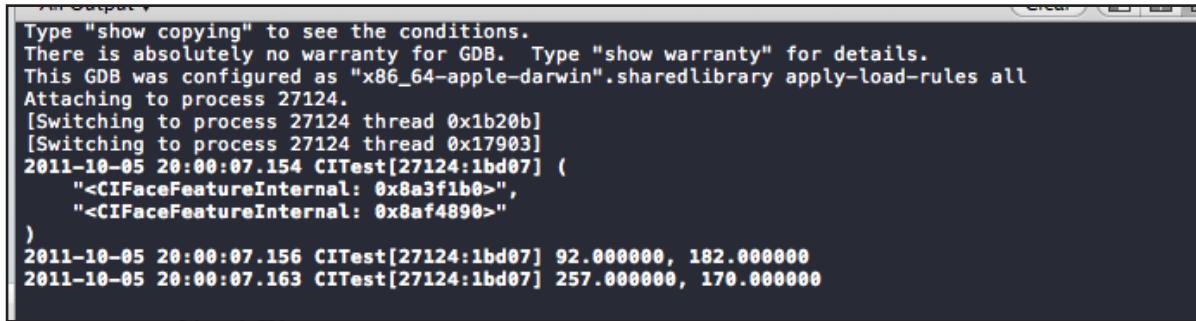
Now add this line to our method that retrieves photos from our photo album, right before the [self changeValue:amountSlider] line:

```
[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 we don't want to be waiting for it to finish. Go ahead and run this, making sure you've got a photo with good, front facing faces in it.



You should see the following in the log.



```

Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".sharedlibrary apply-load-rules all
Attaching to process 27124.
[Switching to process 27124 thread 0x1b20b]
[Switching to process 27124 thread 0x17903]
2011-10-05 20:00:07.154 CITest[27124:1bd07] (
    "<CIFaceFeatureInternal: 0x8a3f1b0>",
    "<CIFaceFeatureInternal: 0x8af4890>"
)
2011-10-05 20:00:07.156 CITest[27124:1bd07] 92.000000, 182.000000
2011-10-05 20:00:07.156 CITest[27124:1bd07] 257.000000, 170.000000

```

Now let's do something with the information we've collected from our face detector. We're going to be creating small boxes that we'll composite with our `inputImage`, so that each feature, mouth, left eye, right eye, each have their own box. This way we can see how well the face detection is working.

These next two methods are beasts, so we'll walk through it methodically. Be sure to add these above `hasFace`:

```

-(CIImage *)createBox:(CGPoint)loc color:(CIColor *)color {
    CIImage *colorFil = [CIFilter filterWithName:@"CIConstantColorGenerator"
        keysAndValues:@["inputColor", color, nil].outputImage;
    return [CIFilter filterWithName:@"CICrop"
        keysAndValues:kCIInputImageKey, colorFil, @"inputRectangle",
        [CIVector vectorWithCGRect:CGRectMake(loc.x - 3, loc.y - 3, 6, 6)],
        nil].outputImage;
}

```

In this first method we are creating a new `CIImage` with a small box at the center point of the passed in location. We're going to be using these boxes to mark the eyes and mouth of our faces.

The first filter creates a solid color image, the `CIConstantColorGenerator` takes only one input in the form of a `CIColor`. We'll show you how to instantiate a `CIColor` object next, but it's what you'd expect.

We pass the first filter's `outputImage` (`CIImage`) into the next filter. The `CICrop` filter takes an `inputImage` and a `CIVector` that we create with a `CGRect`. We're just creating a rect that six pixels wide and six pixels tall, with the center in the provided location. Each time we run this we'll get a `CIImage` that has a small box in the location we provide. We're basically using Core Image filters to draw boxes.

On to the face marking:

```

- (void)filteredFace:(NSArray *)features {
    CIImage *outputImage = beginImage;

```



```
for (CIFaceFeature *f in features) {
    if (f.hasLeftEyePosition) {
        outputImage = [CIFilter filterWithName:
            @"CIComposite"
            keysAndValues:kCIInputImageKey,
            [self createBox:CGPointMake(f.leftEyePosition.x,
            f.leftEyePosition.y)
            color:[CIColor colorWithRed:1.0 green:0.0 blue:0.0
            alpha:0.7]],
            kCIInputBackgroundImageKey, outputImage, nil].
            outputImage;
    }
    if (f.hasRightEyePosition) {
        outputImage = [CIFilter filterWithName:
            @"CIComposite"
            keysAndValues:kCIInputImageKey,
            [self createBox:CGPointMake(f.rightEyePosition.x,
            f.rightEyePosition.y)
            color:[CIColor colorWithRed:1.0 green:0.0 blue:0.0
            alpha:0.7]],
            kCIInputBackgroundImageKey, outputImage, nil].
            outputImage;
    }
    if (f.hasMouthPosition) {
        outputImage = [CIFilter filterWithName:
            @"CIComposite"
            keysAndValues:kCIInputImageKey,
            [self createBox:CGPointMake(f.mouthPosition.x,
            f.mouthPosition.y)
            color:[CIColor colorWithRed:0.0 green:1.0 blue:0.0
            alpha:0.7]], kCIInputBackgroundImageKey, outputImage,
            nil].outputImage;
    }
}

[filter setValue:outputImage forKey:kCIInputImageKey];
[self changeValue:amountSlider];
}
```

First we store the reference to our beginImage in an CIImage pointer called outputImage. This just makes it easier to keep the code all uniform. For each feature, we'll be compositing the result of the createBox together with the previous outputImage.

We iterate through our features array that we got from the face detector. For each feature there is a hasFeature boolean that tells us whether the detector was able to detect that feature. Here we're checking each one, and if it exists, we're creating a new filter.



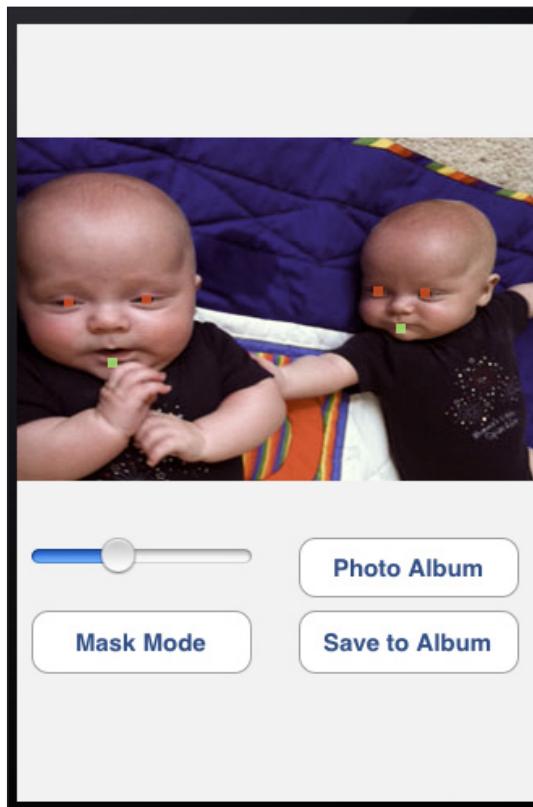
The compositing filter we've seen before. However, what's different is we're using our `createBox` method to create a new `CIImage`, one with a box in the location of the found feature and a `CIColor` that we're providing. We're using red for the eyes and green for the mouth.

When we're finished adding all the boxes we set the value of the output to our input for our class filter and we call the `changeValue` to redraw the image on screen.

Change the `hasFace` method by removing the logging code and adding a call to our `filteredFace` method like this:

```
- (void)hasFace:(CIImage *)image {
    CIDetector *faceDetector = [CIDetector detectorOfType:
        CIDetectorTypeFace context:nil options:[NSDictionary
            dictionaryWithObjectsAndKeys:CIDetectorAccuracyHigh,
            CIDetectorAccuracy, nil]];
    NSArray *features = [faceDetector featuresInImage:image];
    [self filteredFace:features];
}
```

Congrats, you should have an app that can now detect and mark face features!

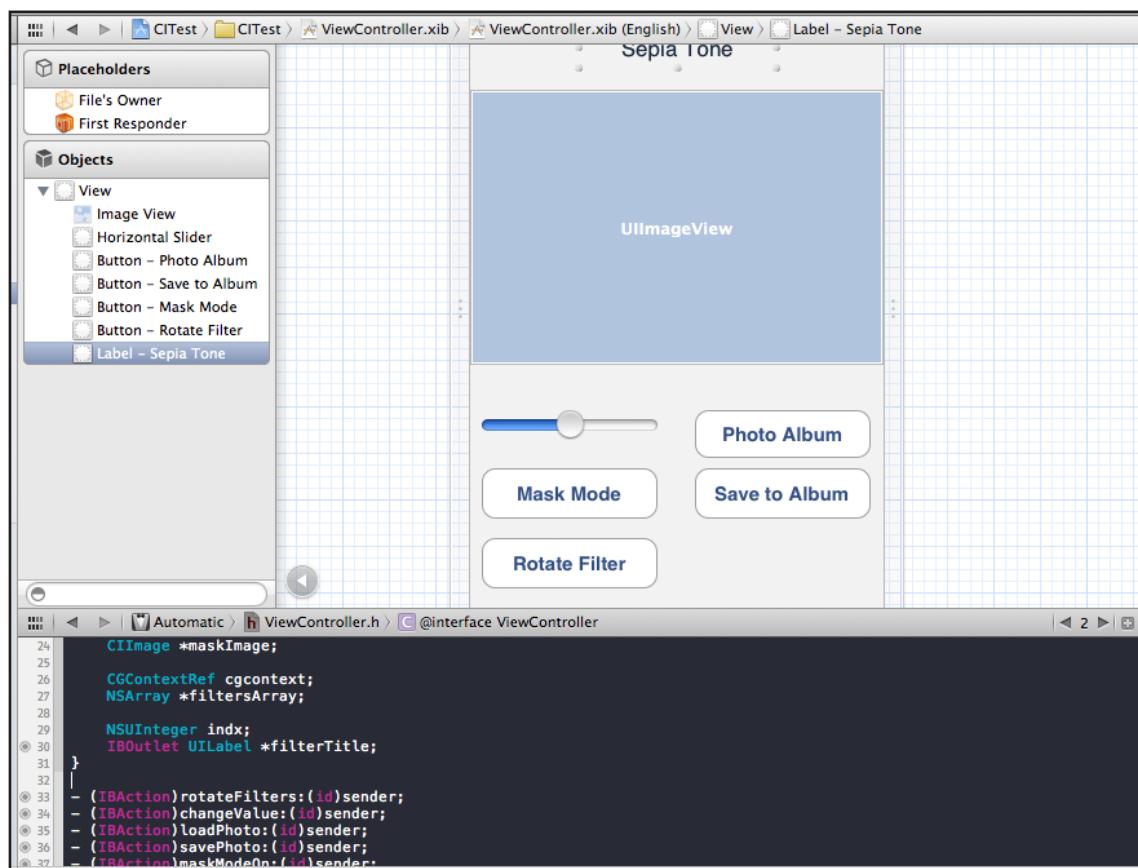


Looking at Other Filters

We'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. We're going to set up a half dozen filters with preset input values, and then use a button to rotate through them.

First, let's add a new button and a label to our .xib. The label will let us know the name of the filter we'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 indx;

```

We'll need to set up the filters in the viewDidLoad method. 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 this code to your viewDidLoad method. This sets up our filters and adds them to our array:

```
CIFilter *affineTransform = [CIFilter filterWithName:  
    @"CIAffineTransform" keysAndValues:kCIInputImageKey, beginImage,  
    @"inputTransform", [NSNumber valueWithCGAffineTransform:  
        CGAffineTransformMake(1.0, 0.4, 0.5, 1.0, 0.0, 0.0)], nil];  
CIFilter *straightenFilter = [CIFilter filterWithName:  
    @"CIStraightenFilter" keysAndValues:kCIInputImageKey, beginImage,  
    @"inputAngle", [NSNumber numberWithFloat:2.0], nil];  
CIFilter *vibrance = [CIFilter filterWithName:  
    @"CIVibrance" keysAndValues:kCIInputImageKey, beginImage,  
    @"inputAmount", [NSNumber numberWithFloat:-0.85], nil];  
CIFilter *colorControls = [CIFilter filterWithName:  
    @"CIColorControls" keysAndValues:kCIInputImageKey, beginImage,  
    @"inputBrightness", [NSNumber numberWithFloat:-0.5],  
    @"inputContrast", [NSNumber numberWithFloat:3.0],  
    @"inputSaturation", [NSNumber numberWithFloat:1.5], nil];  
CIFilter *colorInvert = [CIFilter filterWithName:  
    @"CIColorInvert" keysAndValues:kCIInputImageKey, beginImage, nil];  
CIFilter *highlightsAndShadows = [CIFilter filterWithName:  
    @"CIHighlightShadowAdjust" keysAndValues:  
        kCIInputImageKey, beginImage,  
        @"inputShadowAmount", [NSNumber numberWithFloat:1.5],  
        @"inputHighlightAmount", [NSNumber numberWithFloat:0.2], nil];  
  
filtersArray = [NSArray arrayWithObjects:  
    affineTransform, straightenFilter, vibrance, colorControls,  
    colorInvert, highlightsAndShadows, nil];  
indx = 0;
```

We have provided values for many of these filters. However, filters have default values that are used if we don't supply any. Keep that in mind.

Let's go over what each of these filters do:

- **CIAffineTransform** helps us apply a CGAffineTransform to an image. These transforms can scale, rotate, and skew an image.
- **CIStraightenFilter** is used to fix an image take with the camera tilted slightly. We give it an `inputAngle` property to determine the amount of rotation to correct.
- **CIVibrance** will increase the color intensity.
- **CIColorControls** allow us to change brightness, contrast, and saturation.
- **CIColorInvert** will invert the colors. It only takes one parameter, `inputImage`.



- **CIHighlightShadowAdjust** will allow us to change the values of the highlights and shadows in our image.

Now add this code to our rotateFilter method:

```
- (IBAction)rotateFilter:(id)sender {  
    CIFilter *filt = [filtersArray objectAtIndex:indx];  
    CGImageRef imgRef = [context createCGImage:[filt outputImage]  
        fromRect:[[filt outputImage] extent]];  
    [imgV setImage:[UIImage imageWithCGImage:imgRef]];  
    CGImageRelease(imgRef);  
    indx = (indx + 1) % [filtersArray count];  
    [filterTitle setText:[[filt attributes] valueForKey:  
        @"CIAtributeFilterDisplayName"]];  
}
```

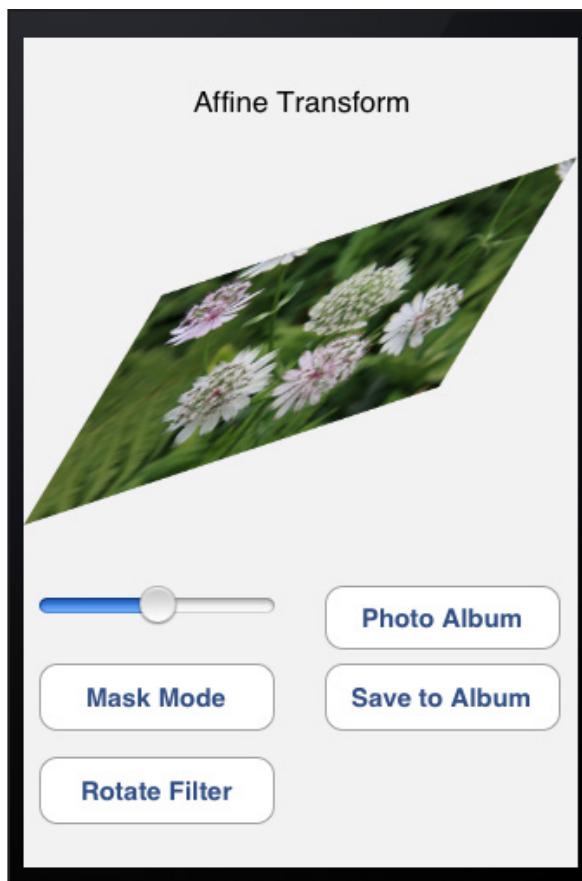
In this method we're using the `indx` variable to retrieve the filter from the array. Then we go through the regular process of converting a `CIImage` output from a filter into a `UIImage` we can use. Finally we increment the `indx` variable and we also update our `filterTitle` label to show the current filter.

These new filters will not interact with the slider. So, when we do decide to use the mask mode button or the slider, it will revert us back to the sepia tone filter. We want to update the label in that case. Add the following method to the end of the `changeValue` method:

```
[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.

We'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 the body of the method:

```
- (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:
```

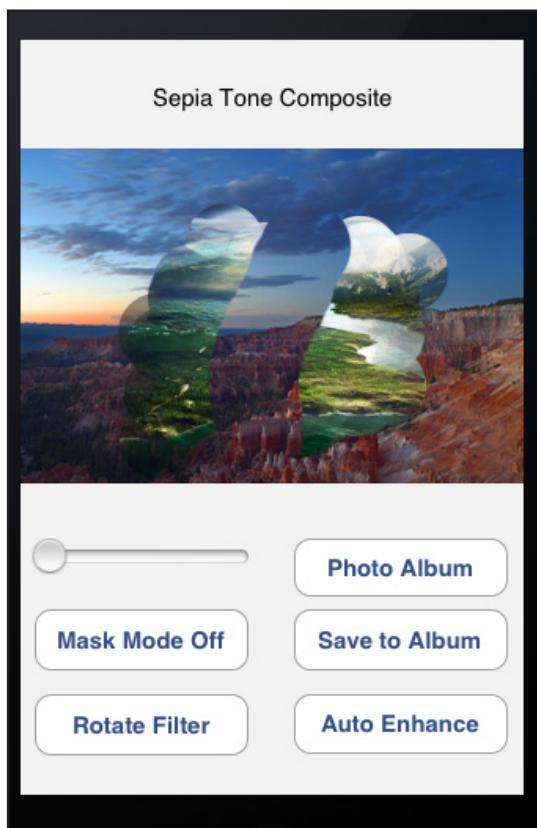


```
    @"CIAutoEnhanceFilterDisplayName"]);
}
[filter setValue:outputImage forKey:kCIInputImageKey];
[self changeValue:amountSlider];
}
```

Here we get the array of filters with the [beginImage autoAdjustmentFilters] call. Then we iterate through each filter. We first have to set our input image. We set that to outputImage, which is the output from the last filter operation. In this way we chain all these filter actions together. Also, we are logging the name of each filter so we can see what the auto-enhancement is doing.

Finally we set the new image as the inputImage for our main filter, and call changeValue to update 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!



Where To Go From Here?

We'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.

We covered the basics of how to set up and use core filters, but we 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.

For more information about Core Image, visit Apple's website. 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>

UIViewControllerAnimated 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

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 a view controller "almost" like the UISplitViewController, but with a wider left-hand-side, or some other tweaks.

In the past, people went ahead and wrote code to contain view controllers inside other view controllers anyway, but because there was no official supported way to do so, there were lots of subtle problems that arose, especially related to orientation changes and view controller lifecycle methods such as "viewWillAppear" not being called.

But with 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 the best part is it's extremely simple to use!

In this tutorial, we'll port an iPhone app to the iPad and present several smaller UIViewControllerControllers inside one large UIViewController. In the process, we'll show you how to use the new UIViewController containment APIs - and why you need them in the first place!



Introducing Xcode Shortcuts

If you look at the starter code for this chapter, you'll find a little project called Xcode Shortcuts.

This is a very simple app that lets you browse through the shortcuts available in Xcode and add them to a list of Favorites that you are trying to learn:

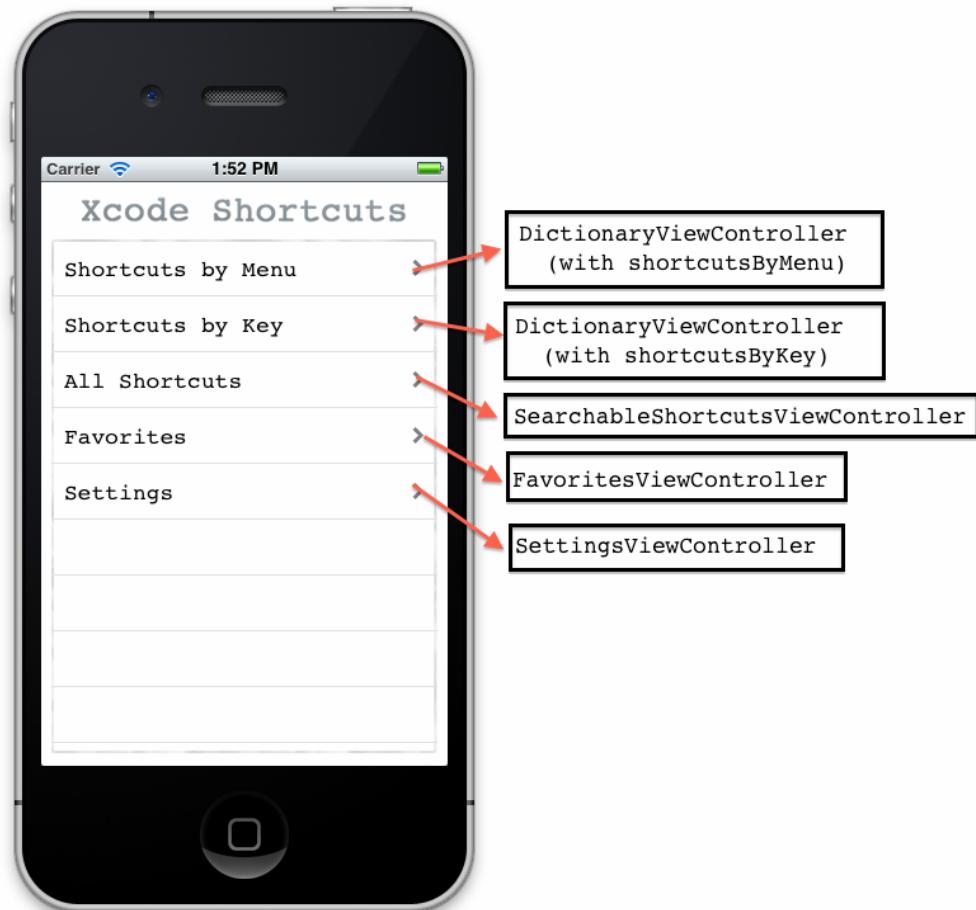


As you try out the app, be sure to rotate the simulator left and right, and notice that the app handles orientation changes nicely. This is especially interesting in the Favorites screen, which reorganizes the shortcuts to take advantage of the extra space:





If you look through the code, you'll see that the Window's root view controller is a Navigation Controller with a view controller called `MainMenuViewController` inside. This view controller simply pushes different view controllers onto the stack, depending on which you choose:



So play around with the app a bit and look over the code - and hopefully learn a few new Xcode shortcuts! When you're ready, keep reading and we'll discuss the task at hand.

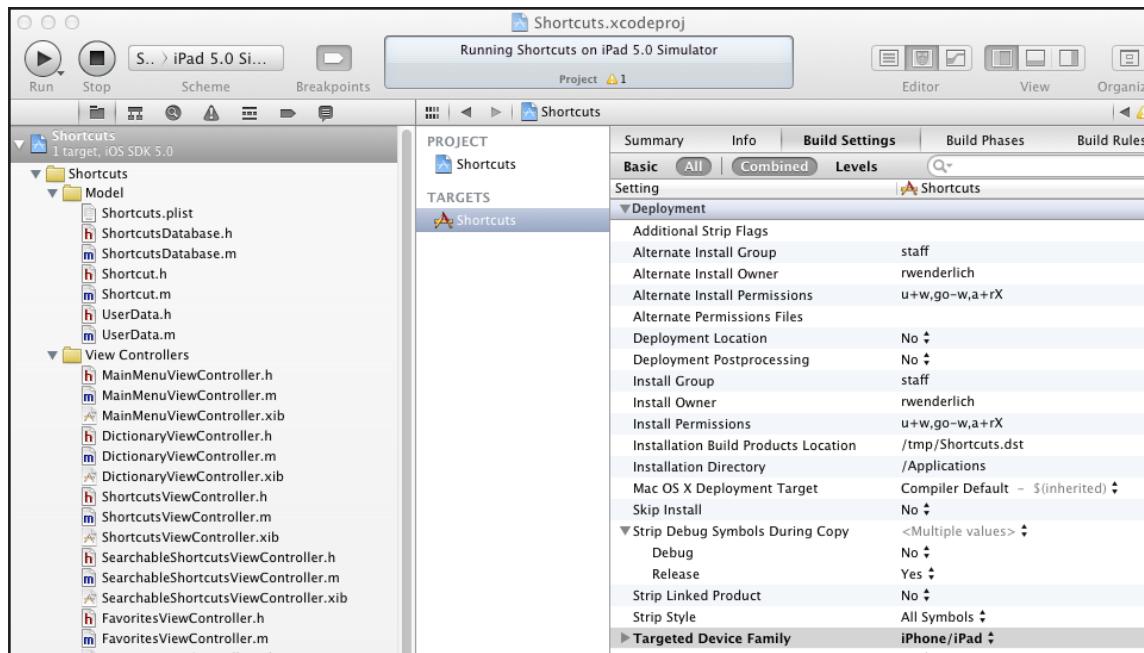
Porting to the iPad: Overview

The goal for this tutorial is to port this app to the iPad using view controller containment. Right now this app isn't even targeted for the iPad, so if you try running it on the iPad Simulator it will show up as an iPhone app like so:

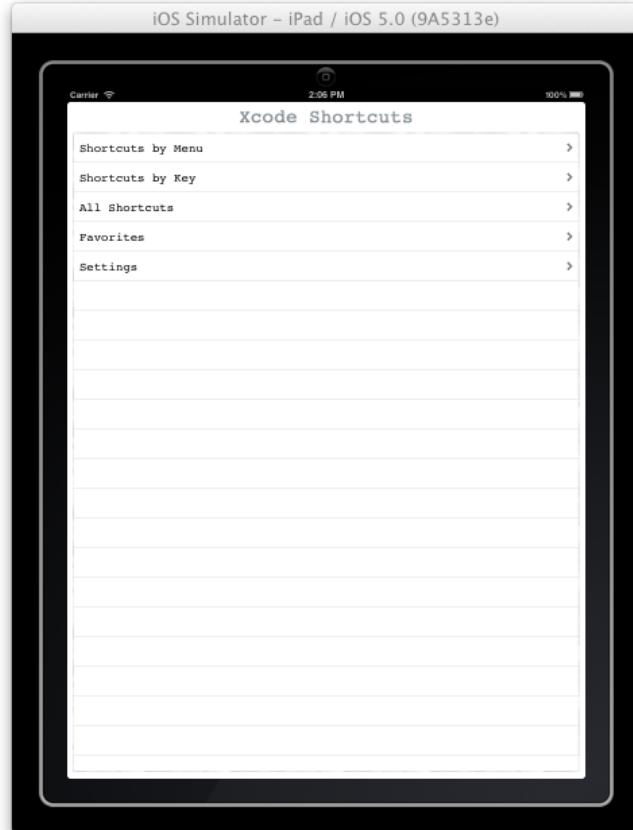


As you probably know, this is quite easy to fix. Click on the Shortcuts project in the Project navigator (the leftmost pane in Xcode), select the Shortcuts target, and set the **Deployment\Targeted Device Family** setting to **iPhone/iPad**:

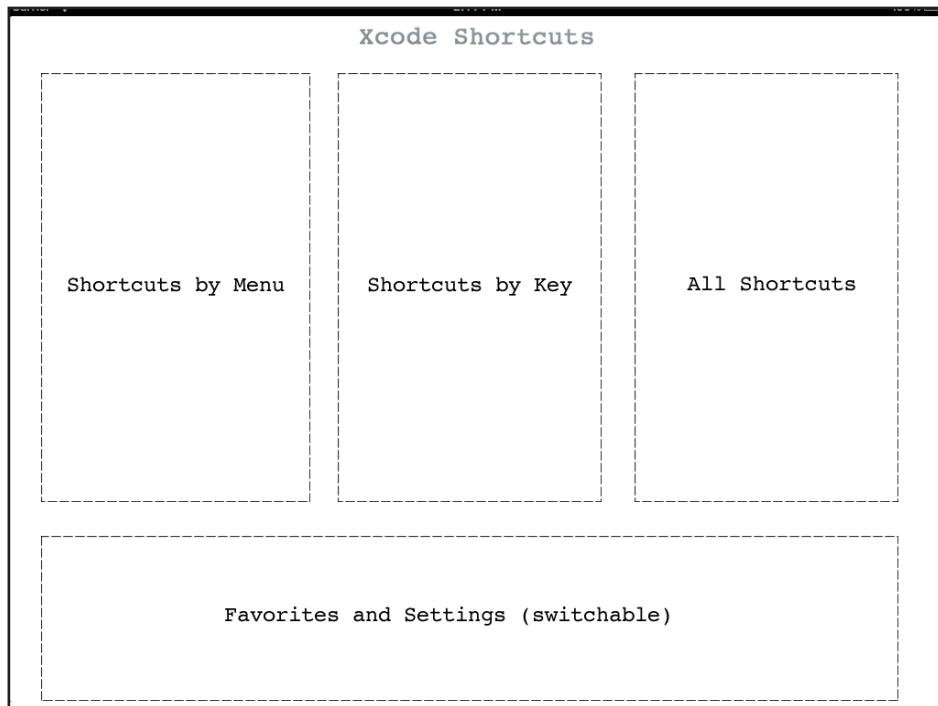




Compile and run, and you'll see that aside from some layout idiosyncrasies (which can be fixed by rotating the app), it sort of works - except it looks completely ridiculous on the large screen!



Since the iPad screen is so large, we can display a lot of information on the screen at the same time. Here's a mockup of how we could lay it out:



This layout flattens our view hierarchy and provides direct access to the three lists of shortcuts. Favorites are shown prominently at the bottom, with the option to alternate with Settings.

Since we are porting this from the iPhone to the iPad, it would be nice to incorporate our existing view controllers into an iPad-specific view controller, rather than recreate a standalone controller.

To do this, we'll create a new `iPadViewController` just for the iPad, with a really big view. The `iPadViewController` will add the existing view controller's views as subviews into its really large view.

The first time we do this we won't use the new `UIViewController` containment APIs (to show you why it's necessary), then we'll add the `UIViewController` containment APIs in to get everything working properly.

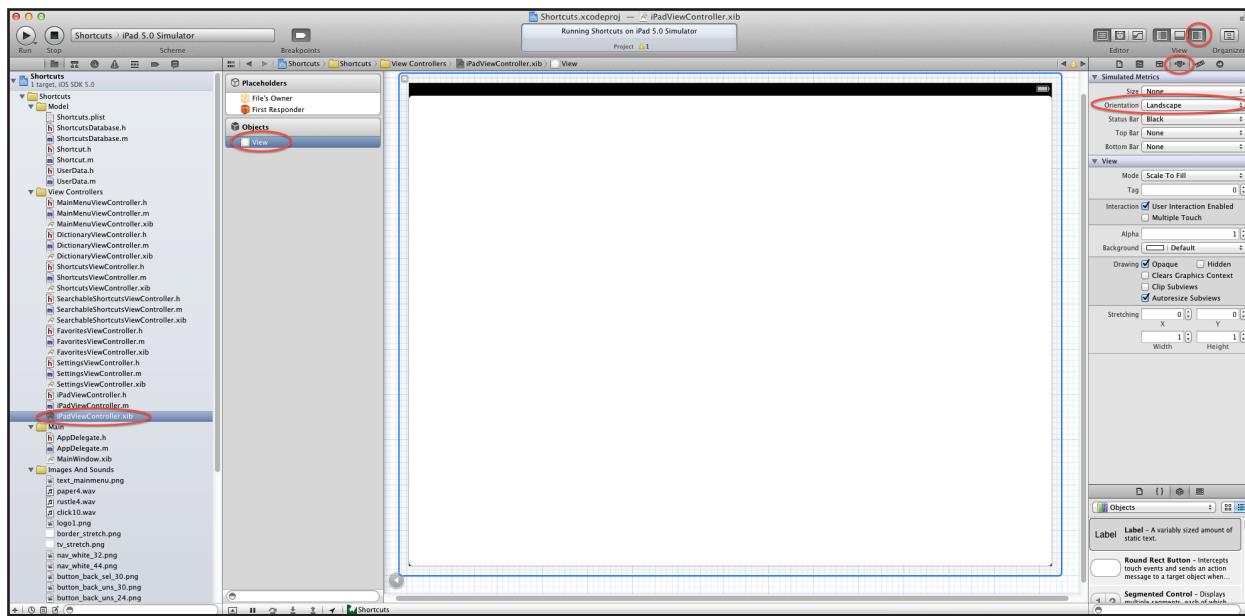
Alright, so let's try this out!



Laying out the iPad View Controller

Control-click the View Controllers group, and select **New File**. Select **iOS\Cocoa Touch\UIViewController subclass**, and click Next. Name the class **iPadViewController**, enter **UIViewController** for subclass of, make sure **Targeted for iPad** and **With XIB for user interface** are selected, click Next, and then click Create.

Open **iPadViewController.xib**, and select the View inside. In the Attributes Inspector, set the Orientation to Landscape (since it will be easier to lay everything out that way):



Also, set the View's background color to a gray color so it's easier to see the subviews we're about to add.

Next, let's create four UIViews for where we're going to put the existing view controller's views inside (and one more for decoration). This will make it easy to visually layout where they should go.

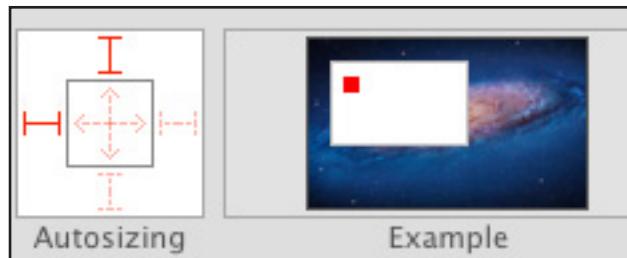
So drag five Views from the Object Library into the view controller, and give them the following settings:

- View 1 (the left): X=19, Y=73, Width=320, Height=433.
- View 2 (the middle): X=354, Y=73, Width=320, Height=433.
- View 3 (the right): X=690, Y=73, Width=320, Height=433.
- View 4 (the decoration): X=18, Y=522, Width=987, Height=210.

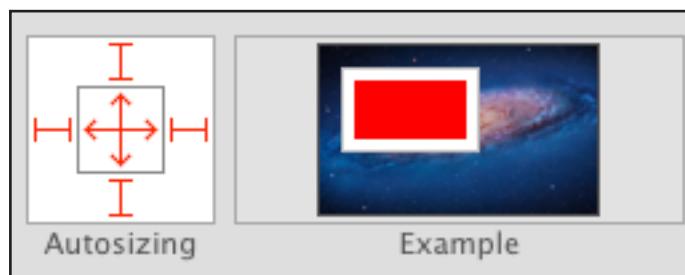


- View 5 (the bottom): X=202, Y=10, Width=775, Height=191.

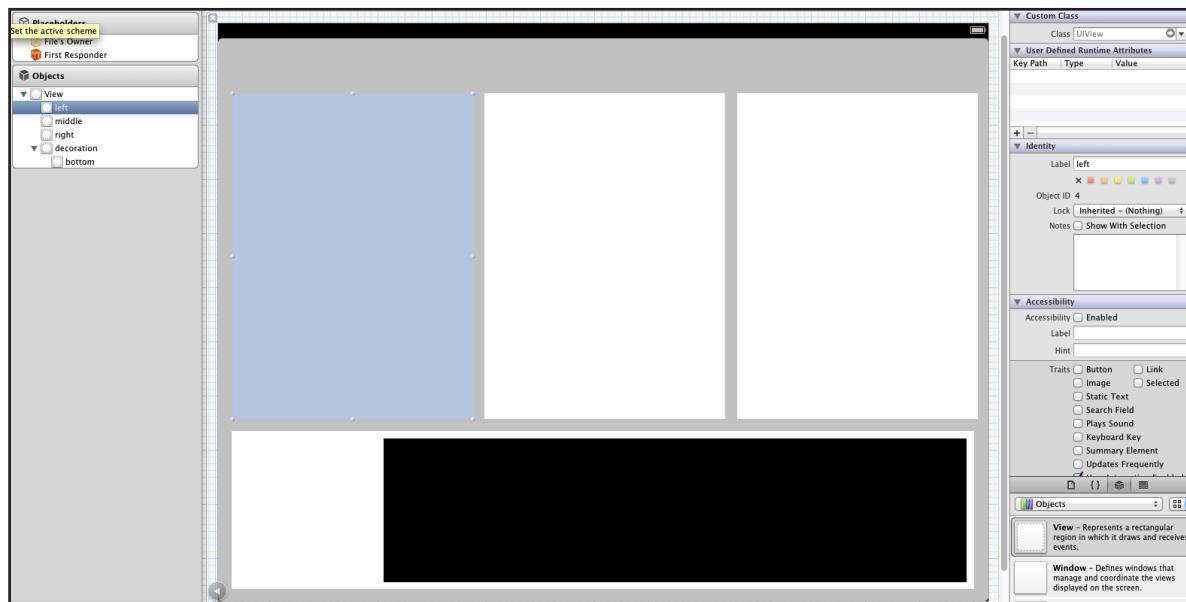
For each of these views, make sure their Autosizing settings (in the Size inspector) are set to this:



Except for the decoration view, which should be set to this:



Change the background color of the bottom View to black so it's easier to see, and give each view a label to make it easier to know what it is in the Objects tree (inside the Identity Inspector). At this point your layout should look like this:



Next, let's add some buttons into the decoration view to let the user switch between settings and favorites, and an image label as well. To do this:

- Drag a Round Rect Button into the decoration view. Set X=14, Y=126, Width=72, Height=72, Type to Custom, Default Image to button_faves_uns.png, and Highlighted Image to button_faves_sel.png.
- Drag another Round Rect Button into the decoration view. Set X=105, Y=126, Width=72, Height=72, Type to Custom, Default Image to button_settings_uns.png, and Highlighted Image to button_settings_sel.png.
- Drag an Image View into the decoration view. Set X=10, Y=10, Width=147, Height=100, and set the Image to textFavorites.png.

Almost done - just need to add two more Image Views for decoration. To do this:

- Drag an Image View into the root View and set X=0, Y=0, Width=1,024, and Height=748. Make sure it's at the top of the Objects list right under the root View so it's behind everything else. We'll be setting this to a background image in code.
- Drag another Image View into the root View set X=23, Y=15, Width=446, Height=42, and the Image to text_xcode.png.

At this point, your layout should look like the following:



That takes care of the layout for now - let's see if we can get this to show up on startup!

Creating a MainWindow.xib for iPad

If you look at the current **MainWindow.xib**, you'll see that it's set up to make the root view controller a UINavigationController with the MainMenuViewController inside. We don't want that for the iPad, we want our new iPadViewController to be the rootViewController.

So to do this, we need to make a new **MainWindow.xib** for the iPad.

Control-click on the Main group and select **New File**. Select **iOS\User Interface\Empty**, and click Next. Select **iPad** for Device Family, and click Next. Save the file as **MainWindow-iPad.xib**, and click Create.

Open up **MainWindow-iPad.xib** and make the following changes:

- Select **File's Owner**, and in the Identity Inspector set the **Class** to **UIApplication**. We do this because the UIApplication class will be loading the MainWindow-iPad XIB on startup, and it sets itself as the File's Owner.
- Drag in an **Object** from the Object Library. Select it and in the Identity Inspector set the **Class** to **AppDelegate**. We do this because we want to create an instance of the AppDelegate class on startup when this XIB is loaded.
- Control-drag from the **File's Owner** to the **App Delegate** object, and connect it to the **delegate** outlet. We do this because we want the UIApplication's delegate to be the AppDelegate.
- Drag in a **Window** from the Object Library. Control-drag from the **App Delegate** object to the **Window**, and connect it to the **window** outlet. We do this because the App Delegate needs a reference to the main window in application:didFinishLaunching.
- Drag in a **View Controller** from the Object Library. Select it and in the Identity Inspector set the **Class** to **iPadViewController**, and in the Attributes inspector set the **NIB Name** to **iPadViewController**. We do this because we want an instance of our iPadViewController to be created on startup when this XIB is loaded.
- Control-drag from the **Window** to the **iPad View Controller**, and connect it to the **rootViewController** outlet. We do this because we want the iPad View Controller to be the first view controller displayed in the Window on startup.

One final step - we need to tell our app to use this new MainWindow-iPad.xib if it is running on the iPad. To do this, open up **Supporting Files\Shortcuts-Info.plist**, control-click and select **Add Row**, select "**Main nib file base name (iPad)**" in the dropdown, and enter **MainWindow-iPad** for the value.



Key	Type	Value
Localization native development region	String	en
Bundle display name	String	Xcode Shortcuts
Executable file	String	\${EXECUTABLE_NAME}
▶ Icon files	Array	(1 item)
Bundle identifier	String	com.razeware.\${PRODUCT_NAME}Identifier
InfoDictionary version	String	6.0
Bundle name	String	\${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environment	Boolean	YES
Main nib file base name	String	MainWindow
▶ Supported interface orientations	Array	(3 items)
Main nib file base name (iPad)	String	MainWindow-iPad

Compile and run in the iPad simulator, and you should see your basic layout appear!



Of course all sort of craziness occurs when you change the device orientation. So let's start connecting everything to outlets and write some code to fix that! :]

Connections and Orientation

Open **iPadViewController.xib**, bring up the Assistant Editor and make sure it's set to **Automatic** so **iPadViewController.h** shows up. Then control-drag from **left**



UIView down below the @interface, make sure the Connection type is set to **Outlet**, name it **leftView**, and click Connect.

Repeat this for:

- **The middle view:** Connect to an outlet named **middleView**.
- **The right view:** Connect to an outlet named **rightView**.
- **The decoration view:** Connect to an outlet named **decorationView**.
- **The bottom view:** Connect to an outlet named **bottomView**.
- **The favorites image view:** Connect to an outlet named **favoritesImageView**.
- **The favorites "star" button:** Connect to an outlet named **favoritesButton**.
- **The favorites "gear" button:** Connect to an outlet named **settingsButton**.
- **The "background" image view:** Connect it to an outlet named **bgImageView**.

We also need to connect the button taps to methods. So control-drag from the **favorites button** down below the @interface in the Assistant Editor, make sure the Connection type is set to **Action** and the Event is set to **Touch Up Inside**, set the name to **favoritesButtonTapped**, and click Connect.

Repeat for the **settings button**, but connect it to **settingsButtonTapped**.

Now let's add a bit of code so the app handles orientation changes correctly. Add the following new methods to **iPadViewController.m**:

```
- (void)layoutForOrientation:(UIInterfaceOrientation)interfaceOrientation {
    if (UIInterfaceOrientationIsPortrait(interfaceOrientation)) {
        leftView.frame = CGRectMake(19, 73, 355, 433);
        middleView.frame = CGRectMake(393, 73, 355, 433);
        rightView.frame = CGRectMake(768, 73, 355, 433);
        bottomView.frame = CGRectMake(165, 10, 558, 447);
        favoritesButton.frame = CGRectMake(47, 276,
            favoritesButton.frame.size.width,
            favoritesButton.frame.size.height);
        settingsButton.frame = CGRectMake(47, 378,
            settingsButton.frame.size.width,
            favoritesButton.frame.size.height);
    } else {
        leftView.frame = CGRectMake(19, 73, 320, 433);
        middleView.frame = CGRectMake(354, 73, 320, 433);
        rightView.frame = CGRectMake(690, 73, 320, 433);
        bottomView.frame = CGRectMake(190, 10, 785, 191);
        favoritesButton.frame = CGRectMake(14, 126,
            favoritesButton.frame.size.width,
```



```
    favoritesButton.frame.size.height);
    settingsButton.frame = CGRectMake(105, 126,
        settingsButton.frame.size.width,
        favoritesButton.frame.size.height);
}
}

- (void)viewWillAppear:(BOOL)animated {
    NSLog(@"iPadViewController - viewWillAppear");
    [self layoutForOrientation:
        [[UIApplication sharedApplication] statusBarOrientation];
}

- (void)willRotateToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
    duration:(NSTimeInterval)duration {
    NSLog(@"iPadViewController - willRotateToInterfaceOrientation");
    [UIView animateWithDuration:duration animations:^{
        [self layoutForOrientation:toInterfaceOrientation];
    }];
}
```

This is just some simple code to place the various views in the proper place for each orientation. This is necessary because we want to move things around in a particular way where autosizing attributes wouldn't have been enough.

Note that we call this layout routine in `willRotateToInterfaceOrientation:duration`, inside an animation block that is the same duration as the passed-in value for a nice smooth transition.

Compile and run, and if you rotate you should see a nice portrait-mode layout!





Adding the Child View Controllers

Now that we have the basic layout in place, we can finally add our child view controllers into the proper areas on the screen.

As mentioned earlier, let's try this at first without using the new UIViewController containment APIs at all, so we can see what breaks (and why these new APIs are necessary!)

Start by opening up **iPadViewController.h** and modify it to look like the following:

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

@class DictionaryViewController;
@class SearchableShortcutsViewController;
@class FavoritesViewController;
```



```

@class SettingsViewController;

@interface iPadViewController : UIViewController {
    DictionaryViewController * _menusViewController;
    DictionaryViewController * _keysViewController;
    SearchableShortcutsViewController * _allShortcutsViewController;
    FavoritesViewController * _favoritesViewController;
    SettingsViewController * _settingsViewController;
    UINavigationController * _menusNav;
    UINavigationController * _keysNav;
    UINavigationController * _allNav;
}

@property (strong, nonatomic) IBOutlet UIView *leftView;
@property (strong, nonatomic) IBOutlet UIView *middleView;
@property (strong, nonatomic) IBOutlet UIView *rightView;
@property (strong, nonatomic) IBOutlet UIView *decorationView;
@property (strong, nonatomic) IBOutlet UIView *bottomView;
@property (strong, nonatomic) IBOutlet UIImageView
*favoritesImageView;
@property (strong, nonatomic) IBOutlet UIButton *favoritesButton;
@property (strong, nonatomic) IBOutlet UIButton *settingsButton;
@property (strong, nonatomic) IBOutlet UIImageView *bgImageView;
- (IBAction)favoritesButtonTapped:(id)sender;
- (IBAction)settingsButtonTapped:(id)sender;

@end

```

Here we are just creating some instance variables for the various view controllers that we want as children.

Next switch to iPadViewController.m and add the following imports to the top of the file:

```

#import "ShortcutsDatabase.h"
#import "DictionaryViewController.h"
#import "SearchableShortcutsViewController.h"
#import "FavoritesViewController.h"
#import "SettingsViewController.h"

```

Here we're importing the various view controllers, and the object we're using to manage the shortcuts (ShortcutsDatabase) so we can pass the view controllers the appropriate data to display.

Next add the following inside viewDidLoad:

```

self.bgImageView.image = [[UIImage imageNamed:@"bg_200"]
resizableImageWithCapInsets:UIEdgeInsetsMake(0, 0, 0, 0)];

```



```
_menusViewController = [[DictionaryViewController alloc]
    initWithNibName:nil bundle:nil];
_menusViewController.navigationItem.title = @"Shortcuts By Menu";
_menusViewController.dict =
    [ShortcutsDatabase sharedDatabase].shortcutsByMenu;
_menusViewController.keys = [ShortcutsDatabase sharedDatabase].menusArray;

_keysViewController = [[DictionaryViewController alloc]
    initWithNibName:nil bundle:nil];
_keysViewController.navigationItem.title = @"Shortcuts By Key";
_keysViewController.dict =
    [ShortcutsDatabase sharedDatabase].shortcutsByKey;

_allShortcutsViewController = [[SearchableShortcutsViewController alloc]
    initWithNibName:nil bundle:nil];
_allShortcutsViewController.navigationItem.title = @"All Shortcuts";
_allShortcutsViewController.shortcutsDict = [ShortcutsDatabase
    sharedDatabase].shortcutsByKey;

FavoritesViewController = [[FavoritesViewController alloc]
    initWithNibName:nil bundle:nil];

_settingsViewController = [[SettingsViewController alloc]
    initWithNibName:nil bundle:nil];

_menusNav = [[UINavigationController alloc]
    initWithRootViewController:_menusViewController];
_keysNav = [[UINavigationController alloc]
    initWithRootViewController:_keysViewController];
_allNav = [[UINavigationController alloc]
    initWithRootViewController:_allShortcutsViewController];
```

We first set the background image to a resizable tiled image - for more details on this new API, see the UIKit Customization chapter.

We then create an instance of each view controller to display, passing in the appropriate data to display when necessary.

Finally, we create three navigation controllers to wrap the view controllers that have multiple levels of detail.

Continue by adding the following code at the end of viewDidLoad:

```
_menusNav.view.frame = leftView.bounds;
_keysNav.view.frame = middleView.bounds;
FavoritesViewController.view.frame = bottomView.bounds;
_settingsViewController.view.frame = bottomView.bounds;
_allNav.view.frame = rightView.bounds;
[leftView addSubview:_menusNav.view];
```



```
[middleView addSubview:_keysNav.view];
[bottomView addSubview:_favoritesViewController.view];
[rightView addSubview:_allNav.view];
```

Here we set the bounds of the root view of each view controller to the size of the placeholder view we made for each. Then, we add the root view of each view controller as a subview to the placeholder views we made.

Note that for the bottom view, we only add the favorites view controller's view as a subview. Later we'll swap this out for the settings view controller.

Remember... directly adding the view controller's view as a subview like this isn't quite enough! We also need to use the new UIViewController containment APIs, but first I wanted to do things this way to show you why it's necessary.

One more modification. Implement the `favoritesButtonTapped:` and `settingsButtonTapped:` methods as follows:

```
- (IBAction)favoritesButtonTapped:(id)sender {
    [[ShortcutsDatabase sharedDatabase] playClick];
    [UIView transitionWithView:bottomView duration:0.5
        options:UIViewAnimationOptionTransitionFlipFromBottom animations:^{
            [_settingsViewController.view removeFromSuperview];
            _favoritesViewController.view.frame = bottomView.bounds;
            [bottomView addSubview:_favoritesViewController.view];
        } completion:NULL];
}

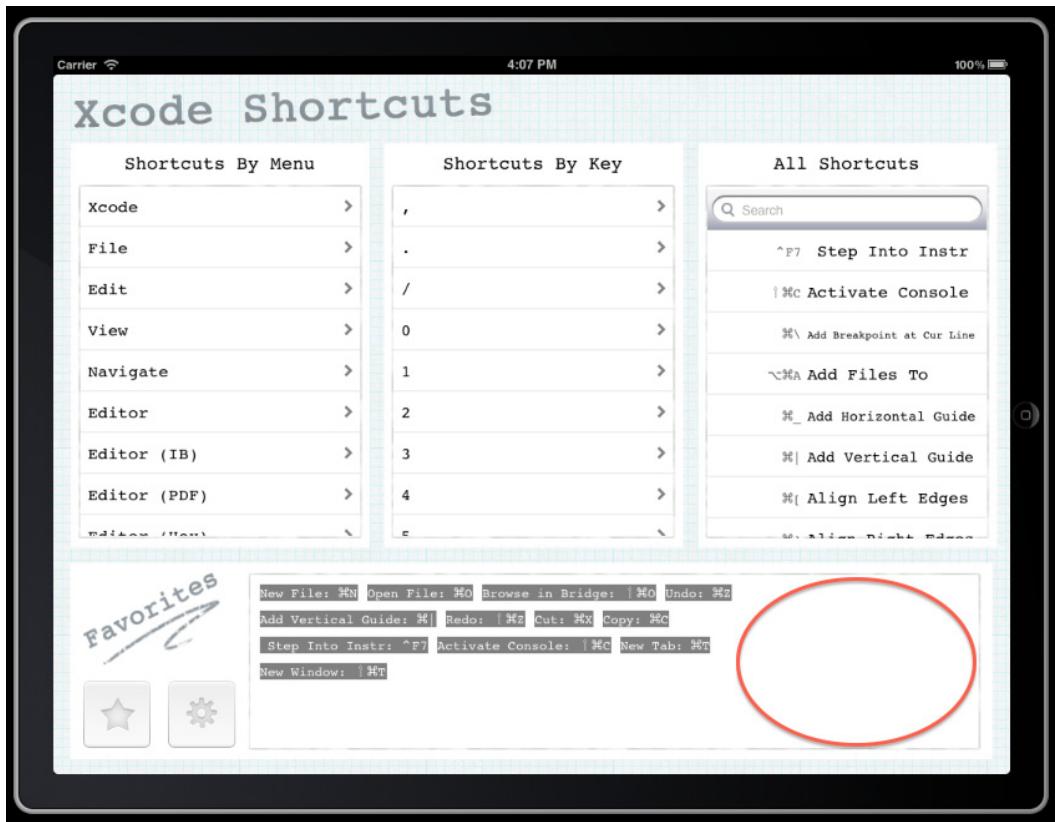
- (IBAction)settingsButtonTapped:(id)sender {
    [[ShortcutsDatabase sharedDatabase] playClick];
    [UIView transitionWithView:bottomView duration:0.5
        options:UIViewAnimationOptionTransitionFlipFromBottom animations:^{
            [_favoritesViewController.view removeFromSuperview];
            _settingsViewController.view.frame = bottomView.bounds;
            [bottomView addSubview:_settingsViewController.view];
        } completion:NULL];
}
```

When the buttons are tapped, we simply swap the appropriate view controller to the front of the bottom view.

Again, this isn't quite the right way to do this in iOS 5, for reasons we'll see shortly! :]

Alright, compile and run and try this out. It seems to work, except for one subtle issue - the favorites view doesn't seem to handle orientation changes properly! If you rotate the app, you'll notice the words don't re-layout to take advantage of the extra space like they did in the iPhone version.



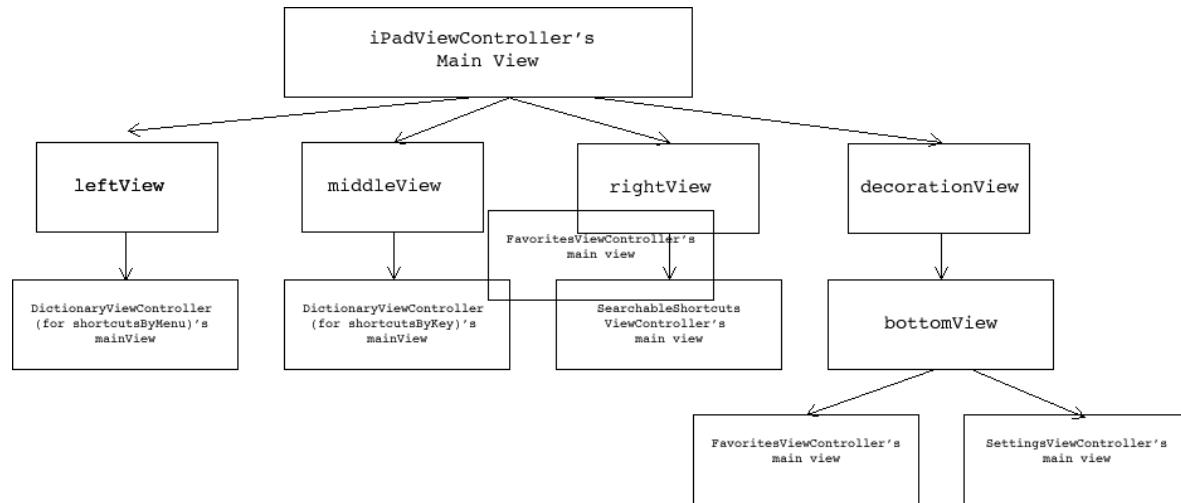


Wonder why this is? This leads us into a discussion of UIViewController Containment!

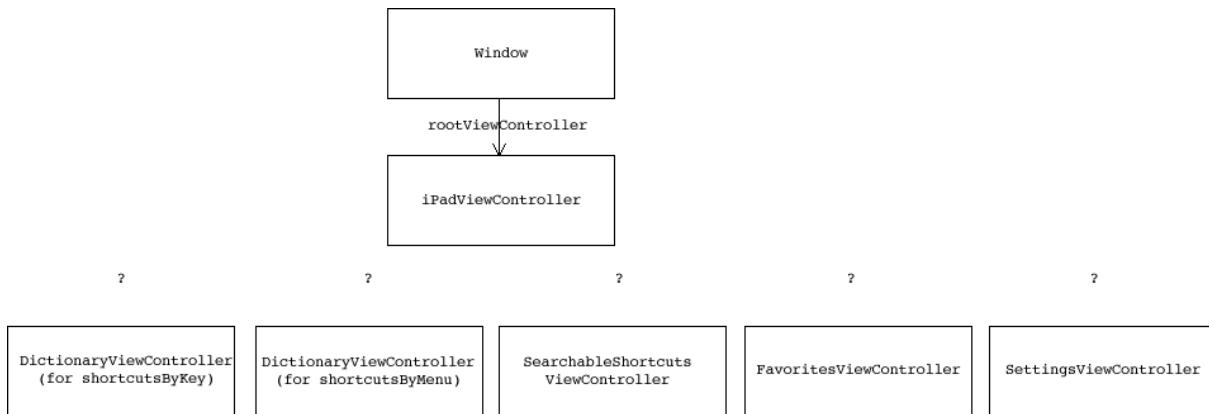
UIViewController Containment Overview

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 our iPadViewController:





It turns out that the same way views have parents, view controllers have parents as well. The only problem is, we haven't written any code to let the OS know that the iPadViewController should be the parent of our 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 `willRotateToInterfaceOrientation`.

In fact, this is why we're having the problem we just identified. The favorites view controller does its relayout when its `willRotateToInterfaceOrientation` 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 `willRotateToInterfaceOrientation` 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 new iOS 5 APIs!



The Containment Implementation

The UIViewController 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 our case, we want to call this from iPadViewController on all of the child view controllers within (the two dictionary view controllers, the searchableshortcuts view controller, the favorites view controller, and the settings view controller). Once the OS knows the hierarchy, it can properly forward on the lifecycle and rotation methods.
2. **removeFromParentViewController:** - To remove a view controller from your list of children, you call this method. We don't need to do this because the view controllers always stay as our children, but you might find this useful if you are making something similar to a Navigation Controller, where you have a detail view controller as a child temporarily, but when you pop out you're done with it so you can remove it from your list of children.
3. **transitionFromViewController:toViewController:duration:options:animations:completion:** - This is the new way to replace a view with an alternative view, or move a subview to the front like we're doing. By using this method, the view controller lifecycle messages are forwarded properly.
4. **willMoveToParentViewController:** - This is automatically called by addChildViewController:, and you might call this to tell a view controller it's about to be removed and has no parent (pass in nil). Also, inside a view controller, if you ever need to have certain code happen when the 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 - and didMoveToParentViewController once it animates in. This will have the effect of calling viewDidAppear on your view controller (even if it's covered up by another view!) What viewDidAppear really means is the view has made its way into the window's hierarchy, so design with that in mind.

So let's try this out! We'll start with hooking up the view controller hierarchy by using addChildViewController. Add the following code to the bottom of viewDidLoad:

```
[self addChildViewController:_menusNav];
[_menusNav didMoveToParentViewController:self];
[self addChildViewController:_keysNav];
```



```
[_keysNav didMoveToParentViewController:self];
[self addChildViewController:_favoritesViewController];
[_favoritesViewController didMoveToParentViewController:self];
```

Note we do not add the settings view controller as a child view controller yet, because its view has not yet been added to the Window's hierarchy.

Next replace the favoritesButtonTapped and settingsButtonTapped methods with the following:

```
- (IBAction)favoritesButtonTapped:(id)sender {
    [[ShortcutsDatabase sharedDatabase] playClick];
    [self addChildViewController:_favoritesViewController];
    [_self transitionFromViewController:_settingsViewController
        toViewController:_favoritesViewController duration:0.5
        options:UIViewAnimationOptionTransitionFlipFromBottom
        animations:^{
            [_settingsViewController.view removeFromSuperview];
            _favoritesViewController.view.frame = bottomView.bounds;
            [bottomView addSubview:_favoritesViewController.view];
        } completion:^(BOOL finished) {
            [_favoritesViewController
                didMoveToParentViewController:self];
            [_settingsViewController removeFromParentViewController];
        }];
}

- (IBAction)settingsButtonTapped:(id)sender {
    [[ShortcutsDatabase sharedDatabase] playClick];
    [self addChildViewController:_settingsViewController];
    [_self transitionFromViewController:_favoritesViewController
        toViewController:_settingsViewController duration:0.5
        options:UIViewAnimationOptionTransitionFlipFromBottom
        animations:^{
            [_favoritesViewController.view removeFromSuperview];
            _settingsViewController.view.frame = bottomView.bounds;
            [bottomView addSubview:_settingsViewController.view];
        } completion:^(BOOL finished) {
            [_settingsViewController
                didMoveToParentViewController:self];
            [_favoritesViewController removeFromParentViewController];
        }];
}
```

Here we add the new view controller as a child view controller, and call didMoveToViewController when the transition is complete. Similarly, we remove the old view controller from its parent when the transition completes as well.



Pretty easy, huh? Compile and run the app, and you'll notice if you rotate the app, now the favorites view controller properly resizes! This is because the `willRotateToInterfaceOrientation` method is forwarded properly now that the view controller hierarchy is set up.

Where To Go From Here?

As you can see, using view controller containment is pretty easy these days, with the new iOS 5 APIs.

You should now know enough to implement container view controllers on your own, but if you want to learn more, check out the "Implementing a Container View Controller" section of the `UIViewController` reference, or watch WWDC 2011's "Implementing UIViewController Containment" video.

And the best part is you're no longer officially restricted to just using the built-in 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!



Working with JSON in iOS 5

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"]
```

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}
```

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 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

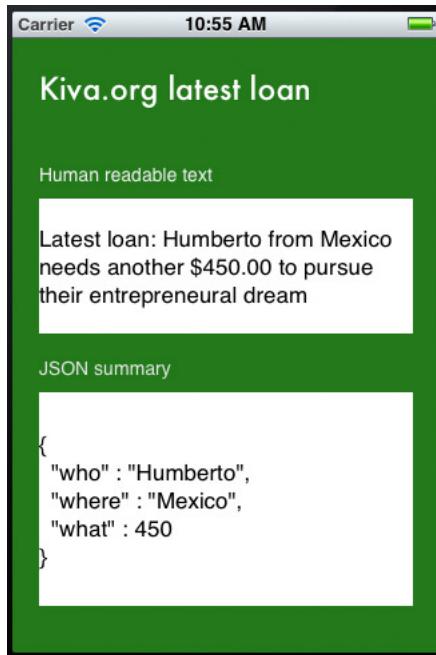
If you've had to parse JSON in your iOS apps in the past, you've probably used a third party library such as JSON Framework.

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 we're going to get hands-on experience with the new native JSON support. We're 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 we're going 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, click Next and save the project by clicking Create.

Let's do a little bit of clean up first - 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"

@implementation ViewController

@end

```

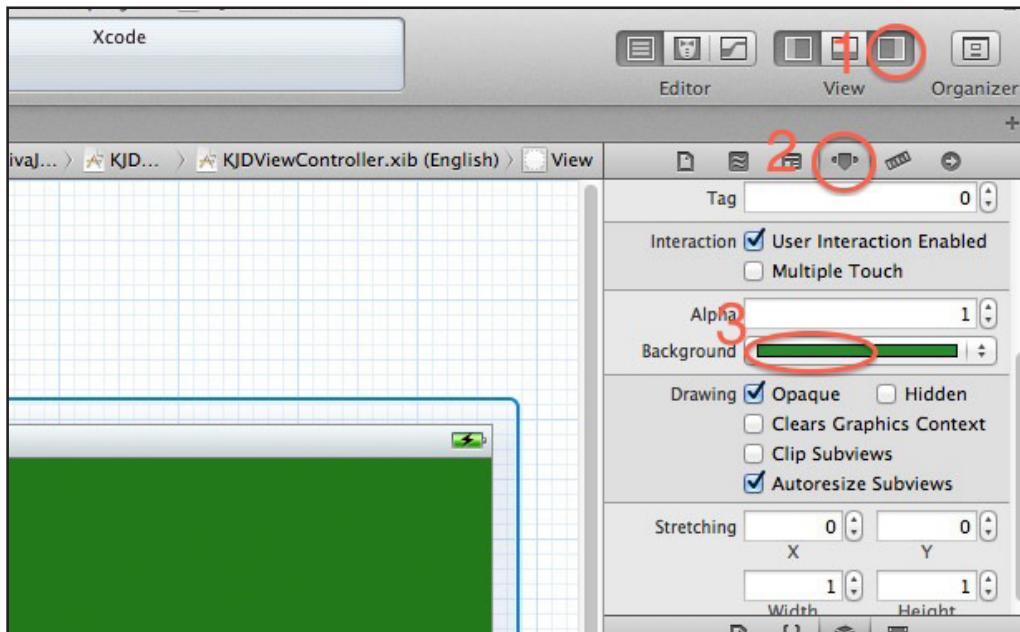
In the first line of code we define a macro that gives us back a background queue - I like having a kBgQueue shortcut for that, so I can keep my code tighter.

In the second line of code we create a macro named kLatestKivaLoansURL which returns us 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 if you want - you'll see Kiva.org's list of currently fundraising loans in JSON format. We're going to use this API to read the list of loans, take the latest one and show the information on the screen.

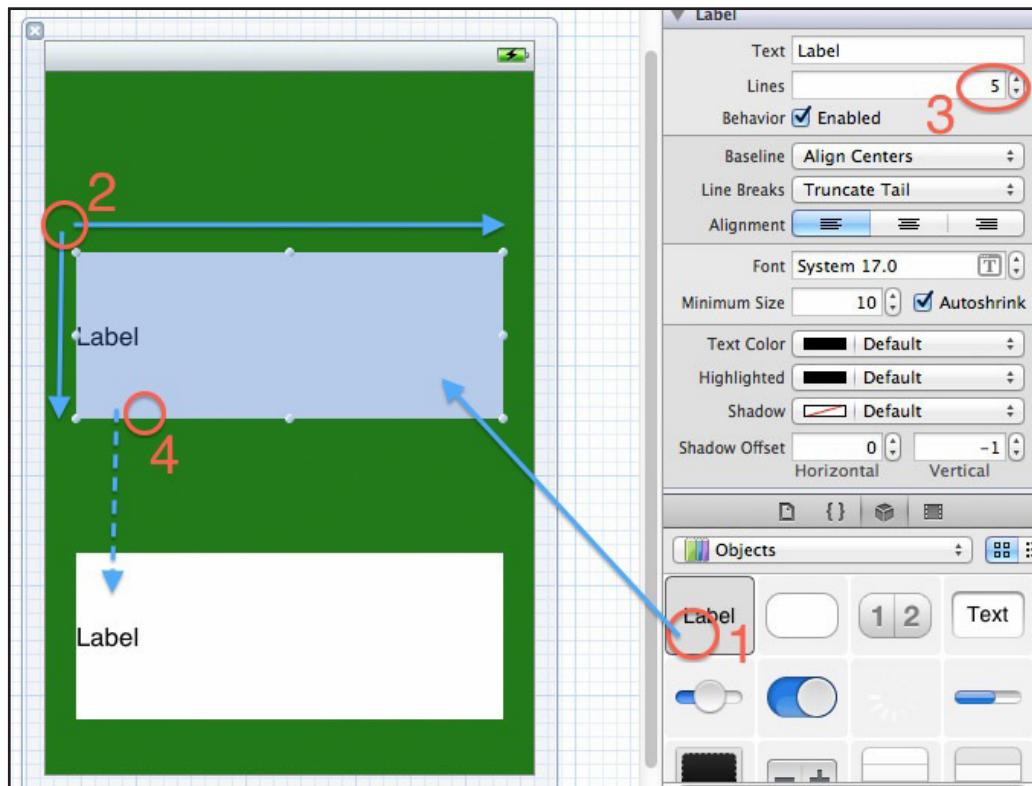
Let's make a little detour and design the application's UI in Interface Builder real quick.

Open **ViewController.xib** in the Project Navigator. This app is supposed to be positive and life-changing, so we need to do something about that background! Select it 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).

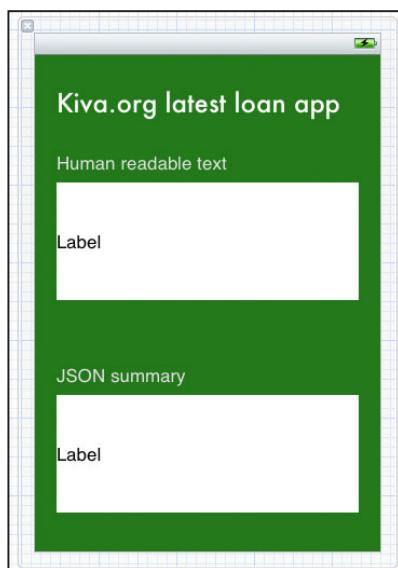


Now grab a label from the Object library and drop it inside the already open view (1). Resize the label so it fits about 4 lines of text and takes almost the screen's

width (2). Then from the Attributes Inspector make the following changes: set "Background" to white, 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 our labels to a couple of `IBOutlets` in our class. Switch to **ViewController.h** and add two instance variables inside the interface:

```
@interface ViewController : UIViewController {
    IBOutlet UILabel* humanReadable;
    IBOutlet UILabel* jsonSummary;
}
```

Then open up **ViewController.xib** again. Control-drag from "File's owner" onto the 1st 5-line label and from the popup menu choose **humanReadable**.

Again while holding the "ctrl" key drag with the mouse from "File's owner" onto the 2nd 5-line label and from the popup menu choose **jsonSummary**.

That concludes the project setup - we're ready to start coding!

Parsing JSON from the Web

The first thing we need to do is download the JSON data from the web. Luckily, with GCD we can do 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 we defined `kBGQueue` as a macro which gives us a background queue?

Well this bit of code makes it so that when `viewDidLoad` is called, we run a block of code in this background queue to download the contents at the Kiva loans URL.

When `NSData` has finished fetching data from the Internet we call `performSelectorOnMainThread:withObject:waitUntilDone:` so we can update the application's UI. We haven't written `fetchedData:` yet but will do so shortly.

Remember 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 we 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 our case the JSON file is relatively small so we'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 objectForKey:@"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. In line 1, we get an NSDictionary from the JSON data. In line 2, we get an NSArray latestLoans which is the loans key in the top JSON dictionary.

Finally in line 3, we dump latestLoans to the console, so we're 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 to the Screen

We're 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 objectAtIndex:0];

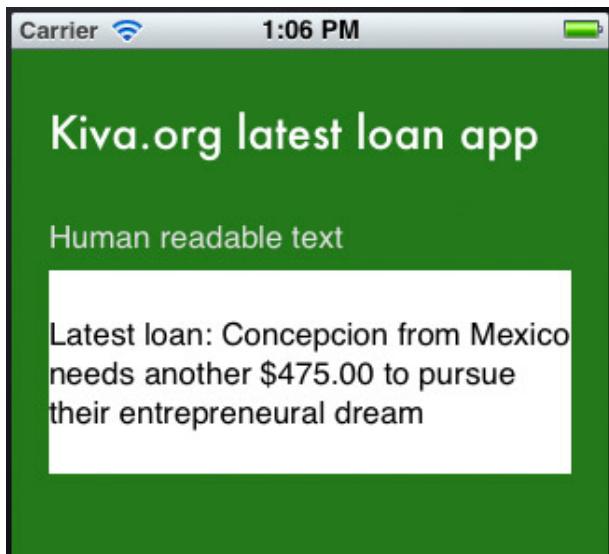
// 2) Get the funded amount and loan amount
NSNumber* fundedAmount = [loan objectForKey:@"funded_amount"];
NSNumber* loanAmount = [loan objectForKey:@"loan_amount"];
float outstandingAmount = [loanAmount floatValue] -
    [fundedAmount floatValue];

// 3) Set the label appropriately
humanReadable.text = [NSString stringWithFormat:@"Latest loan: %@
from %@ needs another $%.2f to pursue their entrepreneurial dream",
[loan objectForKey:@"name"],
[(NSDictionary*)[loan objectForKey:@"location"]
objectForKey:@"country"],
outstandingAmount];
```

The `latestLoans` array is a list of dictionaries, so (1) we get the first (and latest) loan dictionary and (2) we fetch few values about the loan. Finally (3) we set the text of the 1st label in the UI.

OK! Let's have a look - hit Run and see what comes up:





Of course the information you see will be different as Kiva adds loans constantly - but it's clear we achieved what we wanted, we parsed JSON data and visualized some human readable info.

Generating JSON Data

Now let's do the opposite. From the loan NSDictionary that we now have we'll build some JSON data, which we will be able to send over to a server, another app, or do with it whatever else we want.

Add this code to the end of the fetchedData: method:

```
//build an info object and convert to json
NSDictionary* info = [NSDictionary dictionaryWithObjectsAndKeys:
    [loan objectForKey:@"name"],
    @"who",
    [(NSDictionary*)[loan objectForKey:@"location"]]
        objectForKey:@"country"],
    @"where",
    [NSNumber numberWithFloat: outstandingAmount],
    @"what",
    nil];

//convert object to data
NSData* jsonData = [NSJSONSerialization dataWithJSONObject:info
    options:NSJSONWritingPrettyPrinted error:&error];
```

Here we build an NSDictionary called `info` where we store the loan information as who, where, and what in different keys and values.



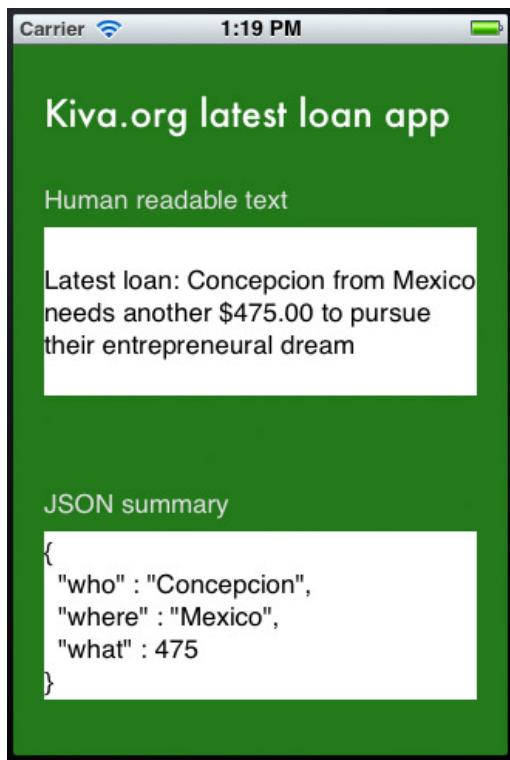
Then we call `dataWithJSONObject:options:error:` - the opposite to the JSON API we just used before. It takes in an object and turns it into JSON data.

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 our job of turning `info` into JSON is finished, but we can't be sure before we see that it is actually so. Let's show the JSON into our 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:` we easily get the text representation of our JSON data and we 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 - we're using it's Objective-C, so we 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.

But basically, we define 2 methods on 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 =  
[NSDictionary dictionaryWithObjectsAndKeys:  
 @"orange",@"apple",@"banana",@"fig",nil];  
NSData* json = [information toJSON];
```

Pretty cool and 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 we go though, I want 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 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!



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 (no pun intended) 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" is 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.

We'll start the project by creating a custom **UIView** class which will have **CAEmitterLayer** as its layer. You can actually achieve this very very easy by overwriting the **+ (Class)layerClass** method of the **UIView** class and returning a **CAEmitter** class. Pretty cool!

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
```

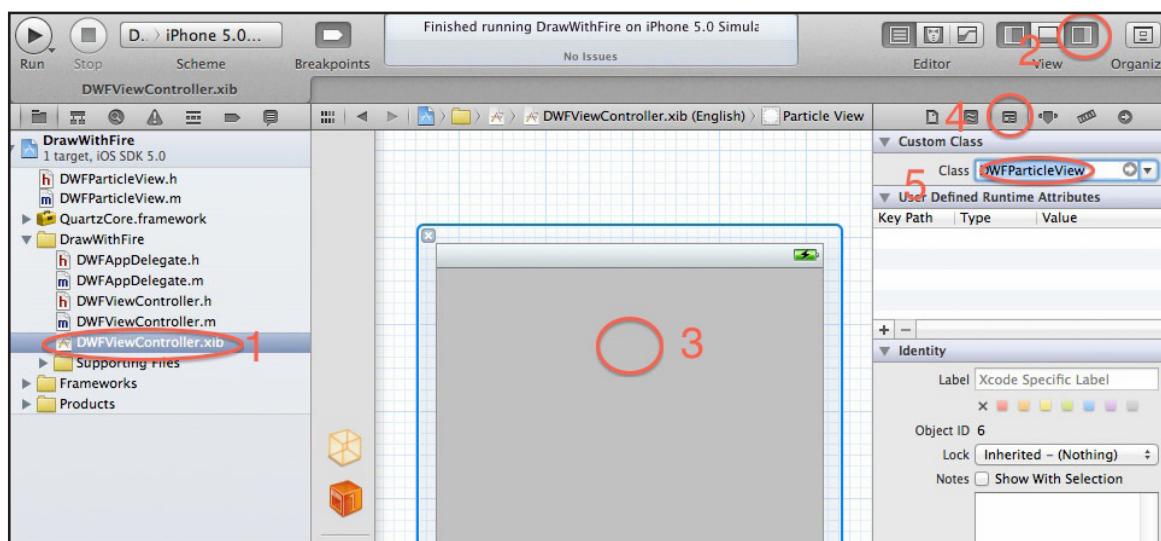
Let's go over the initial code:

1. We create a single private instance variable to hold our **CAEmitterLayer**.



2. In `awakeFromNib` we set `fireEmitter` to be the view's `self.layer`. We store it in the `fireEmitter` instance variable we created, because we're going to set a lot of parameters on this later on.
- 3.`+ (Class)layerClass` is the `UIView` method which 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:



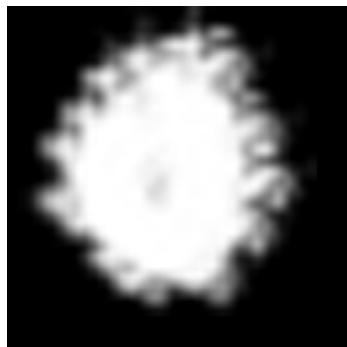
1. Make sure the Utilities bar is visible (the highlighted button on the image above should be pressed down).
2. Select the gray area in the Interface builder - this is the view controller's root view.
3. Click the Identity Inspector tab
4. In the Custom class panel enter `DWFParticleView` in the text 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 for your particles. 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 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 in the colors we'd 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"];

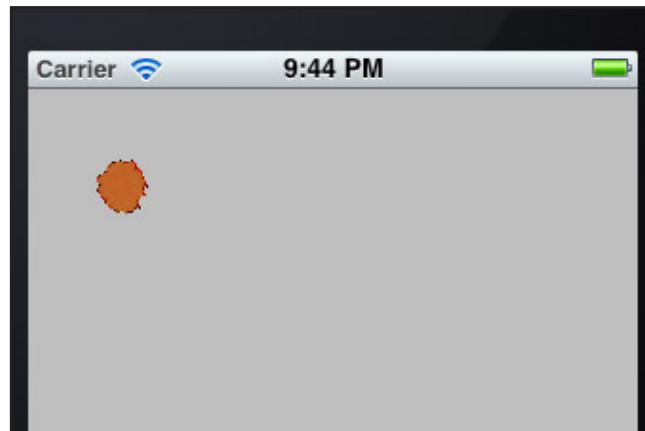
//add the cell to the layer and we're done
fireEmitter.emitterCells = [NSArray arrayWithObject:fire];
```

We're creating a cell instance and setting up few properties. Then we 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!

Next set some properties on the CAEmitterCell. 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. We 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 our case, a particle will live from 2.5 to 3.5 seconds.
- **color**: The color tint to apply to the contents. We choose an orange color here.
- **contents**: The contents to use for the cell, usually a CGImage. We set it to our particle image.
- **name**: You can 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 we might like. You might barely even be able to tell it's doing something, it just looks like an orange splotch!

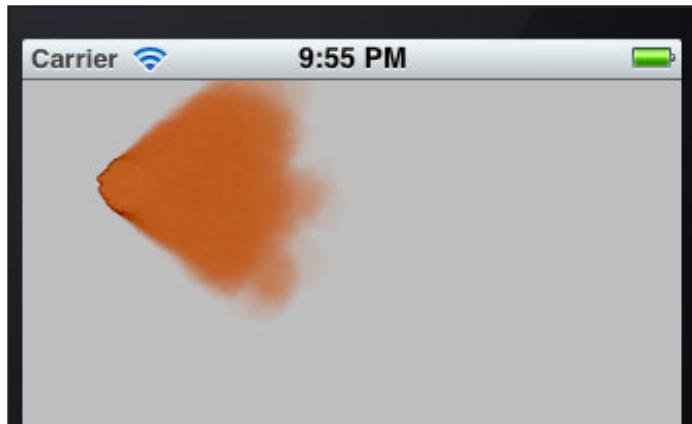
Let's change this a bit to make the particle effect more dynamic. Add this code just before calling `setName:` on the cell:

```
fire.velocity = 10;  
fire.velocityRange = 20;  
fire.emissionRange = M_PI_2;
```

Here we're setting the following new properties on the CAEmitterCell:

- **velocity**: The particles's velocity in points per second. This will make our 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 the a range, it will be +/- 45 degrees).

Compile and run to check out the progress:



OK this is better - we're 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:

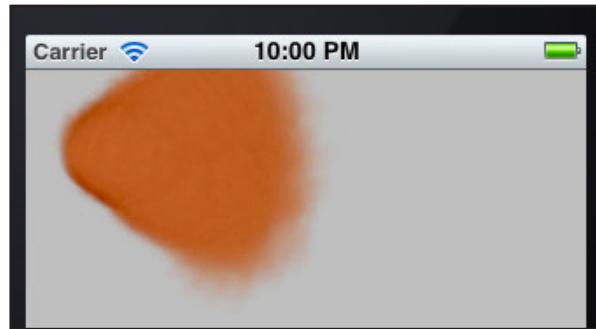
```
fire.scaleSpeed = 0.3;  
fire.spin = 0.5;
```

Here we set two more properties on the CAEmitterCell:



- **scaleSpeed:** The rate of change per second at which the particle changes its scale. We set this to 0.3 to make the particles grow over time.
- **spin:** Sets the rotation speed of each particle. We set this to 0.5 to give the particles a nice spin.

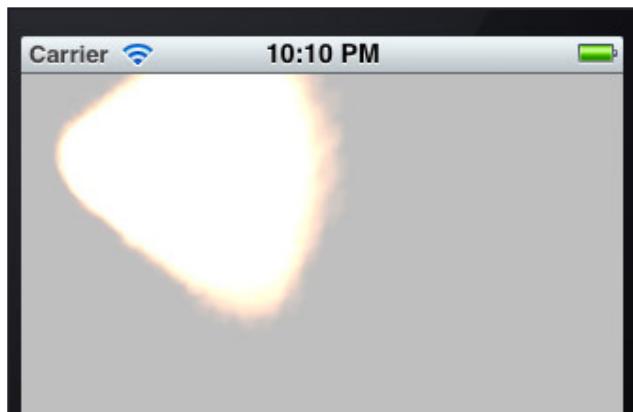
Hit Run one more time:



By now we have something like a rusty smoke - and guess what? CAEmitterCell has a lot more properties to tweak, so kind of the sky is the limit here. But we're 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 our rusty smoke into a boiling ball of fire. Hit Run and check the result:



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 we'll 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 we'll need an outlet for our view so we can tweak it from the view controller. Open **DWFViewController.h** and replace the code with the following:

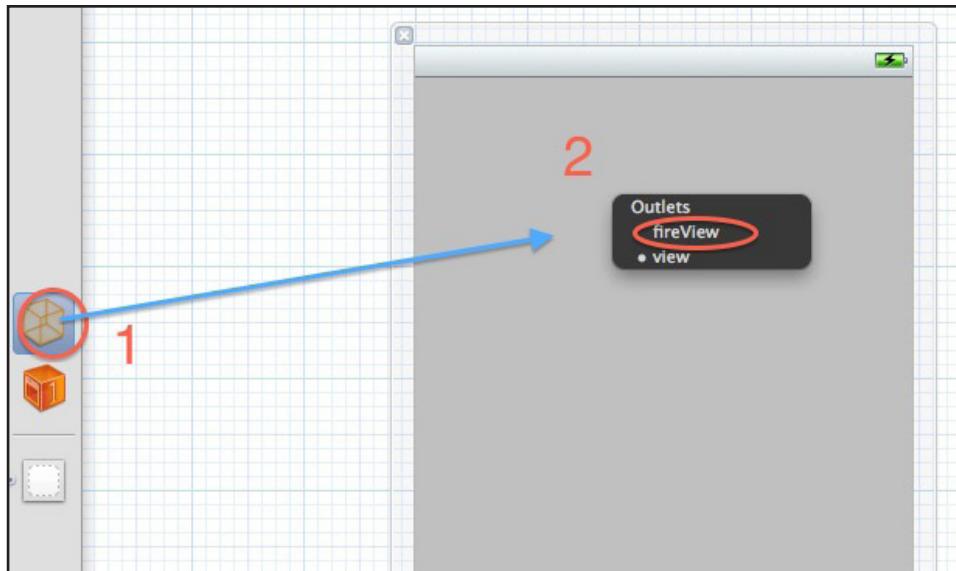
```
#import <UIKit/UIKit.h>
#import "DWFParticleView.h"

@interface DWFViewController : UIViewController
{
    IBOutlet DWFParticleView* fireView;
}
@end
```

As you see we import our custom view class and we declare an instance variable for the `DWFParticleView`.

Next open **DWFViewController.xib** and control-drag from the **File's Owner** to the root **view**, and choose **fireView** from the popup:





Now we 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.



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 the emitting of particles
    [fireEmitter setValue:[NSNumber numberWithInt:isEmitting?200:0]
        forKeyPath:@"emitterCells.fire.birthRate"];
}
```

Here we're using the `setValue:forKeyPath:` method so we can modify a cell that's already been added to the emitter by the name we set earlier. We use "emitterCells.fire.birthRate" for the keypath, which means the `birthRate` property of the cell named `fire`, found in the `emitterCells` array.

Finally we'll 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
- Add burning flames behind all your text labels in all your apps ;]

Using the iOS Dictionary

by Marin Todorov

In iOS 5, Apple has introduced a new system-wide dictionary service. The best part about it is there's 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 we're going experiment with the new dictionary APIs by creating a very simple game. The user will be presented with a picture and 3 words and he needs to choose the correct word for the picture. When he makes his choice, he'll be presented with the dictionary entry for the correct word.

In the process of this tutorial, we're also going to cover some new iOS 5 table view magic. I'll show you how to design a cell in Interface Builder, have it as a separate XIB file, and let Cocoa Touch load it automatically and use it for your table cells.

This tutorial requires a device for testing, because at the time of writing the dictionary is not available in the Simulator.

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" is checked (leave the other checkboxes unchecked). Click Next and save the project by clicking Create.

We'll show the pictures and the possible answers in the table view, and when the user taps a cell we'll push a new dictionary view on top. Thus we won't need the details view controller Xcode has created for us by default. So go ahead and delete the following files:

- GTWDetailViewController.h
- GTWDetailViewController.m
- GTWDetailViewController.xib

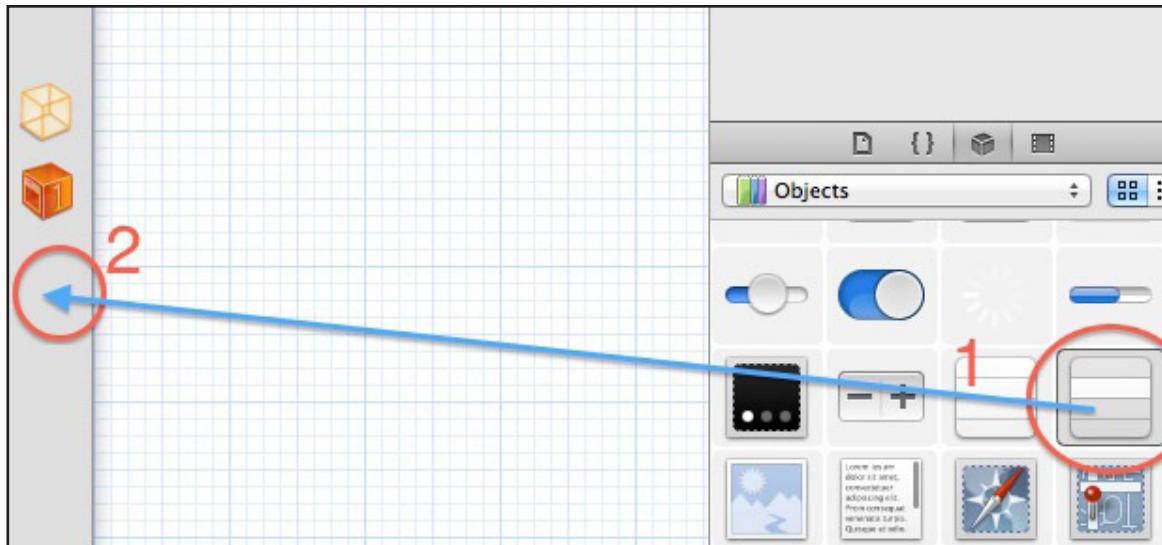
Clean-up is almost finished, let's move on to designing the user interface.

We're going to design a custom table view cell in Interface Builder and let our table automatically fetch it when it needs it.

Create a new file with the **iOS\User Interface\Empty** application template, select **iPhone** for the device family and name it **MyCell.xib**.

We have now an empty XIB file. To be able to use it as a template for a table cell, it needs to have a single UITableViewCell view at the top of its hierarchy. From the Object Library on the right drag the cell object (1) to the XIB's hierarchy strip (2). This will add a table view cell to the design area - cool!

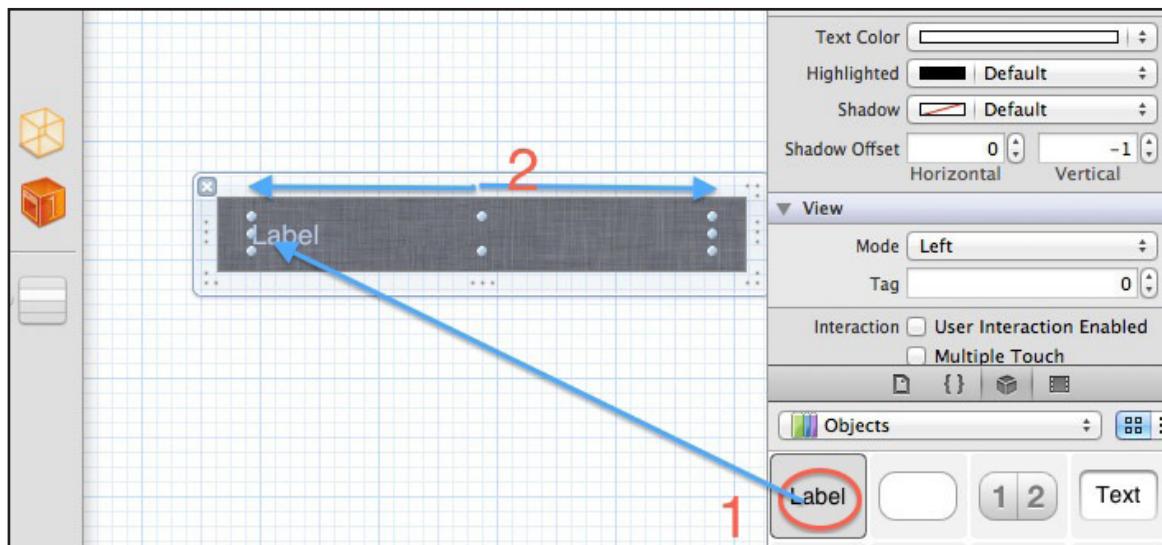




Now you can customize the cell view as you prefer in the Attribute Inspector. I chose to set the "Background" of the cell to "ScrollView Textured Background Color" to give my table cell a funky fabric style background.

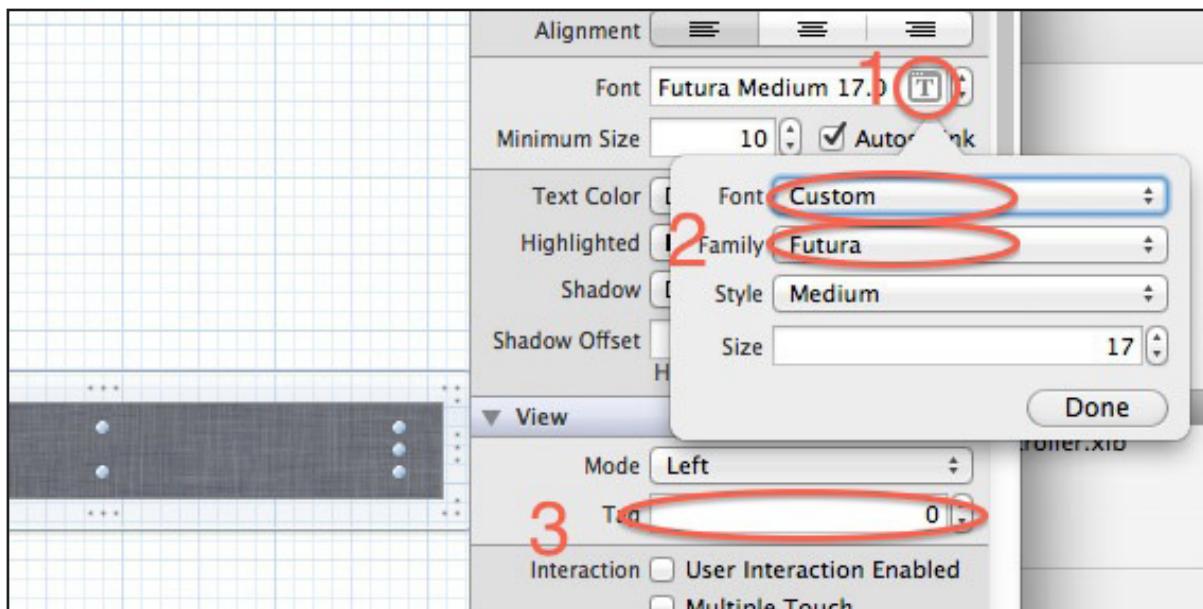
Next find the field "Identifier" and enter the text "MyGreatCell" - this will be the cell's identifier usable from within the app code.

Let's also add a custom UILabel to the cell, so we can do all the UI setup right here in the Interface Builder: Drag a UILabel to the cell and resize it so it fits the table cell view.

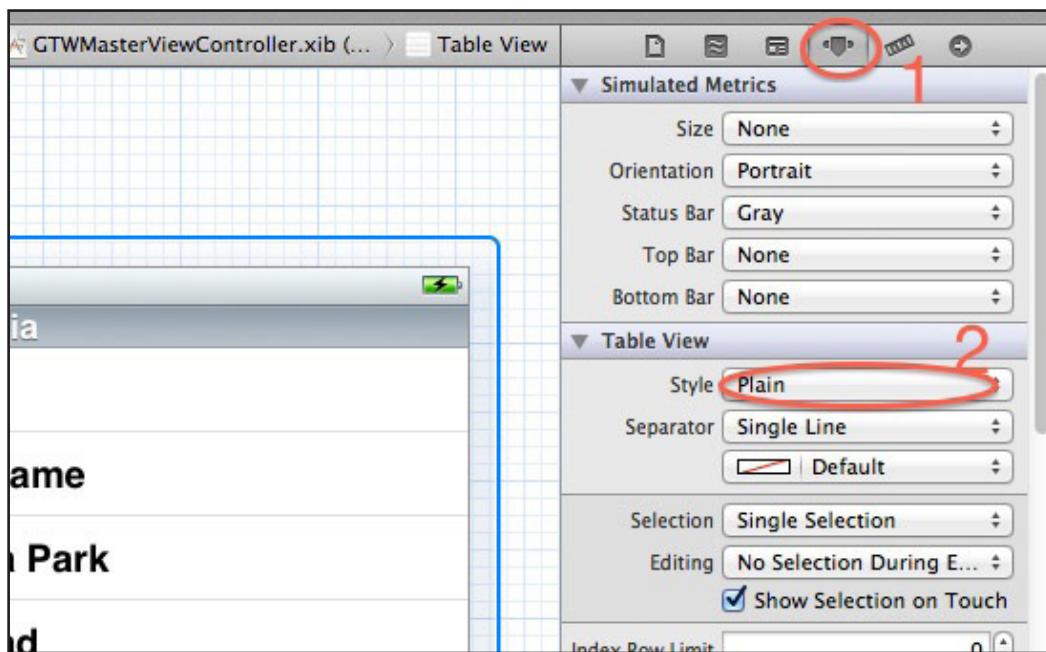


While the label is selected, click on the button inside the "Font" field (1) and from the popup dialogue select "Custom" as Font and "Futura" as Family (2). This will give us a nice looking label!

Finally inside the Tag field (3) enter the value "20". We're going to be accessing the custom label by its tag later on from our code.



Since we're already in Interface Builder, open **GTWMasterViewController.xib** and select the table view. From the Attributes Inspector (1) select "Grouped" for Style (2).



That's about all the interface setup we'll need to do for the project. Good job!

Adding the Implementation

Now it's finally time to move on to code!

Open **GTWMasterViewController.h** and replace it with the following:

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

@interface GTWMasterViewController : UITableViewController
@end
```

If you ask me - this is exactly what the programming gods intended interface files to be - pure cleanliness! In iOS 5 you can hide everything you really don't need/want to expose.

So let's move on to the implementation and see where we declare our instance variables and properties. Open **GTWMasterViewController.m** and replace all the boilerplate code with:

```
#import "GTWMasterViewController.h"

#define kCellTitleTag 20

//Interface declarations
@interface GTWMasterViewController()
{
    int curWord;
}
@property (strong) NSArray* objects;
@property (strong) NSArray* answers;
@end

//Implementation declarations
@implementation GTWMasterViewController

@synthesize objects, answers;

@end
```

So what's going on in this implementation file?

1. First we declare a constant. We'll use this to access our custom label in our custom table cell.
2. Then there's an interface declaration with an empty name category (this also being called class extension). A new feature in Objective-C is that you can now also declare instance variables in the interface categories. Thus, for any instance



variables you don't want declared in your actual interface file, just move them here!

3. Still in the interface section - you see 2 properties, but no corresponding instance variables for them. The compiler now takes care of this itself, hooray!
4. In the implementation section you still need to call `@synthesize` for your properties, but now there's also auto-complete for their names, so that's very cool.

As we're now clear with the code we can move on to the app logic. We'll store our list of words in the `objects` property and the possible answers in the `answers` property. Let's do that - add this method to the implementation body:

```
- (id)initWithNibName:(NSString *)NibNameOrNil  
    bundle:(NSBundle *)nibBundleOrNilOrNil  
{  
    self = [super initWithNibName:nibNameOrNilOrNil  
        bundle:nibBundleOrNilOrNil];  
    if (self) {  
        self.title = @"Guess the word";  
  
        self.objects = [NSArray arrayWithObjects:  
            @"backpack",  
            @"banana",  
            @"hat",  
            @"pineapple",  
            nil];  
  
        self.answers = [NSArray arrayWithObjects:  
            [NSArray arrayWithObjects: @"backpack",  
                @"bag", @"chair", nil],  
            [NSArray arrayWithObjects: @"orange",  
                @"banana", @"strawberry", nil],  
            [NSArray arrayWithObjects: @"hat", @"head",  
                @"hut", nil],  
            [NSArray arrayWithObjects: @"apple",  
                @"poppler", @"pineapple", nil],  
            nil];  
        curWord = 0;  
  
        [self.tableView registerNib:[UINib nibWithNibName:@"MyCell"  
            bundle:[NSBundle mainBundle]]  
            forCellReuseIdentifier:@"MyGreatCell"];  
    }  
  
    return self;  
}
```

Here we override `initWithNibName:bundle:` and do some setup:



- `self.title` sets the navigation bar title.
- `self.objects` is the word list for our game.
- `self.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)
- Finally we're using the new iOS 5 method on the table view called `registerNib:forCellReuseIdentifier:`. This is a fancy way of saying, "Dear table view, for cell identifier "MyGreatCell" if there's no existing cell with that identifier to reuse, load the cell from a NIB file called "MyCell", and please do it every time just by yourself without any further directions". Yes - it's that easy!

Good job so far. If you need a rest - do take five!

Setting Up the Table

Next we're going to add few methods to configure the table view. We want 1 section with a large header on top where we're going to show the image of the current word to guess. Also we'll want 3 rows in the table where we'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
}

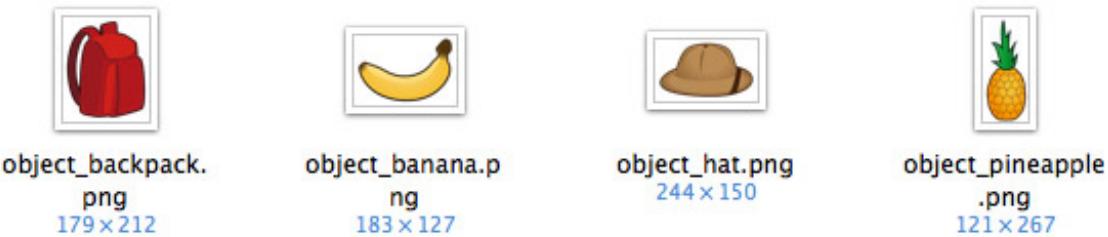
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return 3; //we show 3 possible answers
}

-(CGFloat)tableView:(UITableView *)tableView
    heightForHeaderInSection:(NSInteger)section
{
    return 150.0; //the image of the word is 150px high
}
```

Now in the header of our only section we're going to be showing images called "object_banana.png", "object_hat.png" and so forth. The images we are going to use are created by Vicki Wenderlich and released under the 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 - here's the new code that is specific for iOS 5:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"MyGreatCell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    // no longer need to check for nil and create one otherwise
    // anymore!

    UILabel* cellLabel = (UILabel*)[cell viewWithTag:kCellTitleTag];
    cellLabel.text = [(NSArray*)[self.answers objectAtIndex:indexPath.row]
                      objectAtIndex: indexPath.row];
    cellLabel.textColor = [UIColor whiteColor];
    return cell;
}
```

Let's go over this code :

1. The first two lines you should know very well from every tableView:cellForRowAtIndexPath you've ever written :]
2. Next there's a funky comment - at this point you would normally check if cell is nil and if so do setup and configuration. Here the iOS 5 magic comes into play - if there's no cell to dequeue the table view loads MyCell.xib, gets the top view and fetches it as the cell!
3. Finally - we access our custom label by its tag and load its text from self.answers.

Now we have all the necessary methods to render the table with answers - hit Run in Xcode and see the result! Pretty cool, eh? Note these slick custom cells - entirely designed in Interface Builder!



Now with just one more piece of code we'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 objectAtIndex: curWord]];
    UIImageView* img = [[UIImageView alloc] initWithImage:
    [UIImage imageNamed:imgName]];
    img.frame = CGRectMake(0, 0, 150, 150);
    img.contentMode = UIViewContentModeScaleAspectFit;
    return img;
}
```

Inside `tableView:viewForHeaderInSection` we 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

We want to make the dictionary view controller appear when the player taps a word in the table. In order to have this we'll need a `tableView:didSelectRowAtIndexPath:` method.

When the player taps a row, we'll get the row index and compare the text of the selected cell to the current picture we're showing. If they match, we'll set the text color to green and if not - to red. Finally, we'll show the dictionary definition of the answer the player gave, so he can understand his mistake, or read more about the word he guessed.

So add this method to the implementation:

```
- (void)tableView:(UITableView *)tableView  
didSelectRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    NSString* word = [self.objects objectAtIndex: curWord];  
  
    UITableViewCell* cell = [tableView cellForRowAtIndexPath:  
        indexPath];  
    [cell setSelected:NO];
```



```
UILabel* cellLabel = (UILabel*)[cell viewWithTag:kCellTitleTag];

if ([word compare: cellLabel.text]==NSOrderedSame) {
    //correct
    cellLabel.textColor = [UIColor greenColor];
} else {
    //incorrect
    cellLabel.textColor = [UIColor redColor];
}

[self performSelector:@selector(showDefinition:)
withObject:cellLabel.text afterDelay:1.5];
}
```

Let's go over this bit by bit:

- curWord is the index of the current word in self.objects so we fetch the current word in the NSString instance word.
- We fetch the tapped table cell and de-select it.
- cellLabel is our custom label we've added to the cell - we get it by the tag we set up in Interface Builder earlier.
- If word equals cellLabel.text we set the label's color to green, otherwise to red.
- We delay 1.5 seconds so the player notices the color change, and then we 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 presentModalViewController:dictionaryView animated:YES];

    //also move to next word
    if (curWord+1 < [self.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 `presentModalViewController:animated:` 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 on behind the scenes; keeping that in mind you might want to have a `UIActivityIndicator` or any other HUD showing up as it may actually take a 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 `self.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 a try!



It looks like the game works, but does it really? The word doesn't really change after you close the dictionary view ...



We still don't change the table contents anywhere in the code, so let's do that. We'll 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 we've already changed the `curWord` index when presenting the dictionary, 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 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



96

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, we will be covering two major new features added to the AddressBook framework in iOS 5.

The first new feature we will cover 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 we will cover 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 we're 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 created an online directory and has given you credentials to access it (based on the courses you attend). So we're going to develop an app that connects to the service, gives you a list of names, and allows 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](#): Returns a plist file with the peoples' contact data as a property list.
2. [vCard data](#): You supply a list of IDs and it returns a 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.

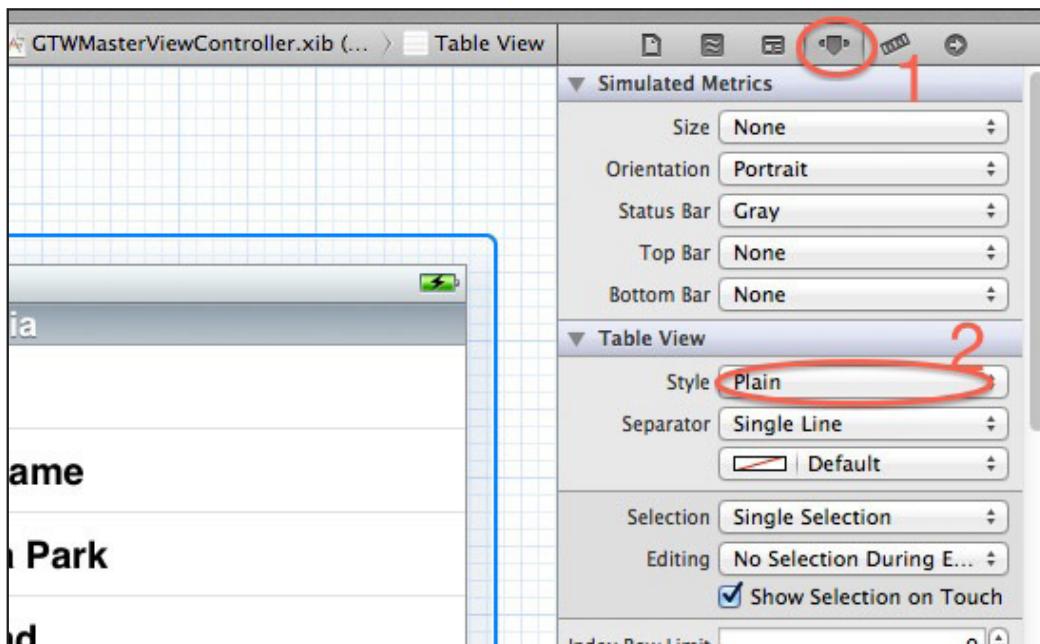
Enter **SocialAgent** for the product name, enter **SA** for the class prefix, select iPhone for the Device Family, and make sure that "Use automatic reference counting" is checked (leave the other checkboxes unchecked). Click Next and save the project by clicking Create.

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.

We'll use the table view controller that Xcode created for us, but we 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
- SADetailViewController.xib

Open **SAMaterVeiwcontroller.xib** 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).



Setup is done - we can move on to coding! Good job!

Preparing the Table View

The next step is to prepare our table view to be able to display the contact information we're going to pull from the web directory.

Start by opening **SAMasterViewController.h**, and replace the contents of the file with the following:

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

@interface SAMasterViewController : UITableViewController
@end
```

That's all we need in the interface file. I love iOS 5.0 interface files, how about you?

In fact we just removed the obsolete references to the details view controller. Next open up **SAMasterViewController.m** and replace the contents of the file with the following:

```
#import "SAMasterViewController.h"

#import <AddressBook/AddressBook.h> //1
#import "SAMasterViewController.h"

#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
```



We start very humbly, but we're 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. We use `dispatch_get_global_queue()` from Grand Central Dispatch here to get a background processing queue. We're going to use that queue so we 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 the data source for our table view.
5. We'll use `importContacts` later on - stay tuned!

We still have a few more things to get out of the way, then we 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 our table view only has 1 section (`numberOfSectionsInTableView:`) and it will have as many rows as there are objects 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 us the following fields:

- name (first name)
- family name (last name)
- telephone number
- Facebook
- Skype



So, we're going to show the name and the family name in each cell in the table. Let's do that by adding a `tableView:cellForRowAtIndexPath:` method to the class:

```
- (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];
    }

    NSDictionary* contact = [contacts objectAtIndex: indexPath.row];
    cell.textLabel.text = [NSString stringWithFormat:@"%@ %@",
                           [contact objectForKey:@"name"],
                           [contact objectForKey:@"family"]];

    return cell;
}
```

This is pretty standard code. It dequeues a new cell and creates one if there's no cell to dequeue. Then it gets an `NSDictionary` from our `contacts` array for the current cell, and sets the cell's text to be the name and family name.

w00t, that's all the setup code we 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

Our plan for the app is to read the contacts from the server as soon as the app is started, load the list of people into the contacts array, and reload the table view.

We'll do that in `initWithNibName:bundle:`, so go ahead and add this code inside **SAMasterViewController.m**:

```
- (id)initWithNibName:(NSString *)nibNameOrNil
  bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil
      bundle:nibBundleOrNil];
    if (self) {
        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
    }
    return self;
}
```

1. We 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. Why do we want that in this app? Well, we'll allow the user to select all contacts they want imported from the server, then tap the import button to bulk import them in one shot!
3. We add a navigation button that calls the `importContacts` method (we wrote a placeholder for this earlier).

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 we'd 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 GCD 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 we call `dispatch_async` and 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 the contents (expecting a plist file format) and then builds an `NSArray` instance from the fetched data.

So in just one line of Objective-C we 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 we pass `dispatch_get_main_queue()` as the first parameter - this will make sure the block we supply to it will be executed on the main thread where we can update the application's UI. And we do just that - we simply call `[self.tableView reloadData];` and the table view reads the list of people from our contacts ivar.

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, we'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.

We'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 he wants to import, and tap the "Import selected" button, so everything is done except the actual import.

This is where we're 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:

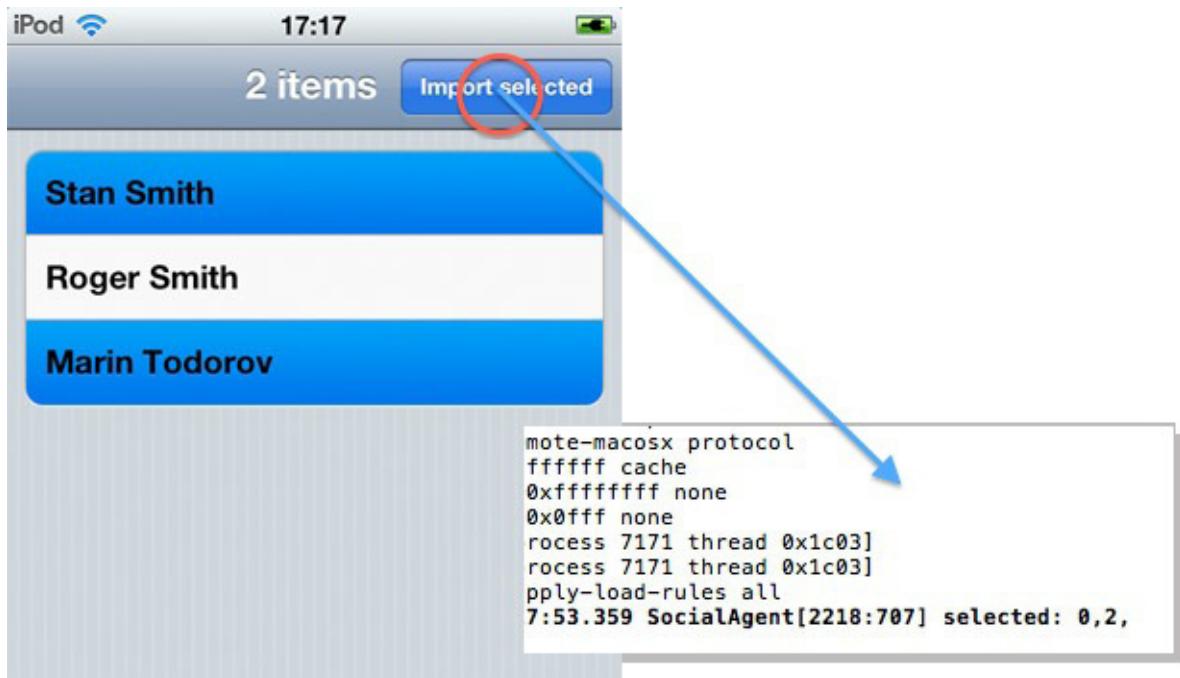
```
NSArray* selectedCells = [self.tableView  
    indexPathsForSelectedRows]; //1  
if (!selectedCells) return; //2  
  
NSMutableString* ids = [NSMutableString stringWithString:@""];  
for (NSIndexPath* path in selectedCells) {  
    [ids appendFormat:@"%@,", path.row]; //3  
}  
  
 NSLog(@"%@", selected); //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 we return in that case.
3. Next in a simple loop we 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 we're going to:

1. Invoke the web service again and get the vCard data to import for the selected people.
2. Cycle through the people we've just imported and get their Facebook user-names, so the app can display them to the user so he can add them on Facebook.
3. Save the new records to the address book and show an alert that we're all done.

We're 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 we 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.
2. `dataWithContentsOfURL:` reads the given URL and returns the response as `NSData`.

Then we just have comment placeholders for everything I mentioned we're going do next.

Parsing the vCard Data

Replace the "parse vCard data" comment with the following code:

```
ABAddressBookRef addressBook = ABAddressBookCreate(); //1
ABRecordRef record = ABPersonCreate(); //2
NSArray* importedPeople = (__bridge_transfer NSArray*)
    ABPersonCreatePeopleInSourceWithVCardRepresentation(
        record, (__bridge CFDataRef)responseData); //3
CFBridgingRelease(record); //4
```

- 1.`ABAddressBookCreate()` gives us an instance of the Address Book. It returns an `ABAddressBookRef`, which comes retained through the bridge.
2. `ABPersonCreate()` creates an empty address book record. We 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 Address Book record we created. The second parameter is the `NSData` of the vCard text. Actually, it needs a `CFDataRef` - so we have to cast it as such through the bridge.
- 4.`CFBridgingRelease()` releases an object, which has been retained through the bridge.

Do you see on line 2 we used `(__bridge_transfer NSArray*)`? This is because the function returns a retained result, so while we're casting it to `NSArray*` we also "transfer" its retain count to ARC - so it'll take care to automatically release it later on for us.

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 we have to release them manually by calling `CFBridgingRelease()`.



We'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 on how to deal with all the new bridge memory management with ARC: If you're 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, which is a pretty good clue on how to fix your code:

The screenshot shows a portion of Xcode's code editor with line numbers 58 through 67. A blue callout box highlights a warning at line 59: "Potential leak of an object allocated on line 59 and stored into 'addressBook'". An arrow points from this callout to the variable 'addressBook'. Other annotations include "Unused variable 'addressBook'" and "Unused variable 'ImportedPeople'". The code itself is in Objective-C and involves creating an Address Book, a record, and an array of people, then releasing the record.

```

58
59     ABAddressBookRef addressBook = ABAddressBookCreate(); // Unused variable 'addressBook'
60     ABRecordRef record = ABPersonCreate();
61     NSArray* importedPeople = (__bridge_transfer NSArray*) ABPersonCreatePeopleInSourceWithVCardReprese
62     CFBridgingRelease(record); // Unused variable 'ImportedPeople'
63
64     // Potential leak of an object allocated on line 59 and stored into 'addressBook'
65
66     // Loop over people and get their Facebook
67

```

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:

```

__block NSMutableString* message =
NSMutableString stringWithString:
    @"Contacts are imported to your Address Book. Also you can add
        them in Facebook: "];

NSString* serviceKey = (NSString*)kABPersonSocialProfileServiceKey;
NSString* facebookValue = (NSString*)
    KABPersonSocialProfileServiceFacebook;
NSString* usernameKey = (NSString*)kABPersonSocialProfileUsernameKey;

```

First we define a message we'll display in the alert later on. It's a mutable string, so we can append to it the Facebook usernames of the imported people. We mark it with the `block` keyword, because we will be using (and modifying it) from within a block later.



Next we copy the values of a few constants into `NSStrings`. We're doing that because we 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 we're going to do you'll need an insight into an AddressBook record. Let's consider 1 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 we 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 objectAtIndex: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 objectForKey:kServiceKey];
        //5
        if ([facebookValue compare: curServiceValue]
            == NSOrderedSame) {
            //6
            [message appendFormat: @"%@, ",
                [profile objectForKey:kUsernameKey]];
        }
    }

    //7
    CFBridgingRelease(profilesRef);
}

```

Here we loop over the list of imported people and for each record we do the following:

1. Store the current person from the list into the personRef variable, and we 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 we end up with a nice NSArray of "profiles" to work with.
3. We loop over "profiles" and we get an NSDictionary with the data of each profile.
4. We store the type of social service into curServiceValue by accessing the key serviceKey (we defined this constant earlier).
5. We compare the Apple defined constant for Facebook with the current service type.
6. If a Facebook profile was found, we add the username to our message string.



7. Finally we call `CFBridgingRelease()` for `profilesRef` so we 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 (we 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 we did just above there) - the info is actually stored in a temporary copy of the address book. So, since at this point we'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 we release the reference we got from the bridge.



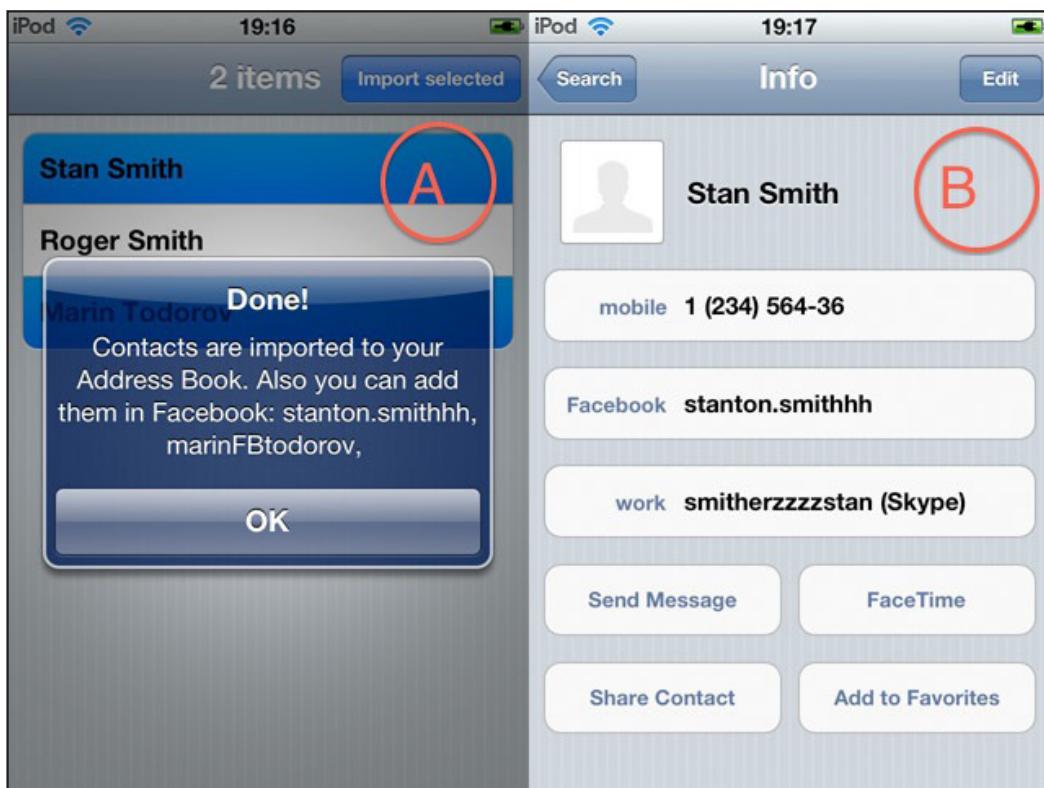
Showing the Done Alert

After all the work is done (in a background queue - whoa this feels like ages ago) we want to show the user the message and let him know the contacts are imported. In order to work with the application's UI we'll need to run some code on 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 we did 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 all the 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.
- Further develop a real web service, possibly one accessible via authentication.
- Produce cool mash-ups like "The Flickr photo stream of your contacts".



97 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 Core Location framework with the new class `CLGeocoder`, and placemarks are now handled by the new class `CLPlacemark`. It's 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 we're going to build an app called BeerAdvisor, that will start by fetching the user's location and show him/her their current address and location on the map. Once the app detects 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

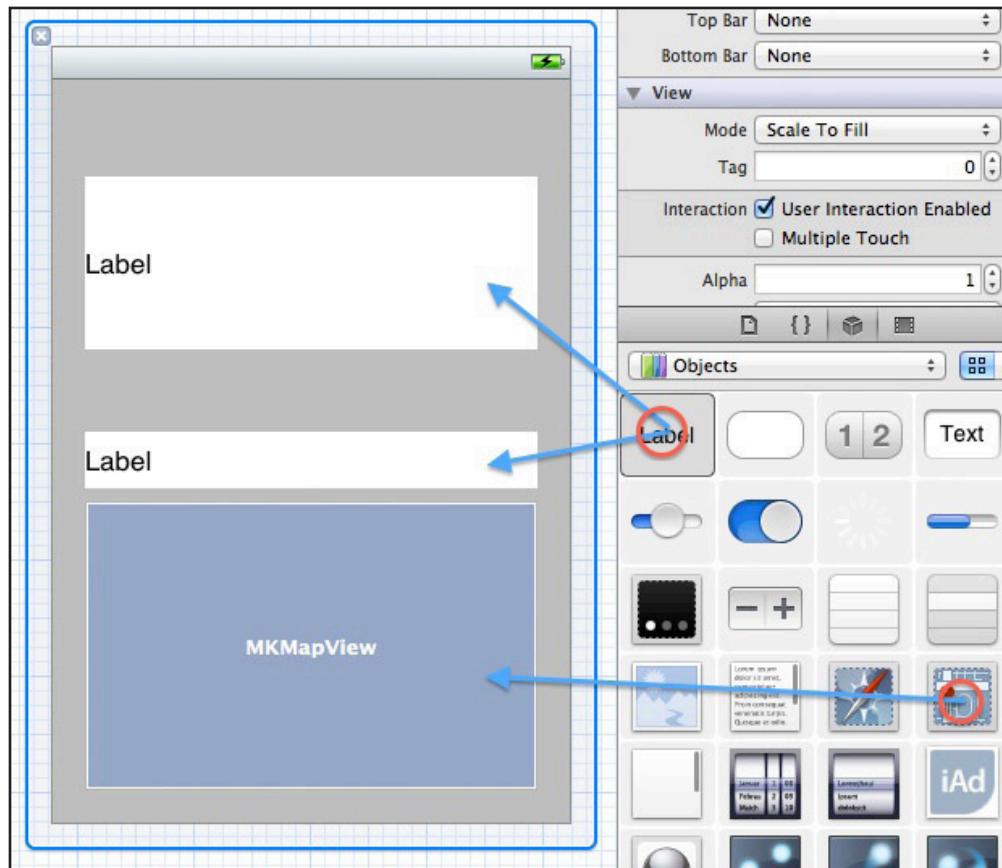
Start Xcode and from the main menu choose **File\New\New Project**. Select the **iOS\Application\Single View Application** template, and click Next. Enter **Beer-Advisor** for the product name, enter **BA** for the class prefix, select iPhone for the Device Family, and make sure that "Use automatic reference counting" is 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 **CoreLocation.framework** to add Core Location capabilities to the project. We'll also be using a map to show the user's location, so click the plus button again and double click **MapKit.framework**.



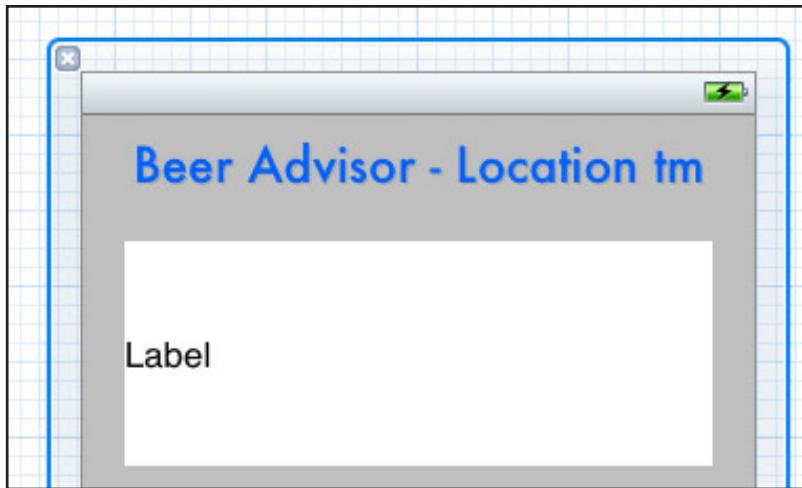
Open up **BAViewController.xib**. We're going to set up the interface to contain a map, a label to show the current coordinates, and a label to show the current address.

Drag 2 labels and 1 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 the background for both labels to white. 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.

One final touch - let's add a heading to the application (beautify as you find fit):



We will need a few outlets to the UI elements we just created. Open **BAViewController.h** and replace the contents of the file with the following:

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

@interface BAViewController : UIViewController {
    IBOutlet MKMapView* map;
    IBOutlet UILabel* locationLabel;
    IBOutlet UILabel* addressLabel;
}

@end
```

Next, open **BAViewController.xib** and make the necessary connections:

- Control-drag from **File's Owner** to the **top label**, and choose **addressLabel** from the popup.
- Control-drag from **File's Owner** to the **bottom label**, and choose **locationLabel** from the popup.
- Control-drag from **File's Owner** to the **map view**, and choose **map** from the popup.

OK - we're all connected! Let's move on to some coding.

Reverse Geocoding With Core Location

First we'd like the application to fetch the user's current location and show him where he is - in a human readable form.



Tracking user's location is done the same way as pre-iOS 5.0. Open **BAViewController.h** and add an instance variable to keep track of the location manager, like so:

```
CLLocationManager* manager;
```

The BAViewController will also be core location manager's delegate for reporting location changes. Thus BAViewController will have to conform to the CLLocationManagerDelegate protocol, so mark it as so:

```
@interface BAViewController : UIViewController  
    <CLLocationManagerDelegate> {
```

We're done with the interface file for the moment - let's move on to the implementation. Open **BAViewController.m** and import the Core Location framework header at the top of the file:

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

With this done - remove all the boilerplate method declarations, we're going to add just what we need as we go.

When the app starts we would like to have changes in location reported to us. So let's hook up to viewDidLoad and fire up the location manager in there. Add this viewDidLoad inside the implementation body:

```
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
  
    //start updating the location  
    manager = [[CLLocationManager alloc] init];  
    manager.delegate = self;  
    [manager startUpdatingLocation];  
}
```

We create a new instance of CLLocationManager and store it in the manager instance variable. We set the delegate to self (the view controller) and finally call startUpdatingLocation so the location manager starts reporting location changes to its delegate.

Now to be good iOS developers, let's 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 we'll need to implement that in our 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];  
    }  
}
```

Notice the method takes `newLocation` and `oldLocation` parameters? This fact helps us track how much the user has moved, and in our case - if the latitude didn't change we 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.

First let's add a declaration of the method, just above the `@implementation`:

```
@interface BAVViewController()  
-(void)revGeocode:(CLLocation*)c;  
@end
```

Note we are adding this method inside a category to make it private. Next add the method itself:

```
-(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 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 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.

We get the 1st result returned from the server and store it in revMark. The CLPlacemark class is new to iOS 5 and provides extended information about a location. Let's have a quick look at the properties we 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 (how cool is that?)



For our project we're going to use only the address dictionary, but getting all sorts of info out of CLPlaceMark is just as easy.

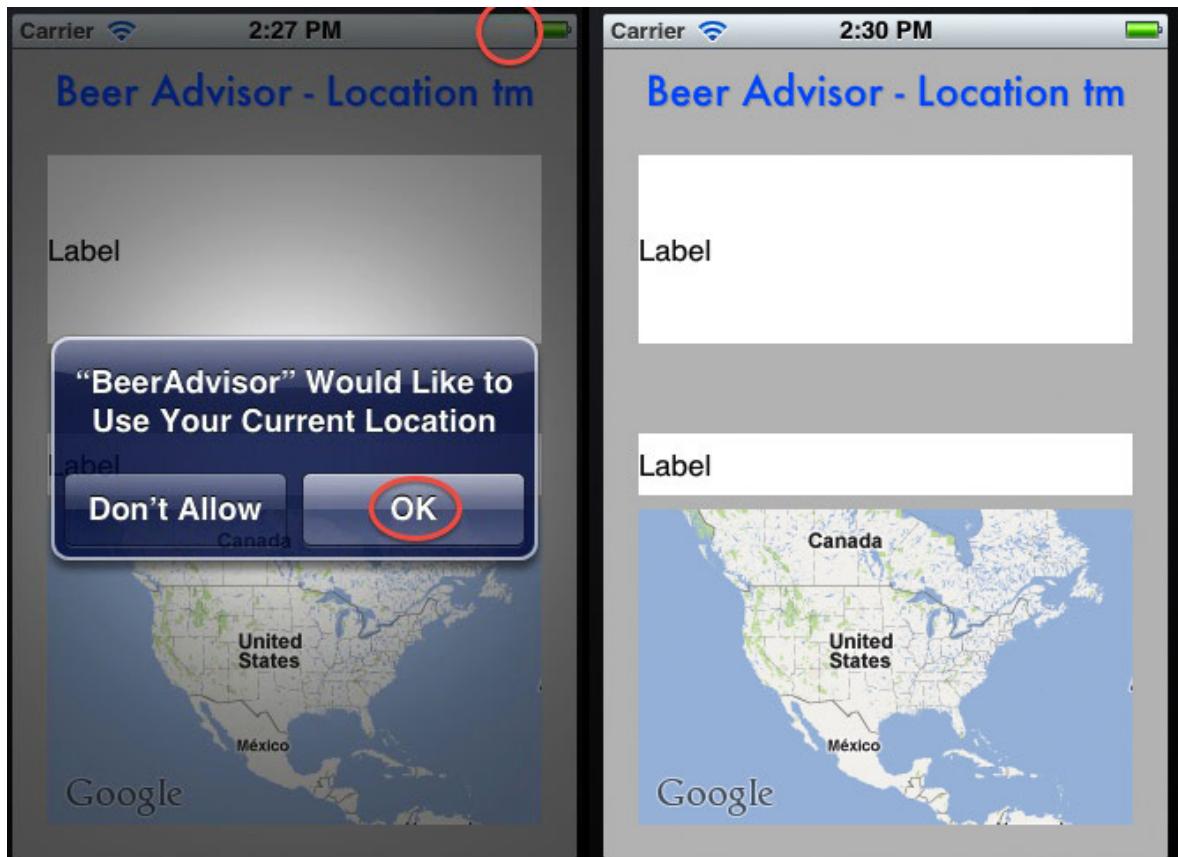
So, let's get the address and show it in our label. Find the "turn placemark to address text" comment and replace it with the following code:

```
NSArray* addressLines =
[revMark.addressDictionary objectForKey:@"FormattedAddressLines"];
NSString* revAddress =
[addressLines componentsJoinedByString: @"\n"];
addressLabel.text = [NSString stringWithFormat:
@"Reverse geocoded address: \n%@", 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. We 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 we've got from the CLPlaceMark. Let's test, hit Run in Xcode:



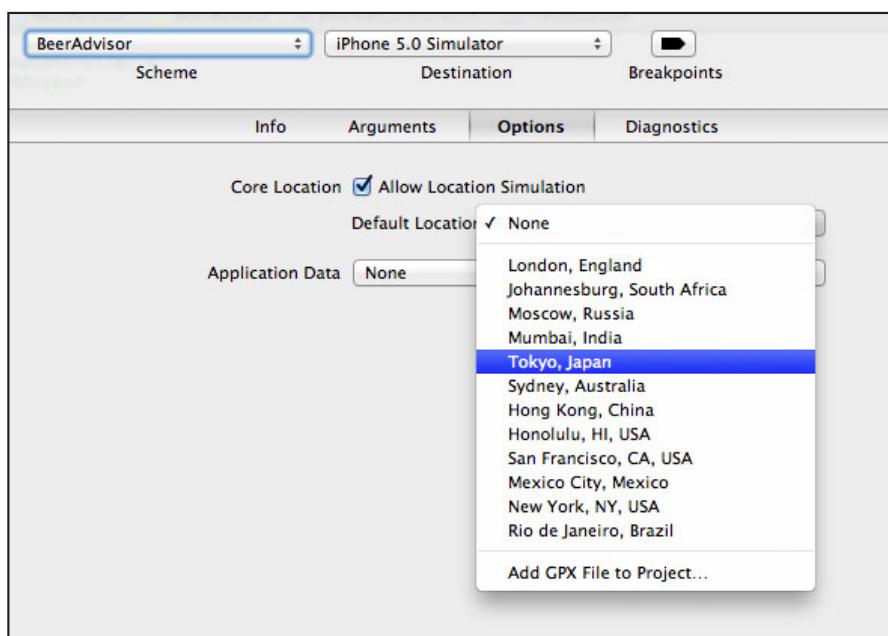
As usual, since the iOS doesn't have permission for the app to use location data, the Simulator asks for permission. Once you tap "OK" the app goes on running. However - that's not exactly what we expected - is it? Well the Simulator does not simulate location - this is what is wrong. That's a bit disappointing at first, but we can easily fix that with the new tools Xcode 4.2 provides us with.

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 your favorite place out of the list of locations pre-defined by Apple. I'll choose **Tokyo, Japan**.



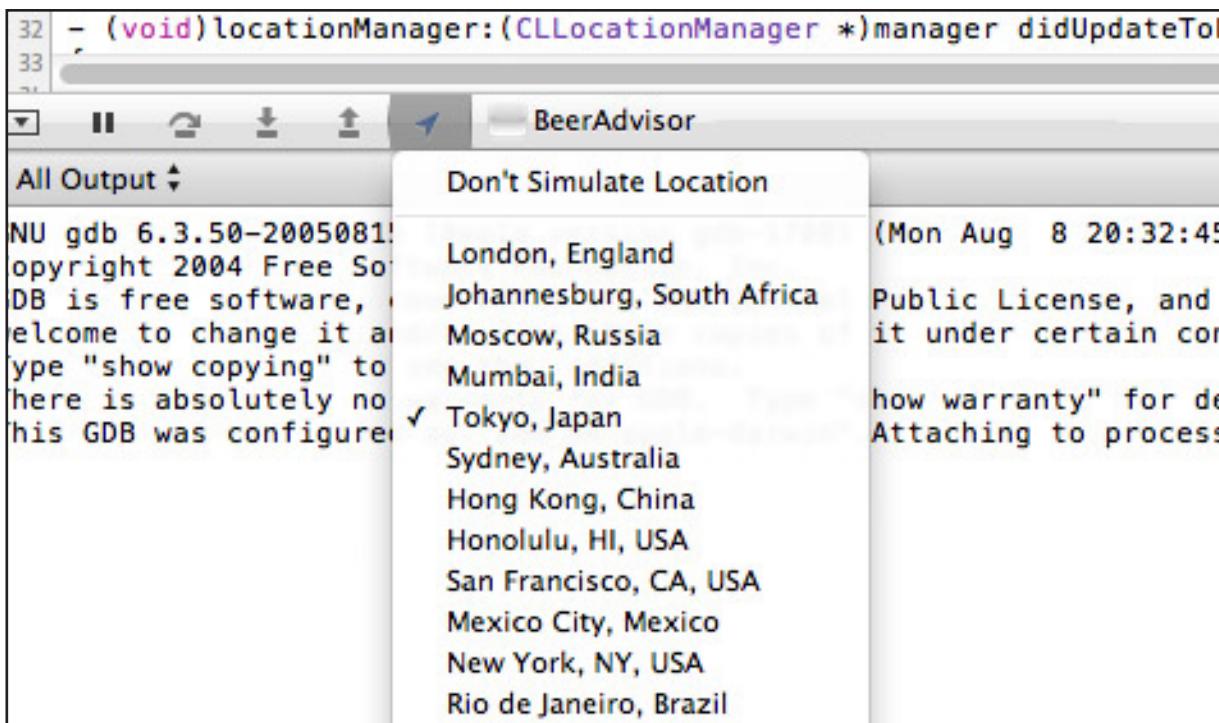
Now click "OK" and hit the Run button again:





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!

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 locations pops up - so you can actually change the currently simulated location at runtime. Try that - choose several different locations and watch the application react and show the addresses.



The screenshot shows the Xcode debugger interface with the "All Output" tab selected. A list of locations is displayed under the heading "Don't Simulate Location". The locations listed are:

- London, England
- Johannesburg, South Africa
- Moscow, Russia
- Mumbai, India
- Tokyo, Japan
- Sydney, Australia
- Hong Kong, China
- Honolulu, HI, USA
- San Francisco, CA, USA
- Mexico City, Mexico
- New York, NY, USA
- Rio de Janeiro, Brazil

At the bottom of the list, there is a checked item: Tokyo, Japan.

Forward Geocoding

Next we're going to take the address we've got, 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 we get the address from the server, we're immediately going to make another call to convert it to coordinates.

Inside the interface declaration (in that same file you have already open) add:

```
-(void)geocode:(NSString*)address;
```

And here's the initial method body to add inside the implementation of the class:

```
-(void)geocode:(NSString*)address
{
    //1
    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 objectAtIndex: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 we 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 place-mark objects, so handling the result goes the same way as what we 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 place-marks as result (that's why it's a good idea to check the error parameter).
4. We get the coordinates from the first `CLPlacemark`.
5. Finally we show the coordinates inside the location label in the UI.





OK - we'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 gonna discuss it in detail. First of all we'll need a custom location annotation class, 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, readonly) CLLocationCoordinate2D coordinate;
-(id)initWithCoordinate:(CLLocationCoordinate2D)c;

@end
```

We define a class which conforms to the MKAnnotation protocol, so we can use it to add placemarks to a MapKit map view. The only required property is the coordinate

property. I also would like to be able to initialize the annotation in one shot, so I chose to have a custom initializer `initWithCoordinate:`.

Next open **BAAnnotation.m** and replace its contents with the following:

```
#import "BAAnnotation.h"
@implementation BAAnnotation

@synthesize coordinate;

-(id)initWithCoordinate:(CLLocationCoordinate2D)c
{
    if (self = [super init]) {
        coordinate = c;
    }
    return self;
}

@end
```

We synthesize the required coordinate property and implement the custom initializer. Now we're 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:

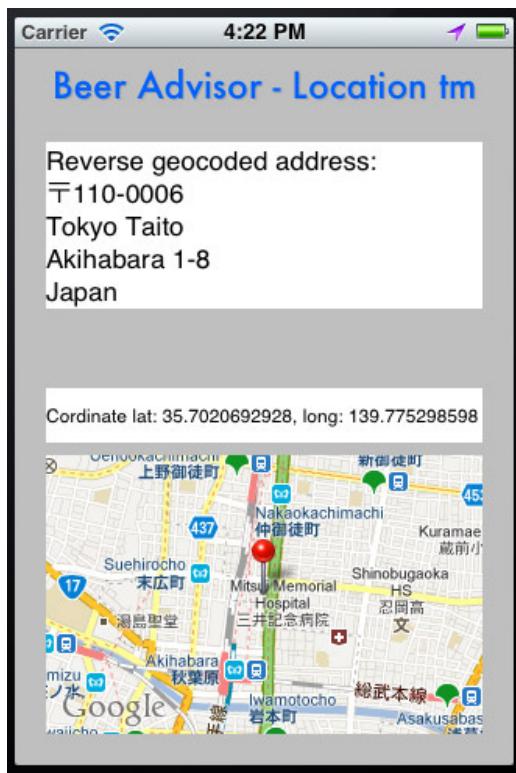
```
//1
CLLocationCoordinate2D coordinate;
coordinate.latitude = lat;
coordinate.longitude = lng;
//2
[map addAnnotation:(id)[[BAAnnotation alloc]
    initWithCoordinate:coordinate] ];
//3
MKCoordinateRegion viewRegion =
    MKCoordinateRegionMakeWithDistance(coordinate, 1000, 1000);
MKCoordinateRegion adjustedRegion = [map regionThatFits:viewRegion];
[map setRegion:adjustedRegion animated:YES];
```

1. First we build a `CLLocationCoordinate2D` with the latitude and longitude we just got back from the server.
2. In one shot we create a new `BAAnnotation` and add it to the map.



3. We build a region around the coordinate and set it to the map.

Now you can hit the Run button again and see the result:



At this point we have the app forward and reverse geocode with Core Location! To celebrate, let's make an alert popup when we go into a good area for drinking beer.

Monitoring for a Given Area

Monitoring for a given region is supported prior to iOS 5.0, but now there's a new API called `startMonitoringForRegion:` which doesn't take an accuracy parameter as the one available in 4.x, so we're going to use the new one.

The "Beer Advisor" app will only have one pre-defined 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. We build a `CLLocationCoordinate2D` with the coordinate of the area's center - more or less (I just sampled it by hand in google maps).
2. We create `kreuzberg CLRegion` which has the chosen center and a radius of 1000 meters (about 0.6 miles).
3. `startMonitoringForRegion:` will tell the location manager to let us 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 which region the user entered inside the alert view - 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.

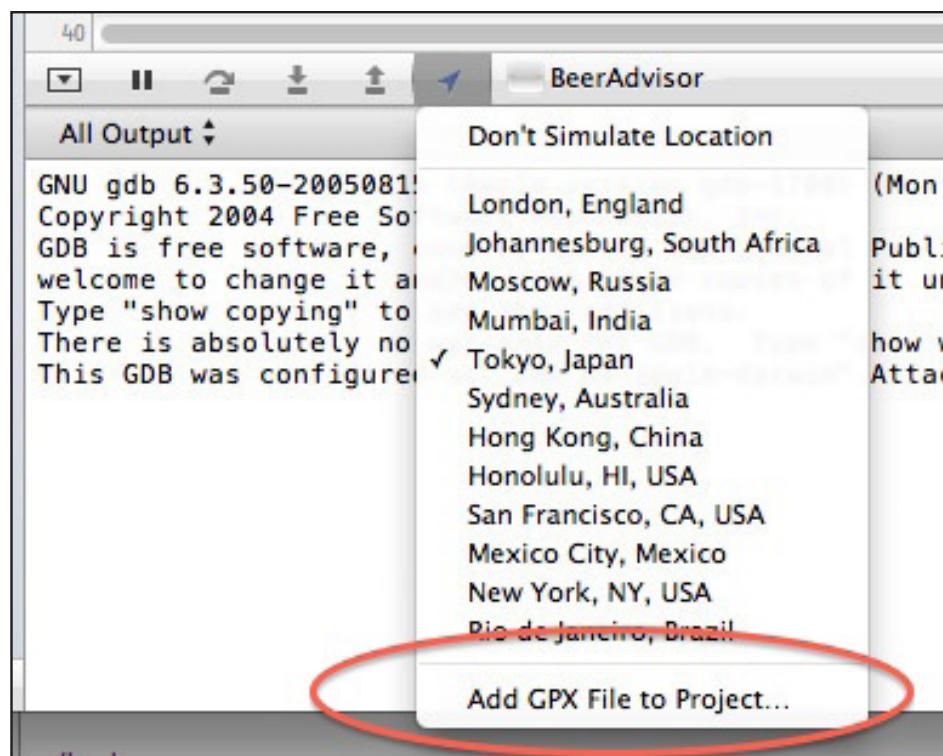
At this point a question comes to mind. Apple doesn't seem to be big fans of Kreuzberg - they didn't find it necessary to include it in their predefined locations list, so we have a problem how to test this functionality without flying over to Berlin!

Location in the Simulator - Part 2

Luckily Apple allows to add 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 there's an option at the bottom called "Add GPX file to project...?"?

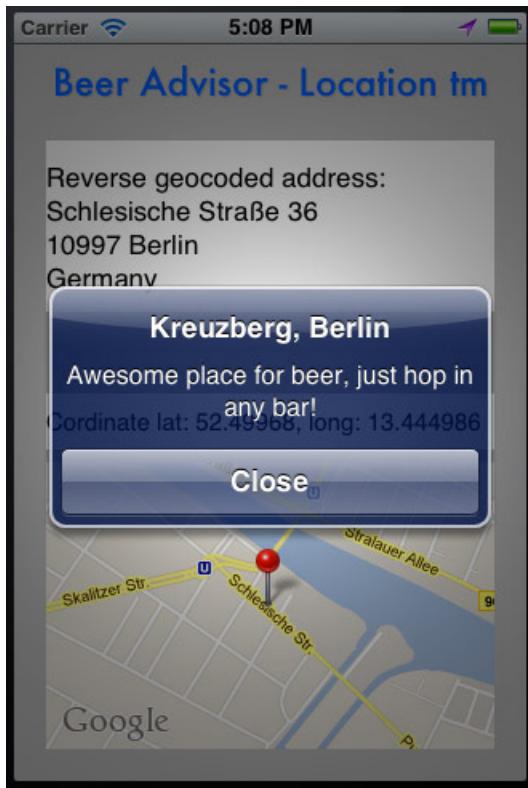




GPX files are plain text files written in markup which contain GPS data. If you're interested in the GPX file format have a look here : http://en.wikipedia.org/wiki/GPS_eXchange_Format

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 which 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 pub 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.

98 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 though also few other new APIs and it will be really unfair if they did go unnoticed. In this tutorial I'm gonna 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 which 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" is 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 **GameKit.framework** to add Game Center capabilities to the project.

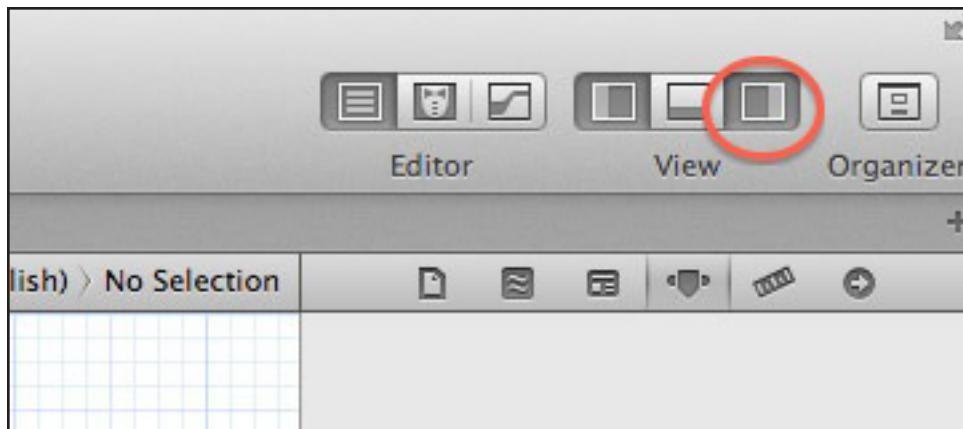
There's one final step of the setup: Setting up Game Center for your demo project, unfortunately this is out of the scope of this tutorial, but here's what you need to do in short:

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 **GCDViewController.xib** and bring up the Utilities bar on the right:

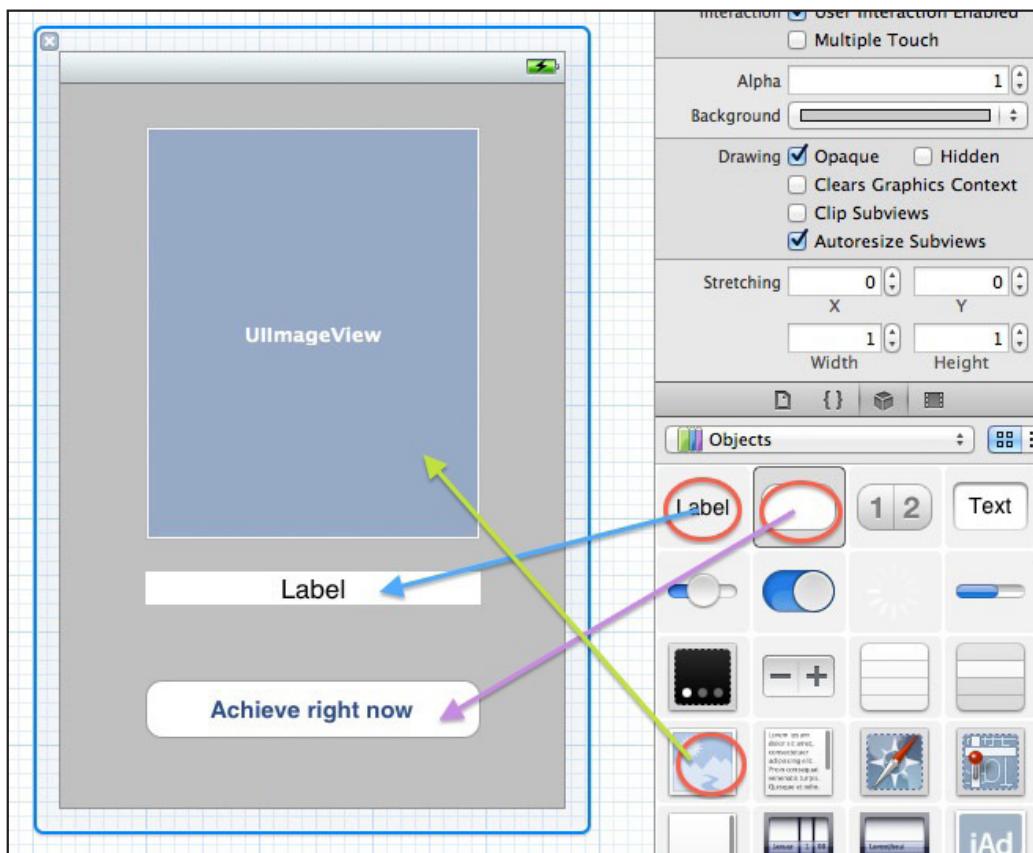




For the complete demo project we're going to need:

1. A UIImageView to show the current player's photo.
2. A UILabel to show the logged player's handle.
3. A UIButton which 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.h** and replace the boilerplate with this code:

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

@interface GCDViewController : UIViewController
{
    IBOutlet UIImageView* photoView;
    IBOutlet UILabel* nameLabel;
}

-(IBAction)achieveRightNow;

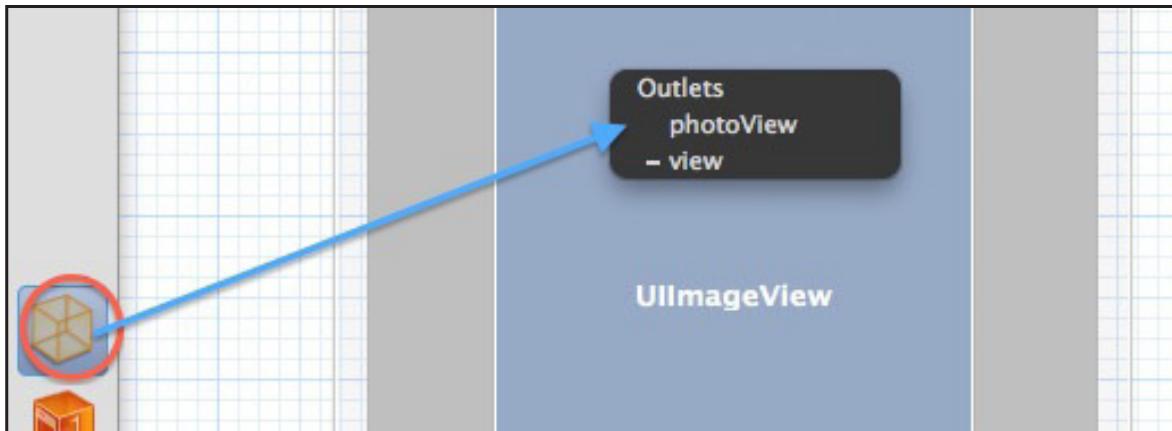
@end
```

We have 2 instance variables which are outlets to our UI elements. `photoView` will help us 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.

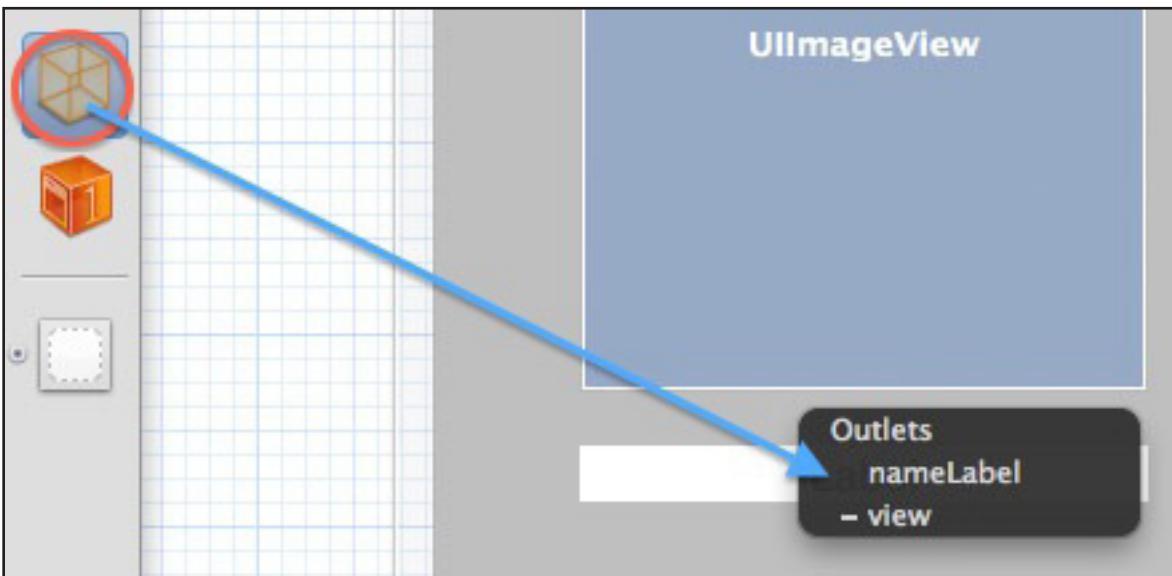
Note: One of the new features of the iOS 5 compatible Xcode is that you can declare `IBOutlets` in either the interface file, a class extension in the implementation file (a class category) or the implementation itself. And that's really cool, but while I was preparing the tutorials for this book I actually had problems with all but the former. Very rarely, just sometimes an `IBOutlet` would appear connected, but it actually won't work if it's declared anywhere else but the interface file. So for the moment being I'm still declaring the `IBOutlets` in the interface of my view controllers, just to be on the safe side.

Let's connect the UI elements and go on further with coding. Switch back to **GCD-ViewController.xib** and control-drag from **File's owner** to the **UIImageView** and from the popup select **photoView**:





Similarly, control-drag from **File's owner** to the **UILabel** and from the popup select **nameLabel**:



Last but not least, control-drag from the **UIButton** to **File's Owner**, and choose **achieveRightNow**:



Good! The interface is all connected, and we can start coding.

Loading the Current Player's Photo

"Look ma' they got me soul on the Game Center!" I hope this pun works, as indeed you can have now your picture on the Game Center and all games can access it programmatically via the GameKit framework.

We want to make our app log the player into Game Center, fetch their photo and show it. We'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]
authenticateWithCompletionHandler:^(NSError *error) { //2

    dispatch_async(dispatch_get_main_queue(), ^(void) { //3

        nameLabel.text = [GKLocalPlayer localPlayer].alias; //4
        //show the user photo
        //show logged notification

    });
}
}];
```

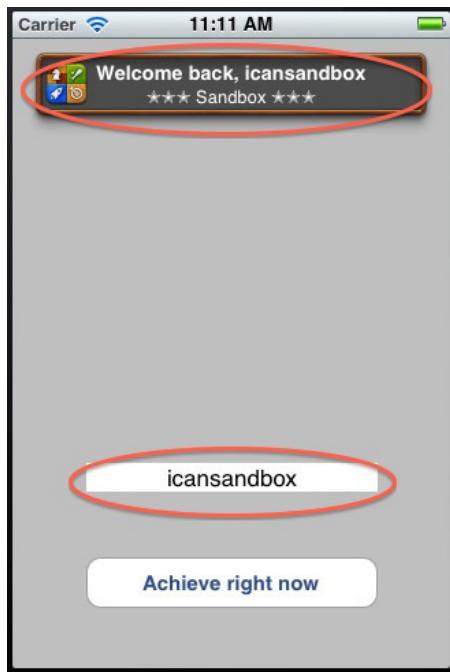
Here's what this code does, section by section:

1. We first check if the local player is already authenticated with Game Center.
2. If not - we call authenticateWithCompletionHandler: and we 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(),...) since you can only modify the UI from the main queue.
4. Finally [GKLocalPlayer localPlayer].alias is the logged player handle - we show it in our UI label

There's also couple of comments which we'll replace with code a bit later.

If you hit Run and test the app right now, you'll see the following result:





Everything's working - a notification banner shows up from Game Center and the label shows the currently logged player. Note we're running against the Game Center sandbox servers, not the real servers, because our app is still in development.

If this doesn't work for you, and instead you see something like this:



This of course means your bundle id is not recognized by Game Center - go back to the "Project setup" section in this tutorial where I explain what you need to activate the demo project with Game Center. If you don't have a developer account - you'll still be able to follow through most of the tutorial, so just close the alert.

Next we're going to fetch the picture of the currently logged player from Apple's servers and show it on the screen. Just above the @implementation we'll add a class extension in order to define a new private method:

```
@interface GCDViewController()  
-(void)loadPlayerPhoto;  
@end
```

Next 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";  
            }  
  
        });  
    }];  
}
```

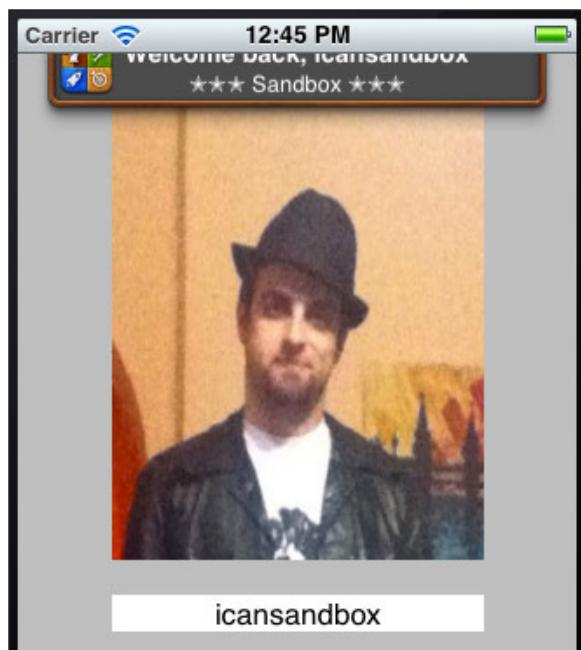
So to get the photo for the local player in iOS 5, all you have to do is call `loadPhotoForSize:withCompletionHandler:` like you see here. Alternatively if you're reading somebody else's photo you'll have to load a `GKPlayer` instance for him/her and then load the photo.

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`.

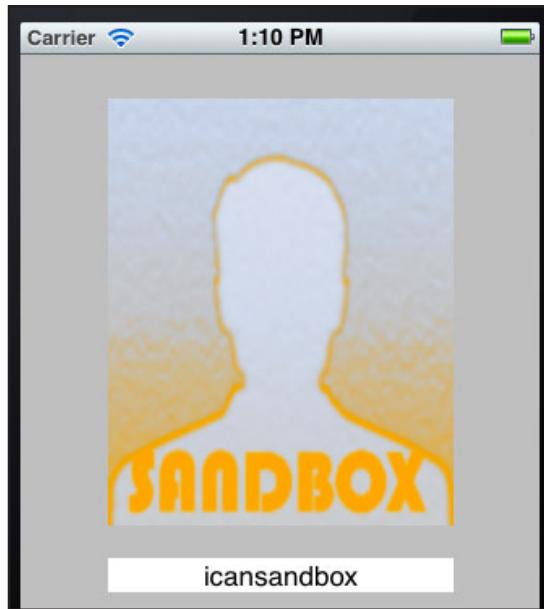


When there's a response from the server the block you pass is invoked with the following parameters: `UIImage *photo` and `NSError *error`. `photo` is the player's picture and `error` denotes whether there was an error fetching the photo. **Important:** the server will return `error` if the player didn't 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: Right now the sandboxed Game center environment will delete your photos about 2-3 minutes after you upload them and replace them with a standard Game center picture. I couldn't find indication whether this behavior is here to stay or it's some temporary thing. But just so you know - if you run your test app in like 5- 10 minutes after you uploaded your photo through the Game Center app, you could see something like this:



Showing A Notification Banner

Showing a message in the standard UI of Game Center is utterly easy.

Let's wrap up a simple method for our class. Make the following changes to **GCDViewController.m**:

```
//add just below the line "@interface GKViewController()"  
-(void)showMessage:(NSString*)msg; //1  
  
//add inside the implementation  
-(void)showMessage:(NSString*)msg  
{ //2  
    [GKNotificationBanner showBannerWithTitle:@"GameKit message"  
                                message:msg  
                           completionHandler:^{}];  
}
```

First we declare the method in the class extension.

Second, 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 bold), 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 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 we'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 our view controller's implementation:

```
-(IBAction)achieveRightNow
{
    GKAchievement* achievement= [[GKAchievement alloc]
        initWithIdentifier:
            @"writinggamecenterapichapter"];
    achievement.percentComplete= 100;
```



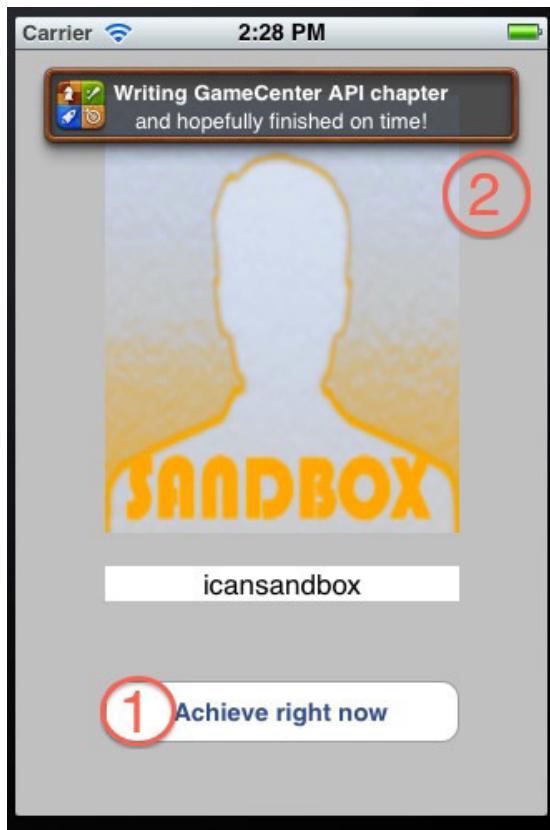
```
achievement.showsCompletionBanner = YES;
[achievement reportAchievementWithCompletionHandler:^(NSError *error)
{}];

}
```

We've already connected earlier the button in the UI to this IBAction, so let's just quickly have a look at the code. We create a new GKAchievement, passing in the identifier we set in Game Center. You should change this to whatever identifier you used.

We then set it 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!



99 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 roll-back 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 we're 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 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" is checked (leave the other checkboxes unchecked). Click Next and save the project by clicking Create.

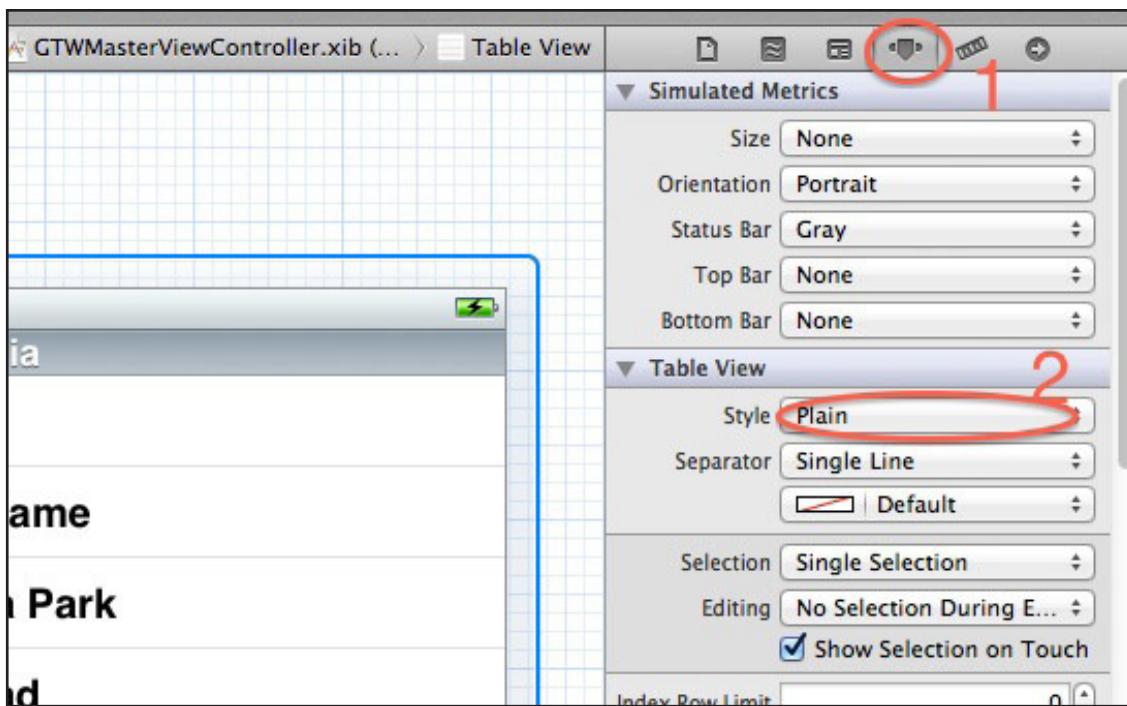
Select the project and select the GoodTimesTVGuide target. Open the **Build Phases** tab and open the **Link Binary With Libraries** fold. Click the plus button and double click **EventKit.framework** to add Calendar access to the project. In our project we're going to also use the view controllers Apple provides for editing events and choosing from the existing list of calendars so click again the plus button and also add the **EventKitUI.framework**.

We'll use the table view controller that Xcode created for us, but we 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
- GTDetailViewController.xib

Now open **GTMasterViewController.xib**, 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).





OK - a little bit of cleanup in the code and we're going to be ready to start. We need to remove the references to the details view controller that Xcode added for us. Open **GTMasterViewController.h** and replace everything with this code:

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

@interface GTMasterViewController : UITableViewController
{
    NSArray* shows;
}
@end
```

Since we're going to work with EventKit and EventKitUI we import EventKitUI.h in the interface of the view controller - this way we'll have access to both frameworks in the header and implementation files.

Also we'll need one instance variable - the shows array. This will hold the list of TV shows our Good Times TV Guide will provide schedules for, and will be the data source for our table view.

Let's move on to the implementation file - we'll add the basics and kick it off onto the EventKit magic. Open **GTMasterViewController.m** and replace the boilerplate code with:

```
#import "GTMasterViewController.h"

@interface GTMasterViewController()
```



```
//private methods here
@end

@implementation GTMasterViewController

- (id)initWithNibName:(NSString *)NibNameOrNil
    bundle:(NSBundle *)nibBundleOrNilOrNil
{
    self = [super initWithNibName:nibNameOrNilOrNil
        bundle:nibBundleOrNilOrNil];
    if (self) {
        self.title = @"Good Times TV Guide";
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSString* listPath = [[NSBundle mainBundle].resourcePath
        stringByAppendingPathComponent:@"showList.plist"];
    shows = [NSArray arrayWithContentsOfFile: listPath];
}

@end
```

We do little bit of basic setup: first we declare an empty class extension `GTMasterViewController()` where we will be adding private method declarations as we develop the project. Then inside `initWithNibName:bundle:` we set the title for the navigation controller to be "Good Times TV Guide".

Finally in `viewDidLoad` we load the tv shows data into the `shows` array. The show data we load from a file called "showList.plist". 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
▼ Item 0	Diction...	(6 items)
title	String	How I met your father
url	String	http://www.cbs.com
startDate	String	09/29/2011 19:00 PST
endDate	String	09/29/2011 19:30 PST
dayOfTheWeek	Number	5
tip	String	Usually everyone is a good guy, besides anyone who dates Ted.
▼ Item 1	Diction...	(6 items)
title	String	Prison break-in
url	String	http://www.fox.com/
startDate	String	09/25/2011 14:00 PST
endDate	String	09/25/2011 15:00 PST
dayOfTheWeek	Number	2
tip	String	Jasons is secretly in love with his cell-mate, so he is gonna betray Paul.

For each show we have its 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 our 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 we 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 we show an example of fetching plist data from an online web service.

Now that we 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];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }

    // Configure the cell.
    NSDictionary* show = [shows objectAtIndex: indexPath.row];
    cell.textLabel.text = [show objectForKey:@"title"];

    return cell;
}
```

That's a bit more code - but it's all standard table view code really. In `numberOfSectionsInTableView:` we set 1 section for our table and in `tableView:numberOfRowsInSection:` we tell the table view that we'll have as many table rows as shows there are in the `shows` array.

`tableView:cellForRowAtIndexPath:` is pretty standard too - we dequeue a table cell and if there's not one to dequeue we create a new one. Finally we get the show for that particular cell from the `shows` array and we set the cell's text to the name of the show.

If you hit Run in Xcode right now you'll see that we are ready with the basic interface of the application:



When the user decides that he wants to import a given show schedule into his device's Calendar he'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: we will show an alert to the user when he taps a table row and ask him to which calendar he'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 when you tap a row you'll 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 we're 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 we have for the AppCalendar class. First we import the EventKit framework, and then we define kAppCalendarTitle - this will be the title of our custom calendar. We're defining 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.

We 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**: We 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
{
    //keep a static instance of eventStore
    if (!eStore) {
        eStore = [[EKEventStore alloc] init];
    }
    return eStore;
}

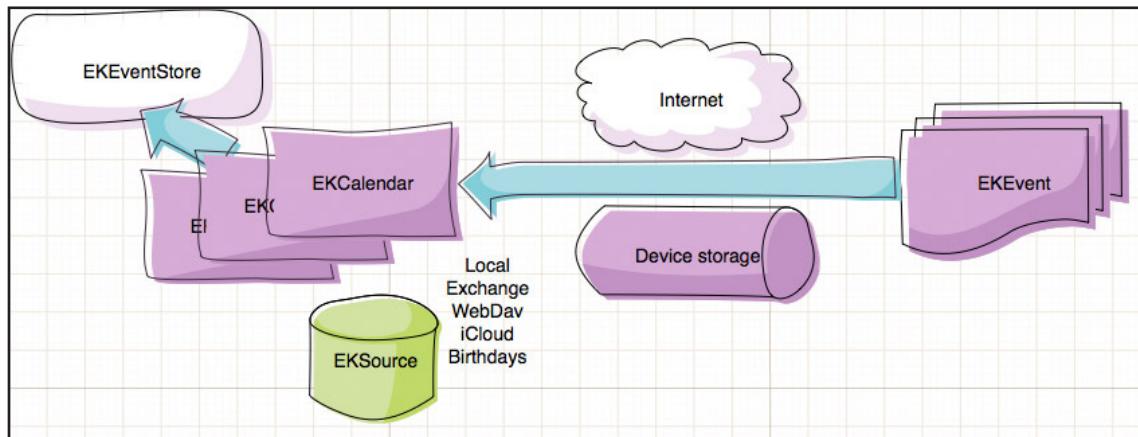
@end
```

It's a good start - we have a static object eStore which will hold the EKEventStore. Then inside the eventStore method, we check if eStore has been initialized. If it has, we just return the static object, otherwise we initialize and return the 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 we go on coding, let's take a look at the EventKit hierarchy of classes so you'll have a better understanding of what we're about to code:



If you look at the scheme above, EKEEventStore class is at the top of the hierarchy - EKEEventStore 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 EKEEventStore 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 calendarWithEventStore: 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 we get the `EKEventStore` instance from the "eventStore" method we did earlier. And then we do few more steps in order to create the calendar:

1. We iterate over the `EKEventStore`'s available sources and when we reach the source with `sourceType` of `EKSourceTypeLocal` we store it in `localSource`; this is the source type we're going to use for our app's calendar.
2. 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).
3. We create a new `EKCalendar` instance by simply calling `[EKCalendar calendarWithEventStore: store]`. Then we assign the calendar's title and the source that we just stored in `localSource`. The events inside the new calendar will all have background color defined by the color set to the property `CGColor`.
4. We 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.
5. Finally if the calendar is successfully created and saved we store its unique calendar identifier in the user preferences, so we can look it up by its id next time we 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. 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 our 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 stringForKey:@"appCalendar"];

    //3
    if (calendarId && (result =
        [store calendarWithIdentifier: calendarId]) ) {
        return result;
    }
}
```



```
//4 check for a calendar with the same name
for (EKCalendar* cal in store.calendars) {
    if ([cal.title compare: kAppCalendarTitle]==NSOrderedSame) {
        if (cal.Immutable == 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;
}
```

Explanation:

1. First we prepare the object to hold the method's result - result. Also we use the previously defined eventStore method to get hold of an EKEventStore instance.
2. We read the key appCalendar from the user preferences. If the application has previously created a calendar, we 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, we can find it by looping over the available calendars and comparing their titles to our desired calendar title. If we find a matching calendar title we also check if the calendar is writable by the current app. If it is, we return that calendar right away and save its unique identifier for future reference.
5. When no calendar is found, we call the previously defined method createAppCalendar to create one for our app to use.
6. Finally we return the result. Notice if the new calendar creation failed the result object will be nil, so the caller can do some error handling

Phew, that was a lot of code! But hey, now we have a class to take care of managing the application's calendar under the hood, so we 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 we'll need to handle the event when the user taps a button on the alert asking him where he wants to import the selected show schedule.

Open **GTMasterViewController.h** modify the interface to mark it as implementing the UIAlertViewDelegate protocol, like so:

```
@interface GTMasterViewController : UITableViewController
<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 objectAtIndex:row]
        toCalendar: [AppCalendar calendar]];
    }
}
```

If buttonIndex is zero, the user chose to import into an already existing calendar. We'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". We then call the method addShow:toCalendar (which we 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"
```

Also, find the comment in the code saying "private methods here" - it's in the body of the class extension. Replace it with the declaration of the addShow:toCalendar:



```
-(void)addShow:(NSDictionary*)show toCalendar:(EKCalendar*)calendar;
```

And 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 objectForKey:@"startDate"]];
    event.endDate   = [frm dateFromString:
                       [show objectForKey:@"endDate"]];
}
```

We start with the basics. [EKEvent eventWithEventStore:] creates a new event in the given event store, and by setting the calendar property we tell the event in which calendar it belongs.

Since the show start and end date 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 - 09/29/2011 19:00 PST (oooh, 24 hour input - how European! :)) Using the date formatter we turn the 2 strings from the plist to NSDate objects and we 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:

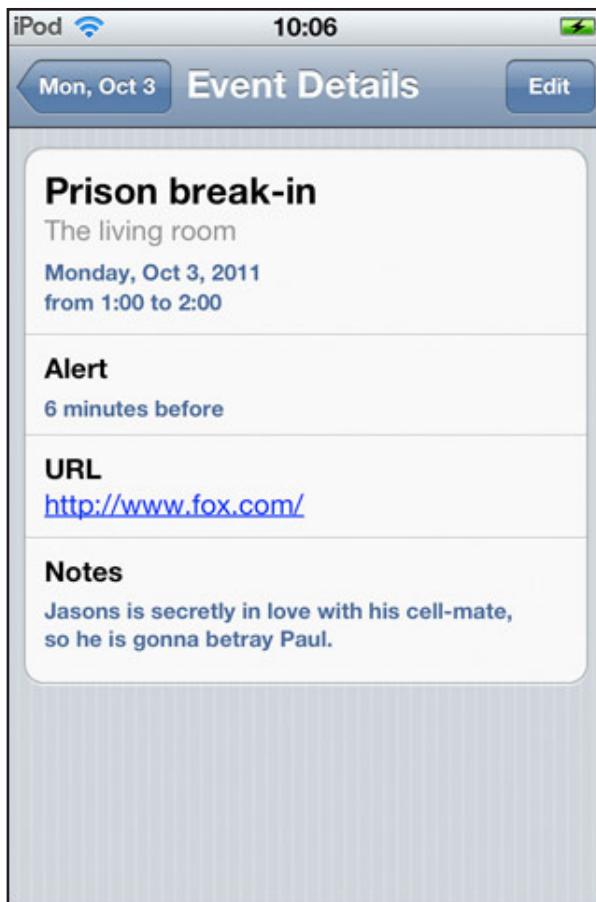
```
event.title = [show objectForKey:@"title"];
event.URL = [NSURL URLWithString:[show objectForKey:@"url"]];
event.location = @"The living room";
event.notes = [show objectForKey:@"tip"];
```

More good stuff - the title of the show goes as title of the event, we set also the URL property of the event, the location is just a string - for all shows we set it to



"The living room" and finally we 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 we're about to create only one calendar entry for the very first episode only!

To fix this, we 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, we're going to make our 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 dayOfWeek: [weekDay intValue] ];
```



```
//2
EKRecurrenceEnd* runFor3Months =
    [EKRecurrenceEnd recurrenceEndWithOccurrenceCount:12];
//3
EKRecurrenceRule* myRecurrence =
[[EKRecurrenceRule alloc]
 initRecurrenceWithFrequency:EKRecurrenceFrequencyWeekly
 interval:1
 daysOfTheWeek:[NSArray arrayWithObject:showDay]
 daysOfTheMonth:nil
 monthsOfTheYear:nil
 weeksOfTheYear:nil
 daysOfTheYear:nil
 setPositions:nil
 end:runFor3Months];
[event addRecurrenceRule: myRecurrence];
```

We get the day of the week the show runs from the property list dictionary (1 means Sunday and so forth) and then we 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. We 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 our example we 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 PST 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!

So finally we'll save the event to the event store and show the event edit dialogue to the user, so he can make any changes if he'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 presentModalViewController:editEvent animated:YES];
```



First we call `saveEvent:span:commit:error:` (notice how we 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 pending like we do? Well the changes are present in our `EKEventStore`, but not in other `EKEventStore` instances we have in the same application or say in the iOS Calendar app. So I hope now you realize why we needed a static instance of `EKEventStore` in first place - so we make sure we always use the same object and we for sure see the pending changes from within our 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 we 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 we set the `editViewDelegate` to `self` (our view controller).

So, open up `GTMasterViewController.h` and mark the view controller as implementing the `EKEventEditViewDelegate`, like so:

```
@interface GTMasterViewController : UITableViewController  
    <UIAlertViewDelegate, EKEventEditViewDelegate>
```

Compile and run, and w00t...

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? We have everything pre-filled and we just need to add the `EKEventEditViewController` methods so we can handle taps on Cancel or Done. Let's dig in!

The protocol requires us 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 (we 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 `EKEventEditViewController` method to save the event or cancel it:

```
- (void)eventEditViewController:
    (EKEVENTEditViewController *)controller
didCompleteWithAction:(EKEVENTEditViewAction)action
{
    NSError* error = nil;
    switch (action) {
        case EKEVENTEditViewActionSaved:
            [[AppCalendar eventStore] commit:&error];
    }
}
```



```
        break;
    case EKEventEditViewActionCanceled:
        [[AppCalendar eventStore] reset];
    default:break;
}

[controller dismissModalViewControllerAnimated:YES];
}
```

When the user taps either Cancel or Done, this method will be invoked and the "action" parameter will be either `EKEventEditViewActionCanceled` or `EKEventEditViewActionSaved` respectively.

When the action is `EKEventEditViewActionSaved`, we just call `commit:` on the event store object, so it'll save all pending changes permanently to the disc. When the user cancels we 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 we do is actually dismiss 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 - we 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 display only the user's writable calendars.



darChooserDisplayWritableCalendarsOnly to show only the calendars you can save events to - the latter is definitely what we want.

The EKCalendarChooser needs a delegate - we 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 his selection - we set showsDoneButton to YES, so we're going to implement the second approach.

This view controller you have to push to the navigation controller's stack, so we pass it to pushViewController:animated:. And what the user will see is the default calendar selection dialogue from the iOS Calendar application:



Now open **GTMasterViewController.h** and right after EKEventEditViewDelegate add yet another protocol:

```
@interface GTMasterViewController : UITableViewController
<UIAlertViewDelegate, EKEventEditViewDelegate,
EKCalendarChooserDelegate>
```

Then switch back to **GTMasterViewController.m** and 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];  
}
```

Explanation:

1. The `selectedCalendars` property of `EKCalendarChooser` is an `NSSet` of the selected calendar objects. In our case we allow only single selection, so the "anyObject" will give us back the selected calendar.
2. We get the selected row index in the table and store it into the `row` variable.
3. We call `addShow:toCalendar:` just like before, but this time we pass the selected calendar as parameter.
4. Finally just pop the calendar chooser view controller - and we're finished!

OK - we 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 - we'll have to take a funky approach to adding the alarms functionality to our application.

So the solution we're 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 quick!



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 we can delete it any time.

But first let's deal with a hidden issue (oh Apple you did it again!). 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 EKEventEditViewActionCancelled action (as before), but if the user just taps "Done" without making modifications to the pre-filled data - again an EKEventEditViewActionCancelled event will get passed back to the delegate.

Ack! So as a workaround, we'll 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 presentModalViewController:editEvent animated:YES];
```

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 we can adjust also the logic inside eventEditViewController:didCompleteWithAction:. Remove the "switch" statement and replace it with this new one:

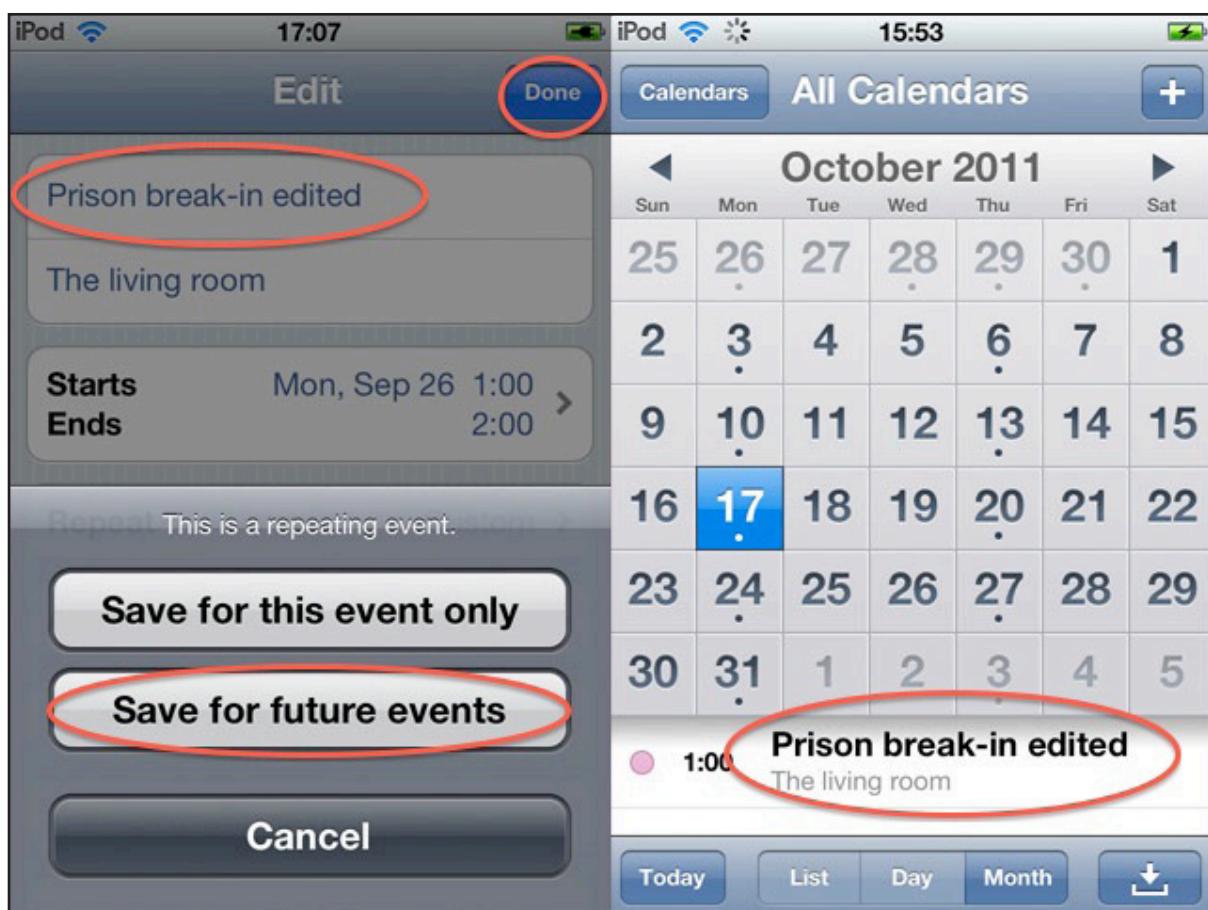
```
switch (action) {
    case EKEventEditViewActionDeleted:
        [[AppCalendar eventStore] removeEvent:controller.event
            span:EKSpanFutureEvents error:&error];
    default:break;
}
```



Say what? 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

You can give that a try:



Adding Alarms to the Calendar

And finally we can add alarms to our calendar entries! Go to the code where we create the event and below the line `event.calendar = calendar;` add also this code:



```
EKAlarm* myAlarm = [EKAlarm alarmWithRelativeOffset: - 06*60 ];  
[event addAlarm: myAlarm];
```

Here we 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 we set the alarm to fire up 6 minutes before the show starts.

That's all! You can give the alarms a try and we 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.



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, we're 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 much 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" 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:

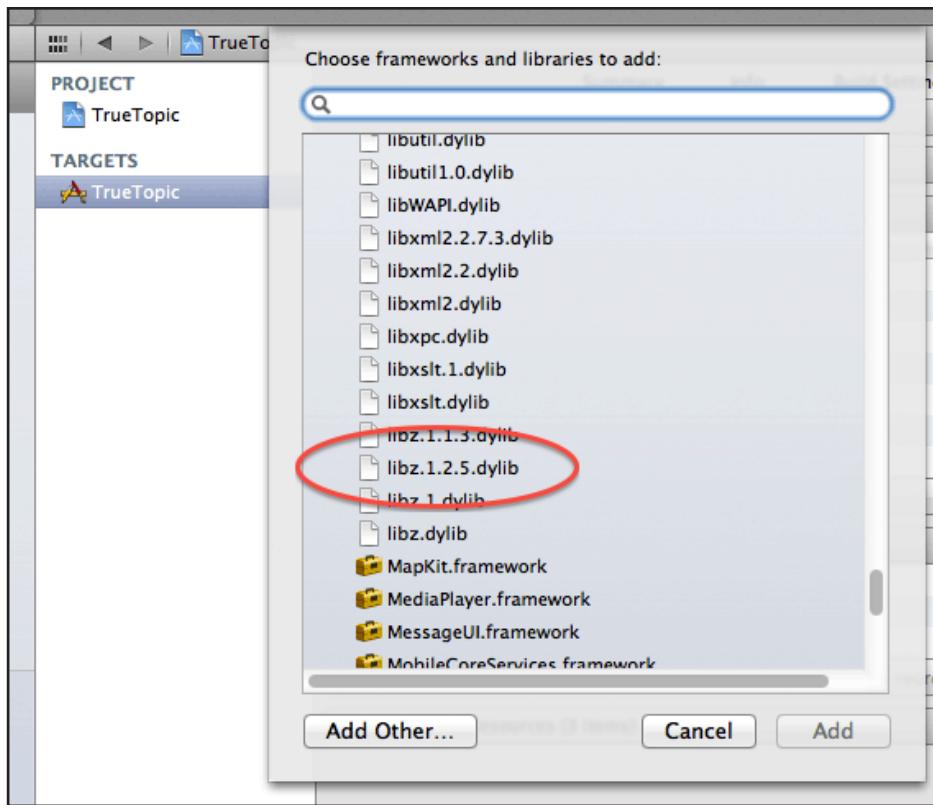
1. Open in your browser the [RaptureXML GitHub page](#)
2. Click the "ZIP" button to download a .zip file to your computer
3. Unarchive the downloaded file and open the RaptureXML folder
4. Inside there's another folder called RaptureXML (with RXMLElement.h and several other files inside), drag it from Finder and into Xcode's Project Navigator
5. 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 library, so let's add it to the project.

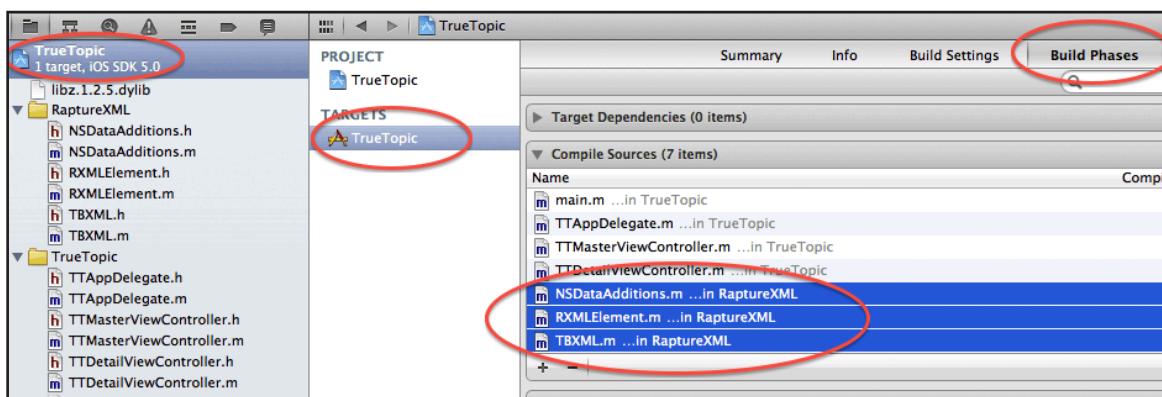
1. Click the project file in the Project navigator and select the TrueTopic target
2. Click on the "Build phases" tab and unfold the "Link binary with libraries" strip
3. Click on the "+" button and double-click "libz.1.2.5.dylib" from the list





At the time of writing this chapter, RaptureXML is not ARC compatible. If this changes by the time you're reading it, just skip the next section :]

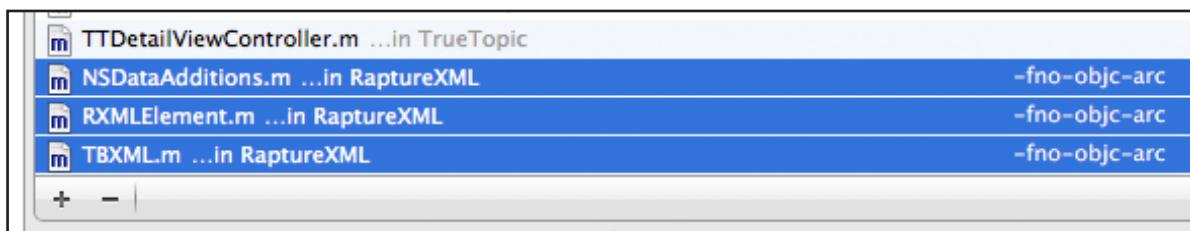
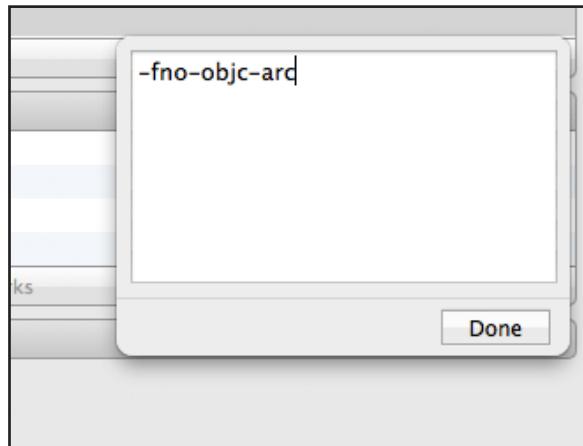
If you hit Cmd-B right now to compile the project you'll get a number of errors, mostly complaining about the use of "autorelease" in the classes we just imported. So let's fix this real quick:



1. While in the "Build phases" tab - unfold the "Compile sources" list
2. As you see the second column in the list is "Compiler flags", we need to pass the "-fno-objc-arc" flag to the compiler for each file in RaptureXML



3. Select the 3 files from the RaptureXML library (NSDataAdditions.m, RXMLElement.m, and TBXML.m) and press "Enter" on your keyboard
4. In the little popup window just enter "-fno-objc-arc" (no quotes) and click "Done"



Cool! This instructs the compiler (and Xcode) to disable ARC for the 3 files from the RaptureXML. This is per-file turning off of ARC for ya! Hit Cmd-B to build the project and make sure that everything compiles just fine. If you still get the same errors, you might have to quit Xcode and restart.

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 we 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"]
```

We'll also need to store the RSS articles in an array and make it the data source for the table view. Create an instance variable to keep track of the articles by replacing the @implementation line with the following:

```
@implementation TTMasterViewController {
```

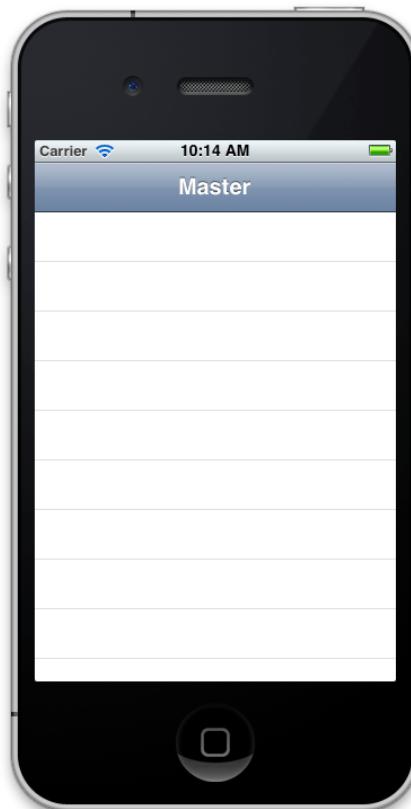


```
NSMutableArray* articles;  
}
```

Next replace the `tableView:numberOfRowsInSection:` method as follows:

```
- (NSInteger)tableView:(UITableView *)tableView  
    numberOfRowsInSection:(NSInteger)section  
{  
    return [articles count];  
}
```

We have now the project ready to run. In fact hit Run in Xcode to make sure you followed everything correctly so far! It should be a blank table, but that's fine for now.



Fetch the Remote RSS Feed

Next we're 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.

So 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
```

And inside **ArticleData.m** put this:

```
#import "ArticleData.h"

@implementation ArticleData
@synthesize title, topic, link;
@end
```

As you see we just need 3 properties and nothing else.

Now open up **TTMasterViewController.m** and add this two imports below the defines at the top:

```
#import "RXMLElement.h"
#import "ArticleData.h"
```

Next we need to fetch the RSS feed, iterate over the "item" elements inside the feed, and the articles array with ArticleData instances. We are going to do that work in `viewDidLoad`, so find it and add this to the bottom:

```
articles = [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 we don't do so much: we initialize the `articles` array as we're going to be adding items inside. Then in a background queue (using the `kBgQueue` macro we created earlier) we 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. I love these kind of APIs that save you all the heavy work!

On the next line we just call:

```
[xml child:@"channel"]
```

which returns `RXMLElement` object for the channel child tag and we immediately call the following on it:

```
children:@"item"];
```

`children:` gives us back array of XML elements. I told you RaptureXML is going to save us 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];  
    [articles addObject: data ];  
}
```

Pretty simple - inside the loop we just create an `ArticleData` instance and set the title and link. We then add the new item to the articles list.

Finally, we have to show the results on the screen, so we'll have to reload the table view. Note we 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, we need to edit `tableView:cellForRowAtIndexPath:` to show the article titles:

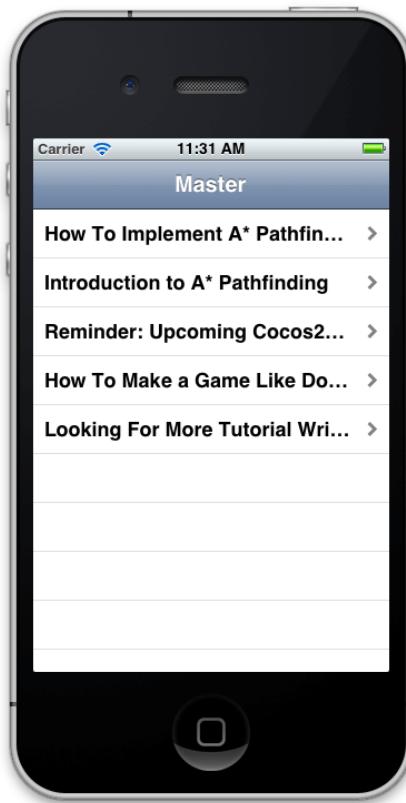
```
- (UITableViewCell *)tableView:(UITableView *)tableView  
    cellForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    static NSString *CellIdentifier = @"Cell";  
    UITableViewCell *cell =  
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];  
    if (cell == nil) {  
     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



```
    reuseIdentifier:CellIdentifier];
cell.accessoryType =
UITableViewCellAccessoryDisclosureIndicator;
}

// Fill in the article data
ArticleData* data = [articles objectAtIndex: indexPath.row];
cell.textLabel.text = data.title;
cell.detailTextLabel.text = data.topic;
return cell;
}
```

Pretty standard - we get a cell with two labels inside (by providing an UITableViewCellStyleSubtitle style to the initializer) and when the cell is created, we get the relevant object from the articles list and just fill in the texts. Now you can hit Run and see the result:



Let me just say how happy I am - there were times when building a simple RSS reader for iPhone was a whole tutorial. Now APIs have progressed so much this little bit of code is all it takes!

Putting NSLinguisticTagger to Work

In this part of the tutorial we 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 to 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"

@implementation TagWorker
{
    NSLinguisticTagger* tagger;
    NSMutableArray* words;
}

-(void)get:(int)number_ofRealTopicsAtURL:(NSString*)url
completion:(TaggingCompletionBlock)block
{
    //initialize the linguistic tagger
    tagger = [[NSLinguisticTagger alloc] initWithTagSchemes:
              [NSArray arrayWithObjects:
               NSLinguisticTagSchemeLexicalClass,
```



```

    NSLinguisticTagSchemeLemma, nil]
options: kNilOptions];

//setup word and count lists
words = [NSMutableArray arrayWithCapacity:1000];
}

@end

```

We have two instance variables - one is the tagger object itself and the other is a list of the word objects. We'll use the words array to store what we find here as we 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. We 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, 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 out plain text, and the article is going to be fetched from a web page. 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 = [NSArray arrayWithObjects:
    @{@"\\A.*?<body.*?>": //1
    @"</body>.*?\\Z": //2
    @"<[^>]+>": //3
    }];

```



```
 @"\"\\W+$", //4
 nil];
```

First we fetch the HTML content of the supplied URL into the text object. Next we define an array with some pretty awkward strings. These are the regular expressions we 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 our purpose 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 we 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. We create the regexes one by one, and then we replace the matches with @"" (no text, empty string)
2. We add a sentence "STOP." at the end of the text - we are going to use that control sentence to detect when we 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:



The diagram illustrates the process of extracting text from a web page. On the left, labeled 'Before', is the original HTML code. It includes a link to a Unity3D tutorial and a paragraph of text about Unity3D. A blue arrow points from the original text to the right side, labeled 'After', where the text has been processed by the Linguistic Tagger API, resulting in clean, readable plain text.

```

<div class="post-image-inner right">
  <a href="http://www.touch-code-unity3d.com/wp-content/themes/magazeen/timthumb.php?w=225&h=246&zc=1" rel="bookmark" title="Permanent Link to Tutorial: Building iPhone Games with Unity3D Final Version!"></a>
</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 400$ for the iOS enabled version!</p>

```

Before

```

heard about Unity3D ; this is a 3D software powerful editor / 3D engine, which creates game bi 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 Unity; series. I wr layout and basic interactionsetting up a scene in audio effectsadding particle systemscreating 2D an to go trough the tutorials you don't need an trial of Unity to play around with it! I wish you l 2.5D game with Unity ; Part 2Marin http://ti Tutorials and excerpts about iPhone and iPad progr iphone-games-with-unity3d/Tutorial: Building iPhon on del.icio.usShare this on FacebookDigg this!Shar developer and publisher. He's got more than 18 yea Marin's homepage &nbsp;&nbsp; &gt; Contact &nbs

```

After

It's not perfect plain text, but is good enough for us. Let's make a little detour for a second. To count the nouns in the text we'll need a class which can hold a string (the word) and the number of occurrences we found. Let's create this class right now, before we 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 we'll have 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
@synthesize word, count;

+(WordCount*)wordWithString:(NSString*)str
{
    WordCount* word = [[WordCount alloc] init];
```



```

    word.word = str;
    word.count = 1;
    return word;
}

@end

```

Pretty standard code. But let's get to something more interesting. We'll 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 we will need an `isEqual:` method to our WordCount class.

Also in the end we will want only the most popular words, so we will have to order the array of found words, thus we will need also a sorting helper method to our WordCount class. Add them now to the implementation:

```

- (NSComparisonResult)compare:(WordCount *)otherObject {
    return otherObject.count-self.count;
}

- (BOOL)isEqual:(id)otherObject
{
    return [self.word compare:
           ((WordCount*)otherObject).word]==NSOrderedSame;
}

```

So, we're using a bit of trickery here. When WordCount objects need to be sorted, we'll be using `compare:` method which compares their count of occurrences. For the `isEqual:` method on the other hand, which will be invoked automatically from `[NSArray containsObject:]` we 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!

We're 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)

```

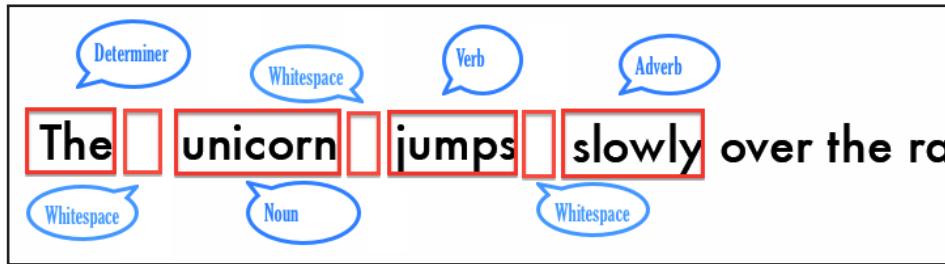


```
{
    //process the tags

    //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 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, 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 we set for the tagging process. The scheme is `NSLinguisticTagSchemeLexicalClass`, which will give us back parts of speech: verbs, nouns, adverbs, adjectives, etc. As "options" we 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 we 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] ];
    double index = [words indexOfObject: word];

    if (index != NSNotFound) {
        //existing word - just increase count
        ((WordCount*)[words objectAtIndex: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. We're interested in `NSLinguisticTagNoun` only.

Next we create a `WordCount` instance with the noun found. We get the index of the `WordCount` for that word inside the `words` list.

So if the index that `indexOfObject:` returns is not `NSNotFound` we cast the element inside the array to a `WordCount` instance and increase the `count` property. Otherwise this is a new word, and we can just add it to the word list.

Phew! Lot of coding. There's just one last final step of the tagging and we'll 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:

```

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 we are sure 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 we just call `sortUsingSelector:` to sort the words array. It'll sort itself in decreasing order (we need the most mentioned words).

Next we create a range which will get us `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 we make a range for just as many as there are found.

Finally we invoke the completion block and we pass over a subarray of the words array - containing just the number of words asked for.

Wow! That concludes the `TagWorker` class. Very soon we will be able to see some results.

Bringing it All Together

Now that we have our 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 articles) {

    TagWorker* worker = [[TagWorker alloc] init];
    [worker get:5 ofRealTopicsAtURL: data.link
        completion:^(NSArray *words) {
            data.topic = [words componentsJoinedByString:@" "];
            //show the real topics
        }];
}
```

We loop over the fetched RSS articles and for each we create a `TagWorker` and tell it to get the 5 most mention words from the article's web page. Upon completion we just join together the returned words and store them in the `topic` property in our table data source.

One detail though, now `data.topic` contains something along the lines of this:

```
<WordCount: 0x68cdf80> <WordCount: 0x68cc1a0> <WordCount: 0x68cc4e0>
```



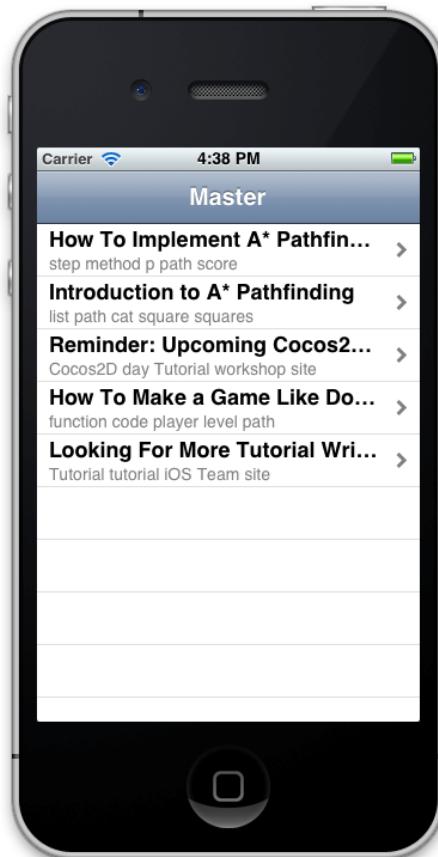
Time for another little trick - `componentsJoinedByString:` actually calls the `description` method on each object, so we're going to override `description` for the `WordCount` class and make it return the stored word. Open up again **WordCount.m** and add this method:

```
-(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 mega-zargoyan-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 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's 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 NSLinguisticTagSchemeLemma 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

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 in iOS 5 and should be ready to start using these cool new techniques 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 come up with!

If you have any questions or comments as you go along, please stop by our book forums at raywenderlich.com/forums.

Thank you again so much 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, Jacob, Matthijs, Felipe, Cesare, Marin, and Ray from the iOS Tutorial Team

