



P r o f e s s i o n a l E x p e r t i s e D i s t i l l e d

iPad Enterprise Application Development BluePrints

Design and build your own enterprise applications for the iPad

Foreword by Cory Bohon

Steven F Daniel

[PACKT] enterprise 
PUBLISHING professional expertise distilled

www.it-ebooks.info

iPad Enterprise Application Development BluePrints

Design and build your own enterprise applications for the iPad

Steven F Daniel



BIRMINGHAM - MUMBAI

iPad Enterprise Application Development BluePrints

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2012

Production Reference: 1150912

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-294-7

www.packtpub.com

Cover Image by Dean Morel (deangmorel@gmail.com)



This material is copyright and is licensed for the sole use by on 7th October 2012

www.it-ebooks.info

Credits

Author

Steven F Daniel

Project Coordinator

Yashodhan Dere

Reviewers

Cory Bohon

Gareth Curtis

Clifford Sharp

Luciano Tolfo

Proofreader

Mario Cecere

Maria Gould

Aaron Nash

Indexer

Hemangini Bari

Acquisition Editor

Rukshana Khambatta

Graphics

Aditi Gajjar

Lead Technical Editor

Arun Nadar

Production Coordinator

Melwyn Dsa

Technical Editor

Lubna Shaikh

Cover Work

Melwyn Dsa

Foreword

With the world changing and technology evolving year on year, businesses are constantly growing more dependent on technology at an ever-expanding rate. With the iPad, the clear cut winner in the tablet market, businesses are making use of the new devices from Apple in their own workflows. The iPad dramatically changes how companies can interact with their data – whether accessing or collecting data in the field or in the office.

This growing market is lending itself to more useful enterprise applications that can be used to drive businesses into the future. Unfortunately, many companies don't know where to begin when developing their own custom applications for their own use. That's where this book comes in.

This book holds your hand and guides you through the building of practical example applications in each chapter. In each chapter, you will learn various frameworks and technologies in iOS, to create stunning applications that take advantage of the iPad features. The features and techniques that you learn in this book can directly be taken and used in your own iPad enterprise application development.

-Cory Bohon

About the Author

Steven F Daniel is originally from London, England, but lives in Australia.

He is the owner and founder of GENIESOFT STUDIOS (<http://www.geniesoftstudios.com/>), a software development company based in Melbourne, Victoria that currently develops games and business applications for the iOS, Android, and Windows platforms.

Steven is an experienced software developer with more than 13 years of experience in developing desktop and web-based applications for a number of companies, including insurance, banking and finance, oil and gas, and local and state government.

Steven is always interested in emerging technologies, and is a member of the **SQL Server Special Interest Group (SQLSIG)** and Java Community. He was the co-founder and **Chief Technology Officer (CTO)** of SoftMpire Pty Ltd., a company that focuses primarily on developing business applications for the iOS and Android platforms.

He is the author of *Xcode 4 iOS Development Beginner's Guide* and *iOS 5 Essentials*.

You can check out his blog at <http://geniesoftstudios.com/blog/>, or follow him on Twitter at <http://twitter.com/GenieSoftStudio>.

Acknowledgement

No book is the product of just the author—he just happens to be the one with his name on the cover. A number of people contributed to the success of this book, and it would take more space than I have to thank each one individually.

A special shout-out goes to Amey Kanse, my Acquisition Editor, who is the reason that this book exists. Thank you Amey for believing in me and for being a wonderful guide throughout this process. I would like to thank Yashodhan Dere for ensuring that I stayed on track and got my chapters in on time, and to Rukhsana Khambatta for taking over as the Acquisition Editor for this book so quickly and brilliantly, during Amey's departure.

I would also like to thank my Lead Technical editor, Arun Nadar, for his brilliant suggestions on how to improve the chapters, and a special thanks to Lubna Shaikh for the fantastic job she has done on this book, ensuring that we met our timeframes and delivery for this book. It has been a great privilege to work with her again on this book.

Lastly, to my reviewers, thank you so much for your valued suggestions and improvements, making this book what it is. I am grateful to each and every one of you.

Thank you also to the entire Packt Publishing team for working so diligently to help bring out a high quality product. Finally, a big thank you to the engineers at Apple for creating the iPad, and providing developers with the tools to create fun and sophisticated applications. You guys rock.

Finally, I'd like to thank all of my friends for their support, understanding, and encouragement during the writing process. It is a privilege to know each and every one of you.

About the Reviewers

Cory Bohon is a professional writer and contributor to MacLife Magazine, and a Mac and iPhone developer, experienced in Java, C/C++, Objective-C, and PHP. He is currently working on a Masters degree in Software Engineering, where his current research interests includes accessible user interface design and mobile application development.

Gareth Curtis was learning to program for the BBC Master computer when he was 10 years old. A career in I.T. was always on the cards and this began in a corporate finance environment. It wasn't until late 2008 when the first iPhone SDK was released by Apple that he really took an interest in development. A few months later, he achieved one of his ambitions in the forming of his own company, Appfidelity Ltd. Appfidelity has since successfully been developing apps for the iPhone, and later the iPad, for a wide variety of clients including apps for sports, finance, fashion, and entertainment. More recently, Gareth has also entered into the realms of iBook publication.

Clifford Sharp has been in the computer industry for over 30 years. In the first 15 years, he performed network and systems administration using VAX/VMS and DECnet then Linux and TCP/IP. In the next 10 years, he designed and created Linux system programs as well as database front-end software using C and Pro*C with Oracle. The last 5 years have been all about iOS Architecture and Development, where he has created iOS apps for AT&T, Network Solutions, DirecTV, Experian, among others.

Luciano Tolfo is a creative and proactive software engineer with more than five years of experience, currently specialized in iOS applications and game development, who loves what he does and enjoys facing new challenges. His background is in the game industry, and he is now working as a full-time freelance mobile developer.

I would like to thank my family and girlfriend for their unconditional support and their patience while I work long hours and for the time I spent reviewing this book. This was the first time I made a technical review for a book and I really enjoyed the process, and I would like to contribute with my feedback on further iOS development books.

You can see my Linked-In profile at <http://www.linkedin.com/in/lucianotolfo>.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter, or the *Packt Enterprise* Facebook page.

This book is dedicated:

To my favorite uncle, Benjamin Jacob Daniel, for always making me smile and for inspiring me to work hard and achieve my dreams. I miss you a lot.

To Choi Chun Chiet, for the encouragement and support during the writing of this book.

To Chan Ban Guan, for the continued patience, encouragement, and support, and most of all for believing in me during the writing of this book.

To my family, for always believing in me and for their continued love and support.

To my niece, Ava Madison Daniel, thank you for continually bringing joy to our family.

To the late Steve Jobs—you will always be an inspiration and a guide towards perfection. Thank you for all the amazing things you've brought to our lives. May you rest in peace.

This book would not have been possible without the love and understanding of everyone I've mentioned. I would like to thank you all from the bottom of my heart.

Table of Contents

Preface	1
Chapter 1: Getting and Installing the iOS SDK	7
Getting and installing the iOS SDK	8
The iOS Simulator	12
Layers of the iOS architecture	13
The Core OS layer	14
The Core Services layer	15
The Media layer	16
The Cocoa-Touch layer	18
Building the HelloWorld application	19
Placing objects within the View	22
Removing the Xcode Developer Tools	24
Summary	25
Chapter 2: Task Priorities – Building a TaskPriorities iOS App	27
Building the TaskPriorities app	28
Adding the required frameworks	30
Creating the main application screen	31
Handling multiple screen orientations when the device is rotated	32
Adding the table control to hold item data	33
Adding the Add button	40
Adding the Refresh button	41
Navigating between screens using Storyboards	51
Implementing the Save record method	61
Implementing the Add a record to the table method	61
Implementing the Cancel method	62
Implementing the Refresh button method	62
Implementing the Delete row method	63
Finishing up	64
Summary	65

Chapter 3: VoiceRecorder App – Audio Recording and Playback	67
Overview of the technologies	68
Building the VoiceRecorder app	68
Adding the AVFoundation and MessageUI frameworks	70
Creating the main application screen	72
Adding the Start Recording button	73
Adding the Play button	74
Adding the Stop button	75
Adding the E-mail button	76
Implementing the View Controller class	80
Implementing the voiceRecord method	83
Implementing the voicePlayback method	85
Implementing the voicePlaybackStop method	86
Implementing the e-mailRecording method	86
Implementing the VoiceVisualizer class	89
Finishing up	92
Summary	94
Chapter 4: Enhanced AddressBook App – Core Data	95
Overview of the Core Data technologies	96
Building the AddressBook application	98
Adding the GameKit framework	100
Building the Core Data model	102
Creating our Core Data model files	104
Adding the Storyboard screen	108
Creating the main application screen	112
Adding a table control to hold the item data	112
Adding the Add button	116
Adding the Action button	117
Navigating between screens using Storyboards	125
Implementing the save record method	132
Implementing the cancel method	133
Implementing the delete row method	133
Implementing the didSelectRowAtIndexPath method	134
Transferring contact details using Bluetooth	135
Implementing the connect method	137
Implementing the Action button method	140
Finishing up	143
Implementing the search functionality	144
Summary	150
Chapter 5: BatteryMonitor Application	151
Overview of the technologies	152
Building the BatteryMonitor application	153
Adding the MessageUI framework to the project	154
Creating the main application screen	155

Adding the Enable Monitoring UISwitch control	156
Adding the Send E-mail Alert UISwitch control	157
Adding the Fill Gauge Levels UISwitch control	158
Adding the Increment Bars UIStepper control	158
Adding the System Information (UITextView) control	160
Building the Battery Monitor functionality	164
Implementing the View Controller class	165
Implementing the determineBatteryStatus: method	167
Implementing the enableMonitoring: method	170
Implementing the sendEmailAlert: method	172
Implementing the fillGauge: method	174
Implementing the totalNoBars: method	175
Implementing the Battery Gauge class	176
Finishing up	184
Summary	185
Chapter 6: RouteTracker Application	187
Overview of the technologies	188
Building the RouteTracker application	189
Adding the Core Location and MapKit frameworks	190
Creating the main application screen	193
Adding the Start Tracking button	193
Adding the Refresh Map button	194
Adding the Change Map Type button	195
Building the RouteTracker functionality	200
Implementing the View Controller class	201
Implementing the startTracking: method	204
Implementing the refreshMap: method	205
Implementing the changeMapType: method	206
Implementing the locationManager: method	207
Implementing the locationManager:didFailWithError: method	208
Implementing the shouldAutorotateToInterfaceOrientation: method	210
Implementing the TrackMapView class	210
Finishing up	215
Summary	217
Chapter 7: VeterinaryClinic Application	219
Overview of the technologies	220
Building the VeterinaryClinic application	220
Building the Core Data model	222
Creating our Core Data model files	226
Adding the Storyboard screen	230
Creating the main application screen	231
Adding the table control to hold pet information	232
Adding the Add button	234
Adding the Edit button	234
Navigating between screens using Storyboards	244

Functionality	255
Implementing the btnSavePet: method	258
Implementing the btnCancel: method	260
Implementing the btnAddPhoto: method	260
Implementing the btnCameraPhoto: method	261
Implementing the Delete row method	262
Finishing up	264
Summary	267
Chapter 8: Social Networking Application	269
Overview of the technologies	270
Downloading the Facebook iOS SDK	271
Registering your iOS app with Facebook	272
Building the Social Networking application	276
Adding the Facebook iOS SDK to our project	277
Creating the main application screen	280
Adding the Sign-in button	280
Adding the Sign-out button	281
Adding the Action button	282
Building the Facebook app functionality	286
Implementing SSO within your app	286
Implementing the Application Delegate class	287
Implementing the View Controller class	292
Adding the LogOut functionality to your app	295
Requesting additional permissions	296
Using the Graph API	298
Integrating with social channels	302
How to handle errors	304
Implementing the postMessageButton: method	305
Implementing the loginButton: method	306
Finishing up	307
Summary	309
Chapter 9: External Displays using Airplay and Core Image	311
Overview of the technologies	312
Building the ExternalDisplays application	312
Adding the Media Player framework to our project	314
Creating the main application screen	315
Adding the Browse button	316
Adding the Camera button	316
Adding the Play Video button	317
Adding the Transitions button	317
Adding the VGA Out button	317

Functionality	320
Implementing the View Controller class	320
Implementing the btnBrowse: method	323
Implementing the btnCamera: method	324
Implementing the btnPlayVideo: method	327
Using AirPlay to present application content to Apple TV	329
Implementing the btnTransitions: method	332
Understanding the Core Image framework	333
Applying image filter effects using the CImage class	335
Applying transitions to images	340
Presenting content out to an external monitor device	342
Implementing the shouldAutorotateToInterfaceOrientation: method	344
Finishing up	345
Summary	346
Chapter 10: Storing Documents within the Cloud	347
Overview of the technologies	348
Methods to store and use documents within iCloud	348
The file coordinator	349
The file presenter	349
Using the iCloud storage APIs	350
Handling iCloud file-version conflicts	352
Building the ScratchPad application	352
Creating the main application screen	354
Adding the table control to hold iCloud document data	354
Adding the Add button	356
Adding the Edit button	356
Navigating between screens using Storyboards	369
Functionality	376
Implementing the btnSave: method	378
Implementing the btnCancel: method	380
Implementing the AddDocumentDetails: method	380
Implementing the EditDocumentDetails: method	381
Finishing up	381
Requesting entitlements for iCloud storage	383
Configuring your iOS device to use iCloud	388
iCloud storage space	391
Summary	393
Index	395

Preface

The iPad is transforming the way businesses work with the power of mobile solutions; these include the manufacturing, retail services, and medical industries. Using the iPad makes it easy to deliver stunning presentations, collaborate with colleagues remotely, and access important business information from wherever your work takes you.

Some businesses have been using the iPad as a mobile sales tool to help manage all of your customer relationships. With its wireless connectivity, iPad gives you an on-the-spot access to your CRM database for customer information, sales data, and task lists.

iPad Enterprise Application Development BluePrints will help you learn how to build simple, yet powerful iOS 5 applications for the iPad, incorporating: storing documents within the Cloud, Facebook integration, Core Image, Route Tracking, Audio Recording and Playback, as well as monitoring the iOS device battery levels.

In this book, I have tried my best to keep the code simple and easy to understand. I have provided step-by-step instructions with loads of screenshots at each step to make it easier to follow. You will soon be mastering the different aspects of iOS 5 programming, as well as mastering the technology and skills needed to create some stunning applications. Feel free to contact me at geniesoftstudios@gmail.com for any queries, or just want to say "Hello". Any suggestions for improving this book will be highly regarded.

What this book covers

Chapter 1, Getting and Installing the iOS SDK, introduces the developer to the Xcode developer set of tools, as well as the capabilities of the iOS Simulator, and each of the layers contained within the iOS architecture, before finally looking at how to create a simple Hello World iOS application.

Chapter 2, Task Priorities – Building a TaskPriorities iOS App, introduces you to the Storyboards feature, and shows how we can use these to create and configure scenes, to build an application that is capable of storing task-related information. We will also look at how we can apply transitions between each scene, to present these programmatically.

Chapter 3, VoiceRecorder App – Audio Recording and Playback, focuses on learning how we can use the built-in microphone of the iOS device, to record and save audio content for playback later. We will learn how to use the Core Graphics framework to draw a visual representation of the voice input, and then learn how to use the MessageUI framework to attach and e-mail the audio content.

Chapter 4, Enhanced AddressBook App – Core Data, focuses on showing you how to use the Core Data framework to create a simple AddressBook application, to directly interface with a SQLite database, to create and store client information. We will also look at how to incorporate the Bluetooth functionality, so that you can send address book information to another iOS device, and have this information received wirelessly and stored within the database at the other end.

Chapter 5, BatteryMonitor Application, shows how we can use the Core Graphics framework to create and draw a gauge that will be used to represent the total amount of battery life remaining on the iOS device. We will also learn how to use the MessageUI framework to send an e-mail when the battery level falls below a set threshold.

Chapter 6, RouteTracker Application, focuses on how to use the Core Location and MapKit frameworks to monitor the current user's location and heading. We will learn how to use overlays, and overlay this onto the map whenever the route taken by the user changes. The route taken by the user is then visually drawn to the overlay and then applied to the map.

Chapter 7, VeterinaryClinic Application, focuses on how to use the Core Data framework to create a simple VeterinaryClinic application to create and edit pet information, through the use of Storyboards. We will look at how to create the application's database schema, as well as learn how to store images to the database using the iOS device's camera, or manually chosen using the UIImagePickerController control.

Chapter 8, Social Networking Application, shows you how to download the Facebook SDK and register your application with Facebook. It also shows you how to use the Facebook APIs to integrate the Facebook functionality into your app, using the **Single Sign-On (SSO)** feature. This provides users the ability to sign into your application using their Facebook identity, so that they can submit notification

requests, or submit content to their timeline. We will learn how to use the Open Graph API and **Facebook Query Language (FQL)** to pass SQL query-like syntax to retrieve information about the current user, and learn how to cleanly handle Facebook errors within our iOS applications.

Chapter 9, External Displays using Airplay and Core Image, focuses on learning about the Airplay and Core Image frameworks, and how to go about using and implementing these into our applications. This chapter also explains the different image filter effects, how to implement transition animations to produce a water ripple effect. It also covers how to incorporate Airplay and VGA-Out functionality into your application, so that you can have your application displayed out to an external device, such as Apple TV or a VGA monitor.

Chapter 10, Storing Documents within the Cloud, introduces you to the benefits of using iCloud, and how to incorporate the iCloud functionality into your applications to store and retrieve files, and its data through the use of the Storage APIs. This chapter will also give you some insight into how to go about handling file-version conflicts when multiple copies of the same file are being updated on more than one iOS device.

Bonus chapter, Packaging and Distributing Your Applications (online: [http://www.packtpub.com/sites/default/files/downloads/Packaging and Distributing Your Applications.pdf](http://www.packtpub.com/sites/default/files/downloads/Packaging_and_Distributing_Your_Applications.pdf)), introduces you to the Apple Human Interface Guidelines, as well as focusses on how to effectively use Instruments within our applications to eliminate bottlenecks that could potentially cause our application to crash on the user's iOS device. We will also take a look at the necessary steps required to successfully submit your applications to the App Store, and explain how to register devices for testing, and how to create and obtain provisioning profiles for both development and distribution.

What you need for this book

This book assumes that you have an Intel-based Macintosh running Snow Leopard (Mac OS X 10.6.2, or later). I would highly recommend upgrading to Lion or Mountain Lion, as there are many new features in Xcode that are available only to these two operating systems.

We will be using Xcode 4.4.1, which is the integrated development environment used for creating applications for iOS development. You can download the latest version of Xcode at the following URL: <http://developer.apple.com/xcode/>.

Who this book is for

If you are an iPad application developer looking forward to building enterprise applications that interact with Facebook, iCloud, Core Location, and the Core Image frameworks into your applications, then this book is for you. You should have a good knowledge of and programming experience with Objective-C and have used Xcode 4.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Enter in HelloWorld as the name for your project."

A block of code is set as follows:

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

@interface TasksViewController : UITableViewController

@property (nonatomic, strong) NSMutableArray *tasks;

@end
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#import "Task.h"

@implementation Task


@synthesize taskName;
@synthesize description;
@synthesize priority;
@synthesize dueDate;


@end
```

Any command-line input or output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
   /etc/asterisk/cdr_mysql.conf
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " Click on the **Next** button to proceed to the next step in the wizard."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting and Installing the iOS SDK

Welcome to the exciting world of iOS programming for the iPad using iOS 5. This latest release of the mobile operating system is packed with some great new features and improvements to the way things used to be done. When Apple hosted its yearly World Wide Developer Conference in June 2011, it introduced more than 200 new features, and an updated SDK that features over 1,500 new development APIs.

The iPad is transforming the way businesses work with the power of mobile solutions. These include the manufacturing, retail services, and medical industries. Using the iPad makes it easy to deliver stunning presentations, collaborate with colleagues remotely, and access important business information from wherever your work takes you.

Some businesses have been using the iPad as a mobile sales tool to help manage all of your customer relationships. With its wireless connectivity, iPad gives you on-the-spot access to your CRM database for customer information, sales data, and task lists.

My goal of this chapter is to introduce you to each of the layers of the iOS architecture, as well as the capabilities of the iOS simulator. We will take a look at the steps involved in getting and installing the Xcode Developer Tools that come as a part of the **Software Development Kit (SDK)**, before finally looking at how to create a simple HelloWorld iOS application.

In this chapter we will:

- Download and install the Xcode development tools
- Learn about the iOS Simulator and the iOS architecture
- Learn about the different frameworks of the iOS SDK

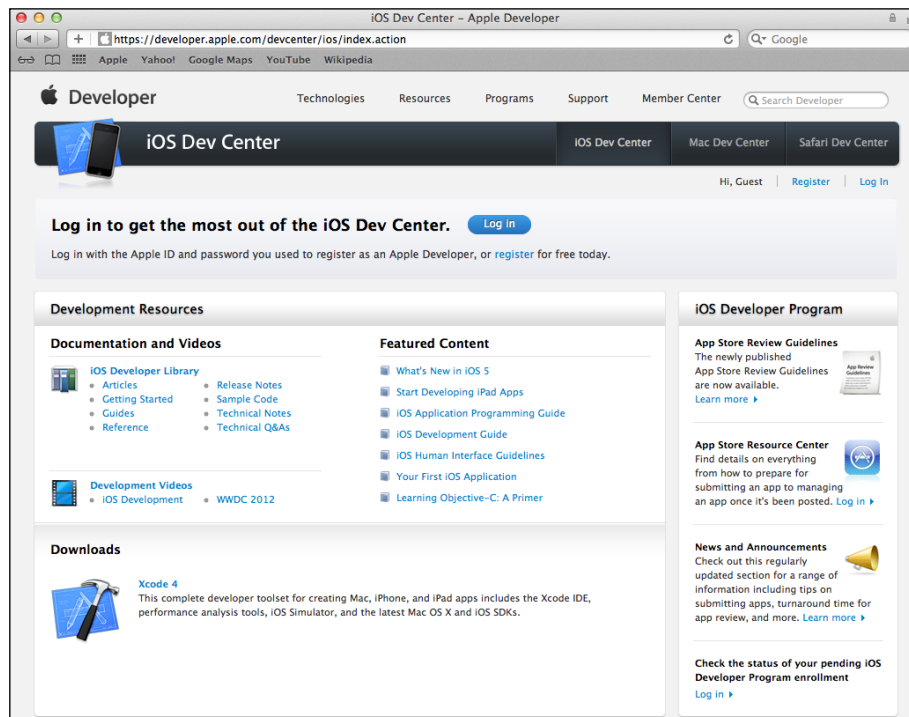
- Create a simple HelloWorld application for the iPad
- Learn how to uninstall the Xcode development tools

We have a fantastic journey ahead of us, so let's get started.

Getting and installing the iOS SDK

Before you can start building iOS applications, you must first sign up as a registered user of the iOS Developer Program. The registration process is free, and provides you with access to the iOS SDK and other developer resources that are really useful for getting you started.

To sign up, you will need to go to <https://developer.apple.com/programs/ios/>, then click on the **Log In** button to proceed, as shown in the following screenshot:



When you become a member, you will have access to numerous resources to help you get started. The following is a short list of some of the things that you will be able to access upon becoming a member of the iOS Developer Program:

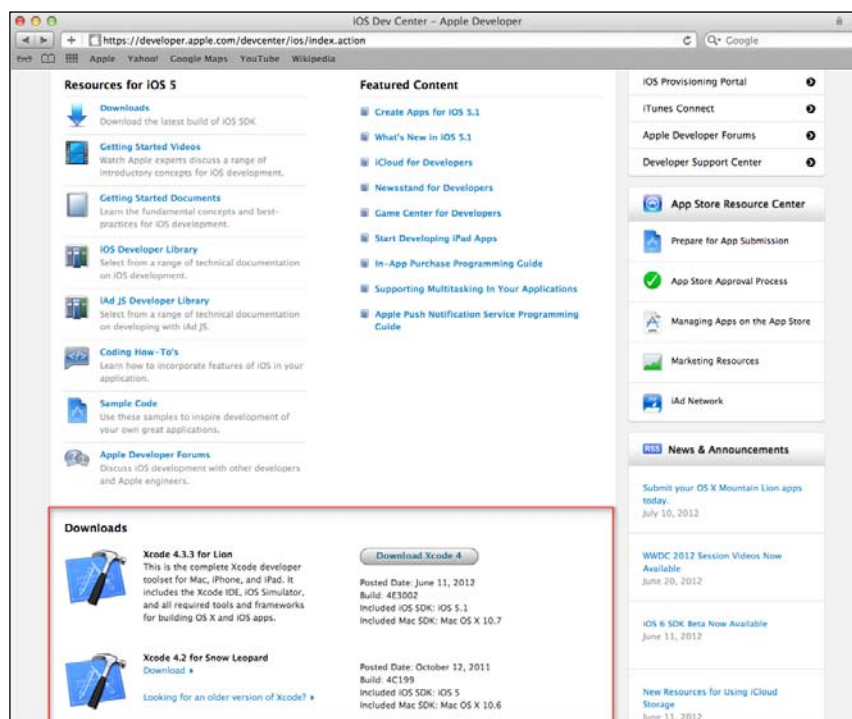
- Helpful getting started guides to help you get up and running quickly
- Helpful tips that show you how to submit your apps to the App Store
- Ability to download current releases of the iOS software
- Ability to trial Beta releases of the iOS software and the iOS SDK
- Access to the Apple Developer Forums

Once you have signed up, you will be able to download the iOS SDK, as shown in the next screenshot. It is worthwhile making sure that your machine satisfies the following system requirements prior to downloading the iOS SDK:

- Only Intel Macs are supported, so if you have another processor type (such as the older G4 or G5 Macs), you're out of luck
- You have updated your system with the latest Mac OS X software updates for either Mac OS X Lion or Snow Leopard

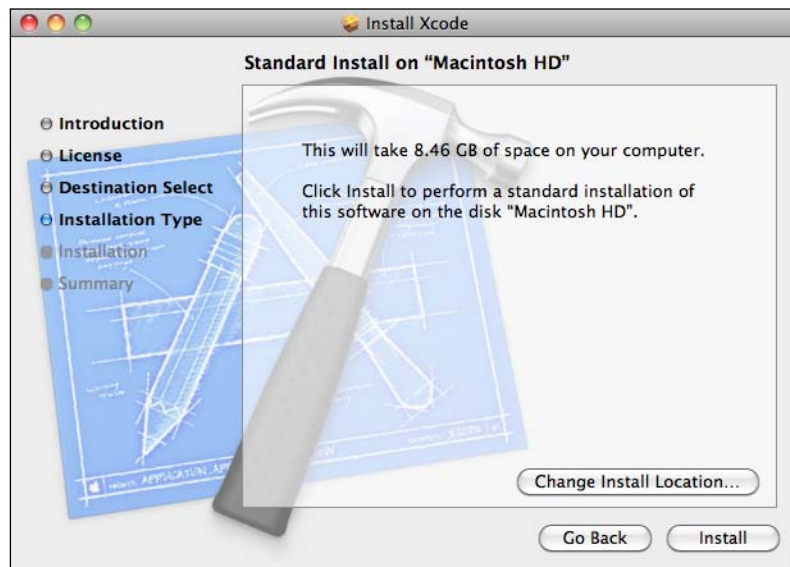


Developing applications for the iPad uses the same **Operating System (OS)** as the iPhone. So you can still use the iPhone SDK. This SDK allows you to create universal applications that will work with both the iPhone and iPad running on iOS 4 and above.

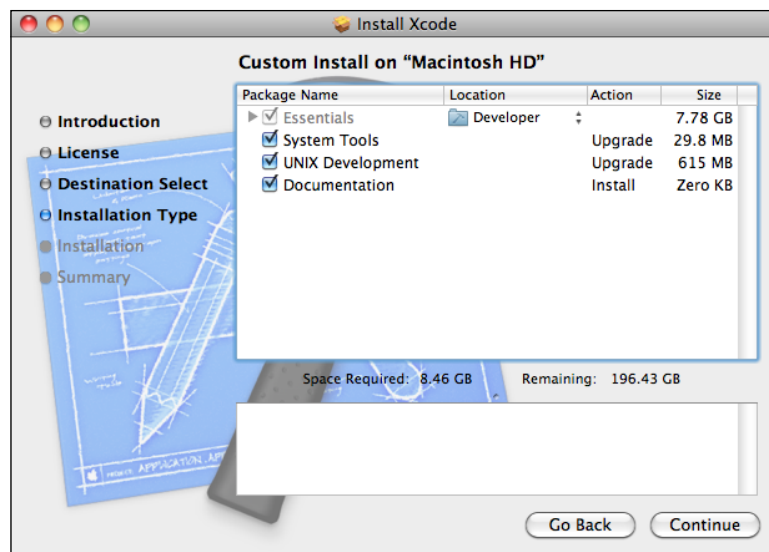


Xcode can also be obtained from the Mac App Store at <http://itunes.apple.com/us/app/xcode/id497799835?mt=12>, depending on whether you have chosen the version for Mac OS X Lion. The installation procedure in the following section shows how to go about installing the iOS development tools for Snow Leopard.

Once you have downloaded the SDK for Snow Leopard, you can proceed with installing it. You will be required to accept a few licensing agreements. You will then be presented with a screen to specify the destination folder in which SDK is to be installed:



If you select the default settings during the installation phase, the various tools (which are explained in detail next) will be installed in the `/Developer/Applications` folder. The installation process takes you through the custom installation option screens. You probably would have seen similar screens to this if you have installed any other Mac software. The following screenshot shows what you will see:



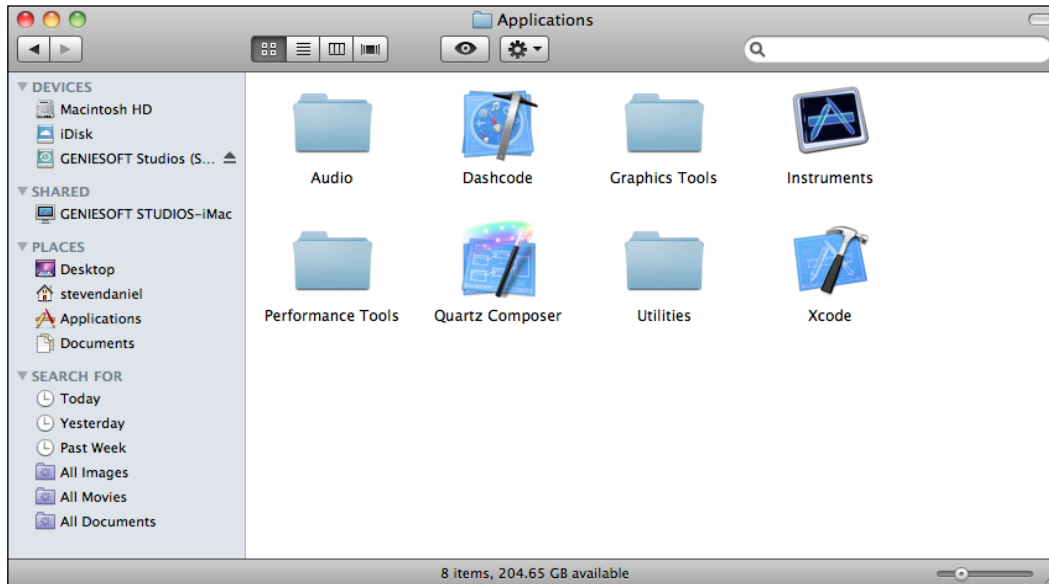
The options in the preceding screenshot give you a little more control over the installation process. For example, you are able to specify the folder location to install Xcode, as well as settings for a variety of other options.

The iOS SDK comes as part of the Xcode developer tools download, which you'll find at <https://developer.apple.com/devcenter/ios/index.action>.

The SDK consists of the following components:

- **Xcode:** This is the main **Integrated Development Environment (IDE)** that enables you to manage, edit, and debug your projects
- **DashCode:** This enables you to develop web-based iOS applications and dashboard widgets
- **iOS Simulator:** This is a Cocoa-based application that provides a software simulator to simulate an iOS device on your Mac OS X
- **Instruments:** These are the analysis tools that help you optimize your applications and monitor for memory leaks in real-time

The following screenshot displays a list of the various tools that are installed as part of the default settings, during the installation phase. These are installed in the / Developer/Applications folder:



The iOS Simulator

The iOS Simulator is a very useful tool that enables you to test your applications without using your actual device, whether this is your iPad or any other iOS device. You don't need to launch this application manually, as this is done when you build and run your application within the Xcode IDE. Xcode automatically installs your application on the iOS Simulator for you.

The iOS Simulator also has the capability of simulating different iOS versions, and this can become extremely useful if your application needs to be installed on different iOS platforms, as well as testing and debugging errors reported in your application when run under different versions of the iOS.

The following screenshot shows the default settings that come as part of the iOS Simulator:



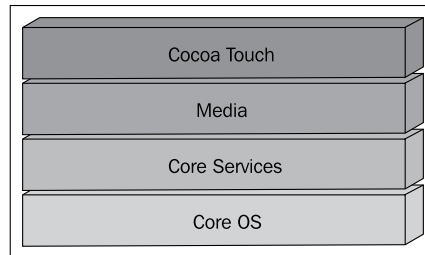
While the iOS Simulator acts as a good test bed for your applications, it is recommended to test your application on the actual device, rather than relying on the iOS Simulator for testing.

This is because the speed of the iOS Simulator relies on the performance of your Mac, instead of the actual device. The iOS Simulator application can be found at the following location: `/Developer/Platforms/iPhoneSimulator.Platform/Developer/Applications`.

Layers of the iOS architecture

Apple delivers most of its system interfaces in special packages called **frameworks**. Using frameworks allows you to link these system interfaces into your application project just as you would in any other shared library. Linking them to your project gives you access to the features of the framework, and also lets the development tools know where to find the header files and other framework resources.

Apple describes the set of frameworks and technologies that are currently implemented within the iOS operating system as a series of layers. Each of these layers is made up of a variety of different frameworks that can be used and incorporated into your applications.



We will now go into detail and explain each of the different layers of the iOS architecture. This will give you a better understanding of what is covered within each of the Core layers.

The Core OS layer

The **Core OS** layer is the bottom layer of the hierarchy, and is responsible for the foundation of the operating system that the other layers sit on top of.

This important layer is in charge of managing the memory – allocating and releasing memory once the application has finished with using it, taking care of file system tasks, handling networking, and other operating system tasks, as well as interacting directly with the hardware.

The Core OS layer consists of the following components:

Component name	Description
OS X Kernel	It is based on Mach 3.0, and is responsible for every aspect of the operating system.
Mach 3.0	It is a subset of the OS X Kernel, and is responsible for running applications within a separate process.
Berkeley Standard Distribution (BSD)	It is based on the Kernel environment within the Mac OS X, and is responsible for managing the the drivers and low-level UNIX interfaces of the operating system.
Sockets	It is a part of the CFNetwork framework, and is responsible for providing access to the BSD sockets, HTTP, and FTP protocol requests.

Component name	Description
Security	The Security framework provides functions for performing cryptographic functions (encrypting/decrypting the data). This includes interacting with the iPhone keychain to add, delete, and modify items.
Power management	It conserves power by shutting down any hardware features that are not being used currently.
Keychain	It is a part of the Security framework, and is responsible for handling and securing data.
Certificates	It is a part of the Security framework, and is responsible for handling and securing data.
File system	The system framework gives developers an access to a subset of the typical tools they would find in an unrestricted UNIX development environment.
Bonjour	It is a part of the CFNetwork framework, and is responsible for providing access to the BSD sockets, HTTP, and FTP protocol requests, and Bonjour discovery over a local-area-network.



For more information on the iOS Core OS layer, please refer to the Apple Developer Connection website at the following link: http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html#//apple_ref/doc/uid/TP40007898-CH11-SW1.

The Core Services layer

The **Core Services** layer provides an abstraction over the services provided in the Core OS layer. It provides fundamental access to the iOS services. The Core Services layer consists of the following components:

Component name	Description
Collections	It is the part of the Core Foundation framework that provides basic data management and service features for iOS applications.
Address book	It provides access to the user's Address Book contacts on the iOS device.
Networking	This is part of the System Configuration framework, which determines network availability and state on an iOS device.

Component name	Description
File access	It provides access to lower-level operating system services.
SQLite	This lets you embed a lightweight SQL database into your application without running a separate remote database server process.
Core data	This framework is provided to ease the creation of data modeling and storage in applications based on Model-View-Controller (MVC). Use of the Core Data framework significantly reduces the amount of code that needs to be written to perform common tasks when working with structured data in an application.
Core location	It is used for determining the location and orientation of an iOS device.
Net services	It is part of the System Configuration that determines whether a Wi-Fi or cellular connection is in use and whether a particular host server can be accessed.
Threading	It is part of the Core Foundation framework that provides basic data management and service features for iOS applications.
Preferences	It is part of the Foundation framework that provides the foundation classes for Objective-C, such as NSObject, basic data types, operating system services, and so on.
URL utilities	Part of the Foundation Framework that provides the foundation classes for Objective-C, such as NSObject, basic data types, operating system services, and so on.



For more information on the iOS Core Services layer, please refer to the Apple Developer Connection website at the following location:
http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html#//apple_ref/doc/uid/TP40007898-CH10-SW5.

The Media layer

The **Media** layer provides multimedia services that you can integrate and use within each of your iOS devices. The Media layer is made up of the following components:

Component name	Description
Core audio	It handles the playback and recording of audio files and streams, and also provides access to the device's built-in audio processing units.
OpenGL	It is used for creating 2D and 3D animations.
Audio mixing	It is a part of the Core Audio framework, and provides the possibility to mix system announcements with background audio.
Audio recording	It provides the ability to record sound on the iPhone using the AVAudioRecorder class.
Video playback	It provides the ability to playback a video using the MPMoviePlayerController class.
Image formats: JPG, PNG, and TIFF	It provides interfaces for reading and writing most of the image formats – part of the Image I/O framework.
PDF	It provides a sophisticated text layout and rendering engine.
Quartz	This framework is used for image and video processing, and animation using the Core Animation technology.
Core animations	It provides advanced support for animating views and other content. This is part of the Quartz framework.
OpenGL ES	This is a subset of the OpenGL framework for creating 2D and 3D animations.



For more information on the iOS Media layer, refer to the Apple Developer Connection website at the following link: http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html#//apple_ref/doc/uid/TP40007898-CH9-SW4.

The Cocoa-Touch layer

The **Cocoa-Touch** layer provides an abstraction layer to expose the various libraries for programming each of the different iOS devices. You can probably understand why Cocoa-Touch is located at the top of the hierarchy due to its support for Multi-Touch capabilities. The Cocoa-Touch layer is made up of the following components:

Component name	Description
Multi-touch events	These are the events, which are used to determine when a tap, swipe, pinch, double-tap has happened; that is, TouchesMoved, TouchesBegan, and TouchedEnded.
Multi-touch controls	It is based on the Multi-Touch model, and determines when a user has placed one or more fingers touching the screen before responding to the action accordingly.
View hierarchy	It deals with the MVC and the objects within the view.
Alerts	Using the UIAlertView class, these are used to communicate with the user when an error arises, or to request further input.
People picker	It is based on the Address Book framework, and displays the person's contact details.
Controllers	It is based on the MVC for presenting standard system interfaces and to provide much of the logic needed to manage basic application behaviors. For example, managing the reorientation of views in response to device orientation changes.
Accelerometer/gyroscope	It responds to motion and measures the degree of acceleration, and rate of rotation around a particular axis.
Localization/geographical	It adds maps and satellite images to location-based apps, similar to the one provided by the Maps application.
Web views	It provides a view to embed the web content and display rich HTML.
Image picker	It provides a potentially multidimensional user-interface element, consisting of rows and components.



For more information on the iOS Cocoa-Touch Layer, refer to the Apple Developer Connection website at the following link: http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html#//apple_ref/doc/uid/TP40007898-CH3-SW1.

Building the HelloWorld application

Before we can proceed with creating our Hello World application, we must first launch the Xcode 4.2 development environment. Double-click on the Xcode icon located in the /Developer/Applications folder.

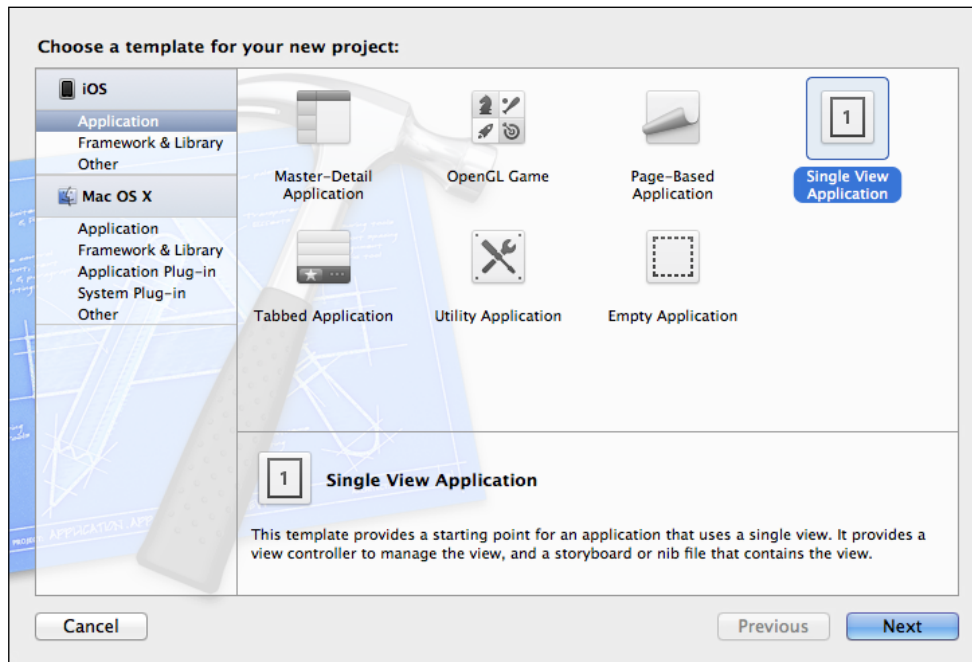
Alternatively, you can use Spotlight to search for this: simply type `xcode` into the **Search** box and Xcode should be displayed in the list at the top. When Xcode is launched, you should see the **Welcome to Xcode** screen, as shown in the following screenshot.

Since we will be creating a variety of different Xcode applications, it may be worth docking the Xcode icon to your Mac OS X launch bar for each access, as we will be using it a lot throughout this book.

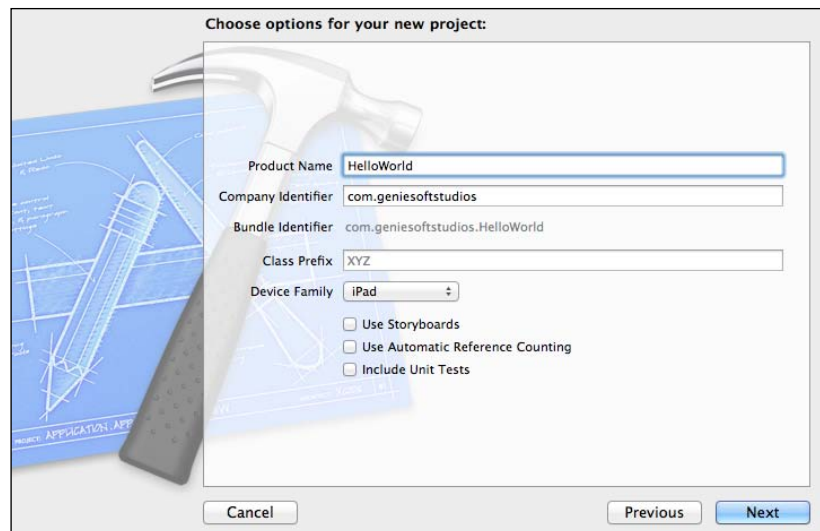


It is very simple to create our application in Xcode. Just follow the steps listed here:

1. Launch Xcode from the `/Developer/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the **Single View Application** template from the **Project** template dialog box:



4. Select **iPad** from under the **Device Family** drop-down.
5. Ensure that the **Use Storyboards** checkbox has not been checked.
6. Ensure that the **Use Automatic Reference Counting** checkbox has not been checked.
7. Ensure that the **Include Unit Tests** checkbox has not been checked.
8. Click on the **Next** button to proceed to the next step in the wizard:

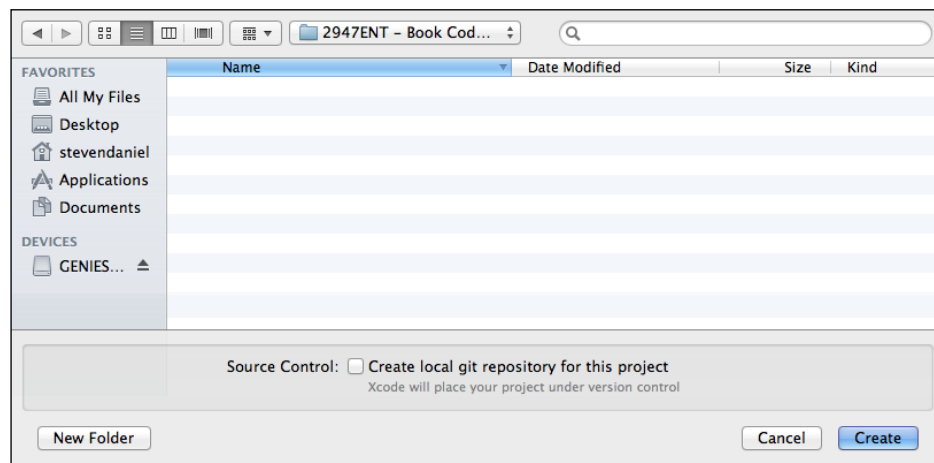


9. Enter in `HelloWorld` as the name for your project, and click on the **Next** button to proceed to the next step of the wizard.



The company identifier for your app needs to be unique. Apple recommends that you use the reverse-domain style (for example, `com.DomainName.AppName`).

10. Specify a location where you would like to save your project:



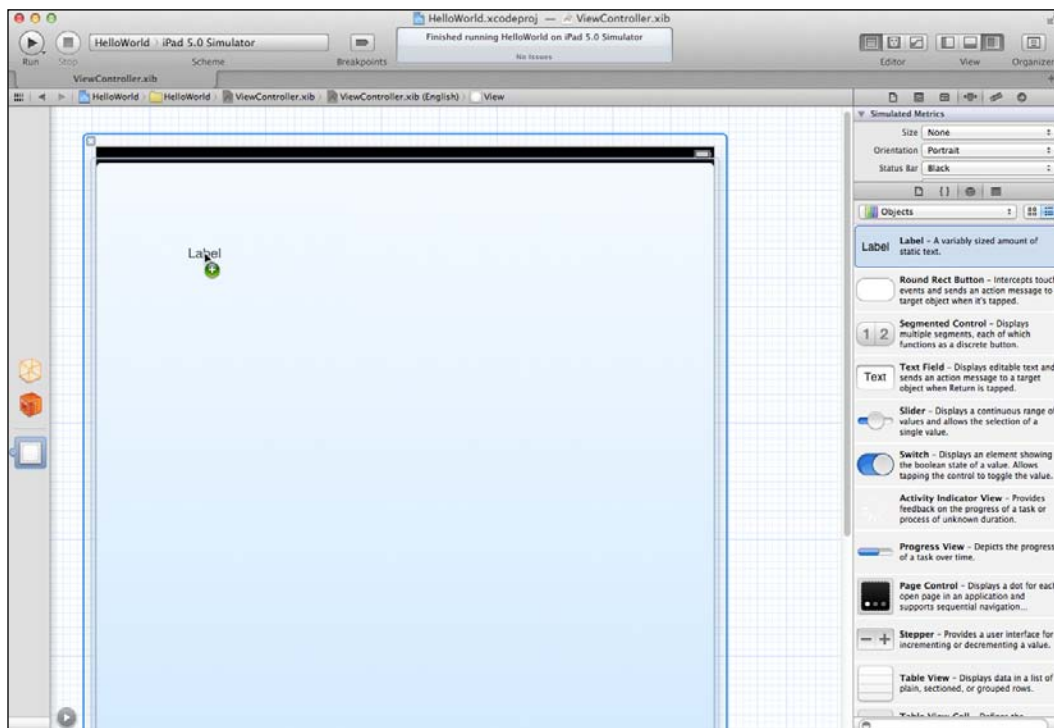
11. Then, click on the **Create** button to save your project at the location specified.

Once your project has been created, you will be presented with the Xcode development interface, along with the project files that the template has created for you, within the **Project Navigator** window.

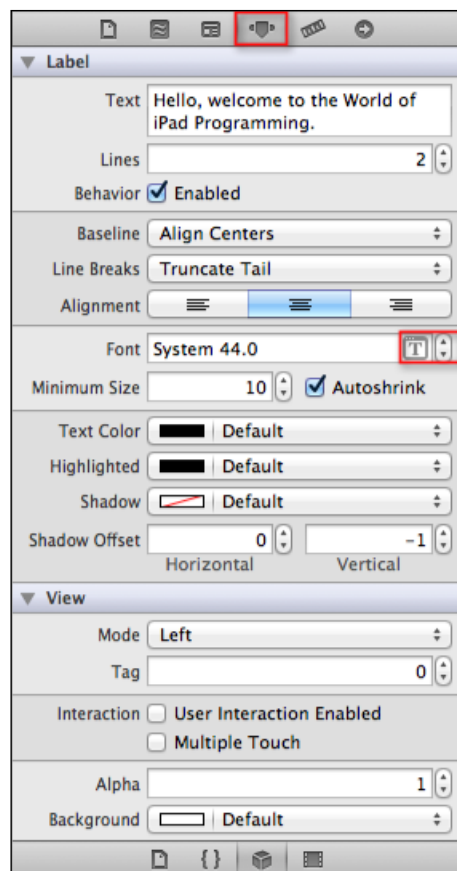
Placing objects within the View

We will now start to build the user interface for our HelloWorld application, using the controls from the Xcode Object Library and changing some of the components properties:

1. From the **Project Navigator** window, select the `ViewController.xib` file. This will display a blank canvas, which will be used for our control placement.
2. Next, simply drag the (UILabel) **Label** item from the **Object Library** onto our view, and resize accordingly, as shown in the following screenshot:



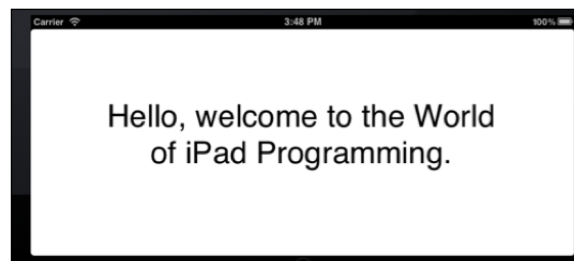
3. Once the label has been added, select this control (using the mouse) and click on the **Object Attributes** button.
4. You will notice that the **Object Attributes** properties pane is displayed and contains various other properties associated with this particular control. You will be able to change the **Text Color** and **Font size** properties of the label, as well as the alignment of the control to position it where you would like it to be situated within the view.
5. Next, change the **Label** text to Hello, welcome to the World of iPad Programming.
6. Then, change the **Lines** property to read 2.
7. Next, set the **Alignment** property to be centered.
8. Finally, change the **Font** properties size to read 44.0, by clicking on the icon to the right, as shown in the following screenshot:



If you have followed the steps correctly, your view should look something similar to the one shown in the following screenshot. Feel free to adjust yours accordingly if it doesn't.



9. Finally, run your application by choosing **Product | Run** from the **Product** menu, or, alternatively pressing *Command+R* to see your changes applied with the HelloWorld application running within the iOS Simulator.



In this section, we learned how to add a `UILabel` label control to our `ViewController` from the Xcode Object library, and manipulate the control properties required for our HelloWorld application.

Removing the Xcode Developer Tools

Should you ever wish to uninstall Xcode (in the event that something went disastrously wrong), it is a very straightforward process. Open an instance of the Terminal window and run the `uninstall-devtools` script, as follows:

```
sudo <Xcode>/Library/uninstall-devtools --mode=all
```

<Xcode> is the directory where the tools are installed. For typical installations, the full path is `/Developer/Library/uninstall-devtools`.



This process applies to installations of the Xcode developer tools, running under Mac OS X Snow Leopard. Before proceeding, please make sure this is what you really intend to do, as once it's gone, it's permanently deleted. In any event, you can always choose to reinstall the Xcode developer tools. It is worth checking that the `/Developer/Library/Xcode/` folder has also been removed. If not, just move it to the Trash.

Summary

In this chapter, we learned about the layers and components of the iOS architecture, as well as the capabilities of the iOS simulator. We also downloaded and installed the iOS 5 SDK and familiarized ourselves with some of the Xcode development tools, before taking a look at how to create a simple HelloWorld application.

In the next chapter, we will get stuck right in to some more complex applications, and look at how to create a task priorities application using the **Storyboard** feature, as well as incorporating the use of `TableViews`.

2

Task Priorities – Building a TaskPriorities iOS App

TaskPriorities is a small application that provides us with the ability to record a list of everyday tasks that need to be attended to. It will record the name of the task and its priority, a brief description of the task, and when it is due to be completed by.

In this chapter, we will be taking a closer look at how we can use the exciting new Storyboard Editor that comes as part of iOS 5, and integrate it into the Xcode 4 IDE, which will enable us to develop our applications quickly, using the least amount of code.

We will start by designing our user interfaces, then create and set up the relationships so that we can navigate between each of the different view controllers within the Storyboard, through the use of segues (pronounced **segway**). We will also create our own `UITableViewController` delegates, so that we can pass back and forward information pertaining to our tasks, for them to be added to each of our table views.

In this chapter we will:

- Build the TaskPriorities application using Storyboards
- Learn how to navigate between each View controller using Storyboards
- Learn how to handle screen orientations when the device is rotated
- Implement the functionality to add an item to `UITableView`
- Implement methods to save and delete items
- Implement methods to refresh the table view items

We have an exciting project ahead of us, so let's get started.

Building the TaskPriorities app

Creating tasks and prioritizing them is one of the most common things that we do in our everyday lives. Project managers do these as part of their job when allocating and prioritizing the urgency of the tasks that are assigned to each member of the team.

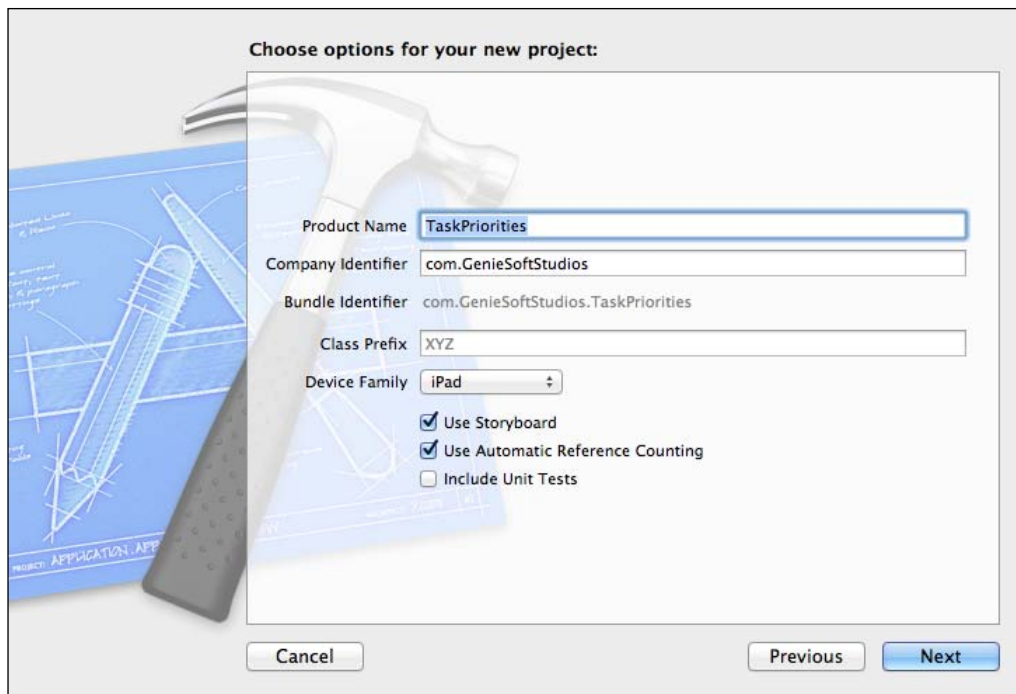
In this section, we will take a look at how to create an application that we can use to run on an iOS device, enabling us to create tasks, set their order of priority, provide them with a brief description, and then assign them with a date when they are due.

We will also write this information into a `UITableView` control, and provide the functionality to delete items that have been previously added to the list.

Before we can proceed, we first need to create our `TaskPriorities` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 1, Getting and Installing the iOS SDK*, under the section named *Building the HelloWorld application*.

It is very simple to create this in Xcode; just follow the steps listed here:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project** or **File | New Project**.
3. Select the **Single View Application** template from the list of available templates.
4. Select **iPad** from under the **Device Family** drop-down.
5. Select the **Use Storyboard** checkbox.
6. Select the **Use Automatic Reference Counting** checkbox.
7. Ensure that the **Include Unit Tests** checkbox has not been selected.
8. Click on the **Next** button to proceed to the next step in the wizard:



9. Enter in `TaskPriorities` as the name for your project, and click on the **Next** button to proceed to the next step of the wizard.
10. Specify the location where you would like to save your project.
11. Then, click on the **Save** button to continue and display the Xcode workspace environment.

Now that we have created our `TaskPriorities` project, we need to add the **Core Graphics** framework to our project. This will enable us to perform the various transition effects between our various view controllers when using Storyboards.

Adding the required frameworks

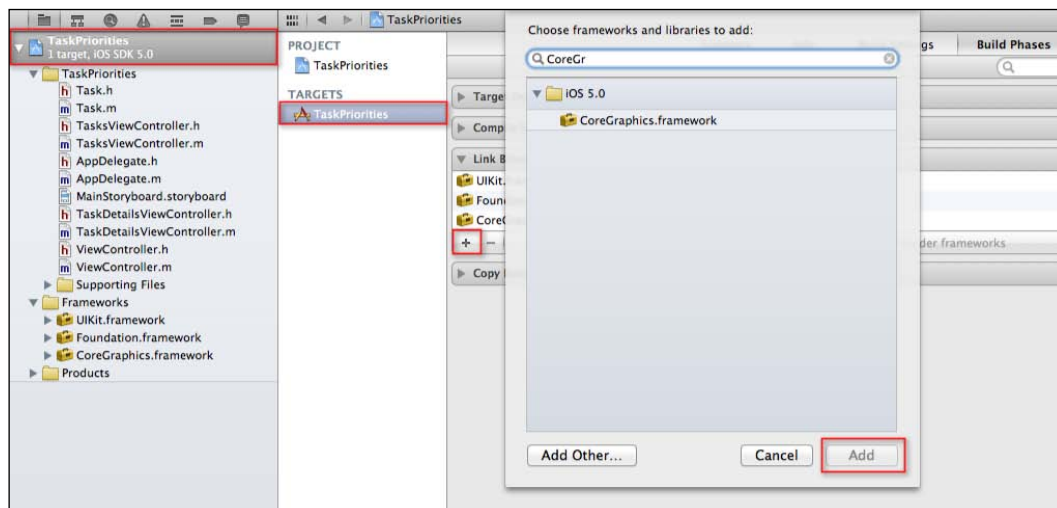
As we mentioned previously, we need to add the Core Graphics framework to our project that will enable us to perform these special effects. To add the Core Graphics framework, select **Project Navigator Group**, and follow these simple steps as outlined here:

1. Click and select your project from **Project Navigator**.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Binary With Libraries** disclosure triangle.
5. Finally, use the **+** button to add the library you want.
6. Select the **CoreGraphics.framework** from the list of available frameworks.



If you can't find the framework you are looking for, there is an added ability to search for this directly, right from within the list of available frameworks.

If you are still confused how to go about adding the framework, follow this screenshot, which highlights the areas that you need to select (surrounded by a rectangle):



Now that we have added **CoreGraphics.framework** into our project, we need to start building our user interface, which will be responsible for allowing us to create and add new tasks directly into our list.

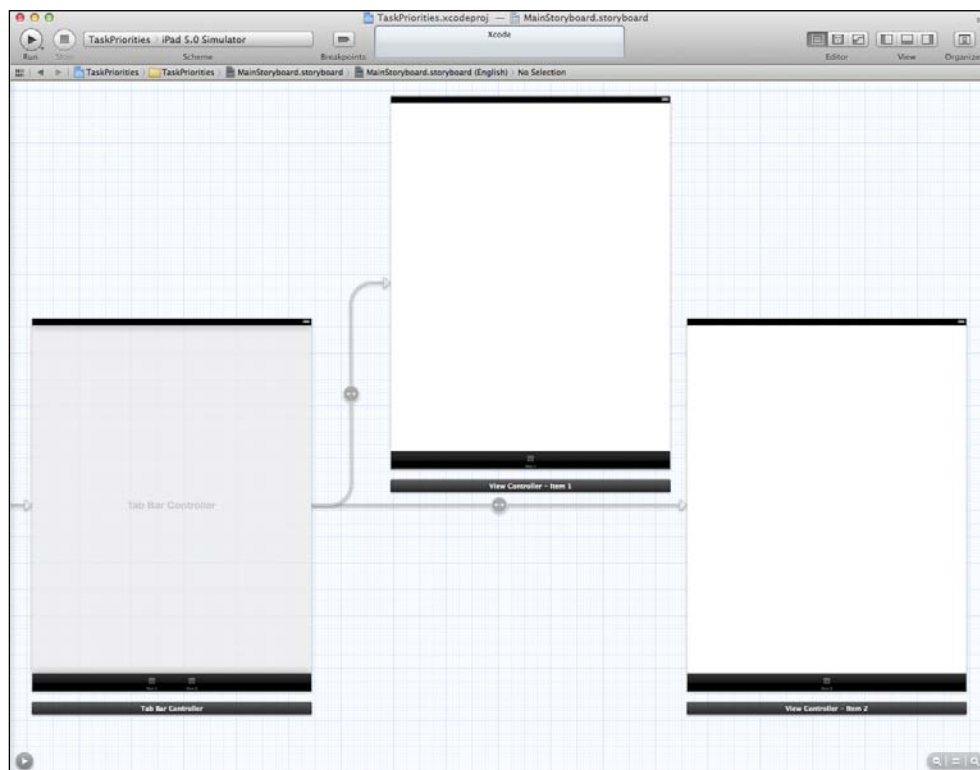
Creating the main application screen

Now that we have created our `TaskPriorities` project, we can start building our user interface using Storyboards, which will be responsible for allowing us to create new tasks and having them stored within a table view.

The screens will consist of a Tab Bar controller, Navigational controller, and View controllers. The **Navigational controller** enables us to create relationships between the other screens within the Storyboard and set up the required connections, known as segues. A **segue** represents a transition from one screen to another.

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Select the **View Controller** control that was originally added by the template, and then delete it.
3. From **Object Library**, select-and-drag a (UITabBarController) **Tab Bar Controller** control, and add this to our view.

If you have followed the steps correctly, your Storyboard should look similar to the following screenshot; if it doesn't look quite the same as mine, feel free to adjust yours:



The **Tab Bar controller** comes pre-configured with two other view controllers, one for each tab that is represented by each image button at the bottom of the **Tab Bar Controller** control. The container relationship between each screen is represented within the Storyboard editor between the Tab Bar controller and the View controllers that it contains.

Handling multiple screen orientations when the device is rotated

There may be times when you may want to have your application content displayed in various views when the iOS device rotates, providing a fast and natural feel.

When designing your iOS applications, think about how the user will be interacting with the *TaskPriorities* application. For example, will the application provide support for both portrait and landscape views, or will it just support portrait mode? Apple states, in their *Human Interface Guidelines*, that the content must be viewable in both portrait and landscape orientations.

Fortunately, this process of having your application support these different views of rotation is quite painless. To allow your application's user interface to rotate and resize into another view, you will need to add just a single method.

When the iOS device wants to check to see whether it should rotate your interface, it sends the `shouldAutorotateToInterfaceOrientation:` message to your view controller, along with the parameter that indicates which orientation it needs to check.

Your implementation of `shouldAutorotateToInterfaceOrientation:` should compare the incoming parameter against the different orientation constants contained within the iOS, by either returning `TRUE` (or `YES`) if you want to support that orientation.

The four basic screen orientations are described in the following table:

Orientation method	iOS orientation constant
Portrait	<code>UIInterfaceOrientationPortrait</code>
Portrait upside-down	<code>UIInterfaceOrientationPortraitUpsideDown</code> (iPad only)
Landscape left	<code>UIInterfaceOrientationLandscapeLeft</code>
Landscape right	<code>UIInterfaceOrientationLandscapeRight</code>

For example, to allow your iOS interface to rotate to either the portrait or landscape left orientations, we would implement `shouldAutorotateToInterfaceOrientation:` in your `viewController.m` view controller, as follows:

```
- (BOOL) shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation) interfaceOrientation
{
    // Return TRUE based on the support orientations listed.
    Return (interfaceOrientation == UIInterfaceOrientationPortrait ||
    interfaceOrientation == UIInterfaceOrientationLandscapeLeft);
}
```

So far we have added a Tab Bar controller, consisting of two view controllers that don't provide any functionality as yet. In this section, we will start building our user interface and add the required controls that will be used to process and hold our task items.

Adding the table control to hold item data

Our next step is to add a `UITableViewController` control that will be used to hold and list our task entries. We will need to include a Navigation controller that will allow us to call the `UITableViewController` view within the Storyboard whenever a new item is to be added. Implementing `UINavigationController` is very simple, and we will take a look at how this is done in a few moments. So, what happens when a view gets displayed? Well, the navigation controller does what is known as a "push its view controller onto the stack."

What this means is that a new instance of the controller is instantiated and added to the stack, then the previous controller gets pushed further down the stack. An example of this would be to think of a set of plates being placed on top of each other.

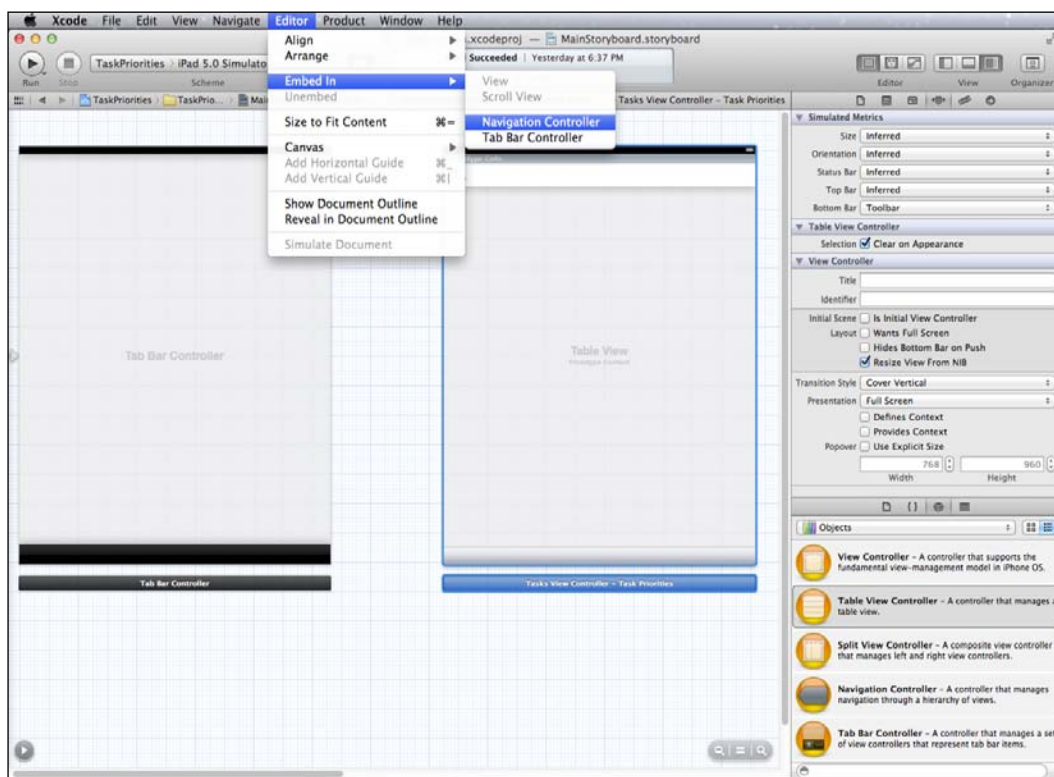
When the user decides it's time to return to the previous screen, the navigation controller pops the current view off the stack, which results in this being unloaded from memory.

The previous view controller then moves to the top of the stack and then becomes active again, hence allowing the user to navigate onto another item. The **Navigation Controller** control enables us to create relationships between the other views within the Storyboard, and set up the required connections, known as segues, which represent a transition from one screen to another.

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UITableViewController`) **Table View Controller** control, and add this to our view.

3. Next, we need to create a Navigation controller between the **Tab Bar Controller** control and `UITableViewController` that we just added. There are two ways in which this can be achieved – you can either drag `UINavigationController` directly onto the view, or you can let Xcode do this for you automatically.
4. Select `UITableViewController` that we just added, and then choose **Editor | Embed In | Navigation Controller** from the **Editor** menu.

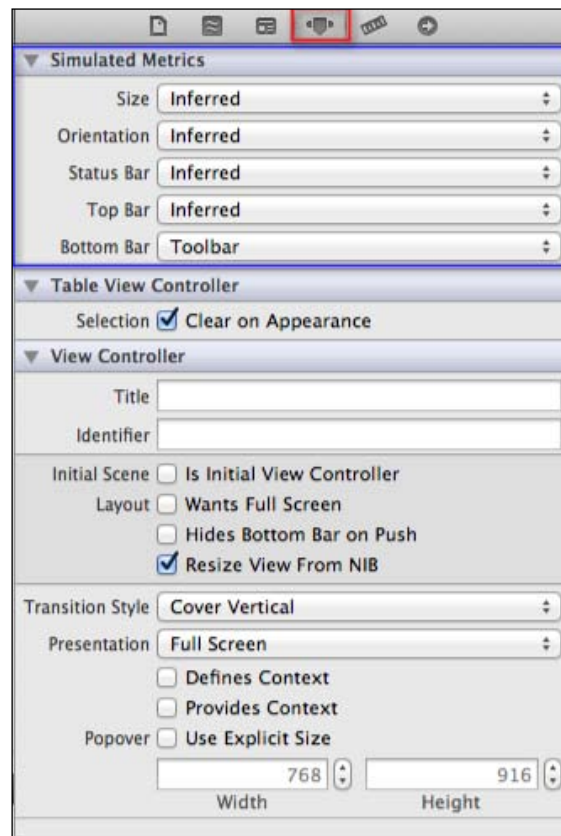
If you have followed the steps correctly, your Storyboard should look similar to the following screenshot; if it doesn't look quite the same as mine, feel free to adjust yours:



You will notice that by embedding the **Table View Controller** control, this automatically gets included within the navigation bar. The **Storyboard Editor** automatically added it in there for us, because the scene will now be displayed inside the Navigation controller's frame.



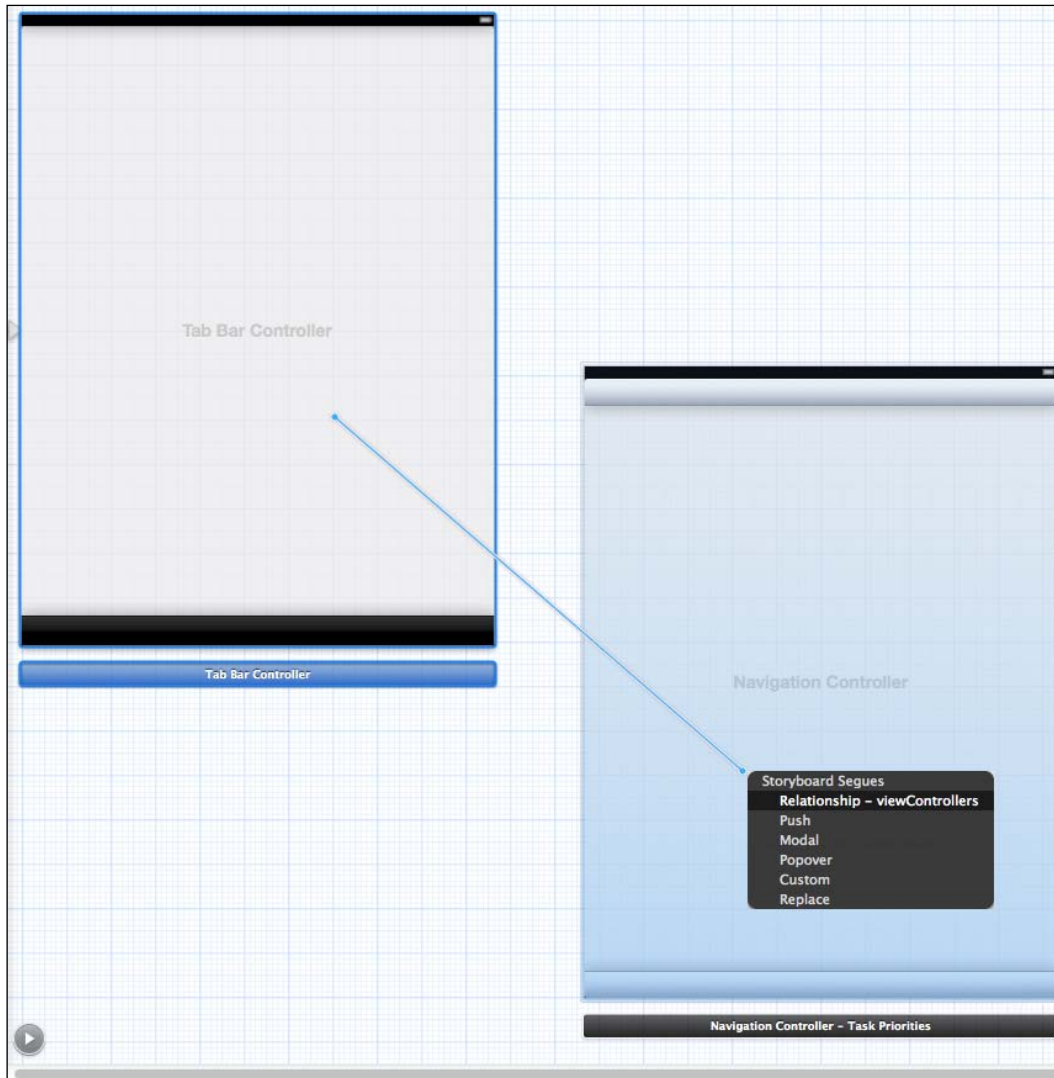
The UINavigationController screen is not a real UINavigationController object; the **Storyboard Editor** has simulated this for us. This can be seen from within the attributes inspector as shown in the following screenshot.



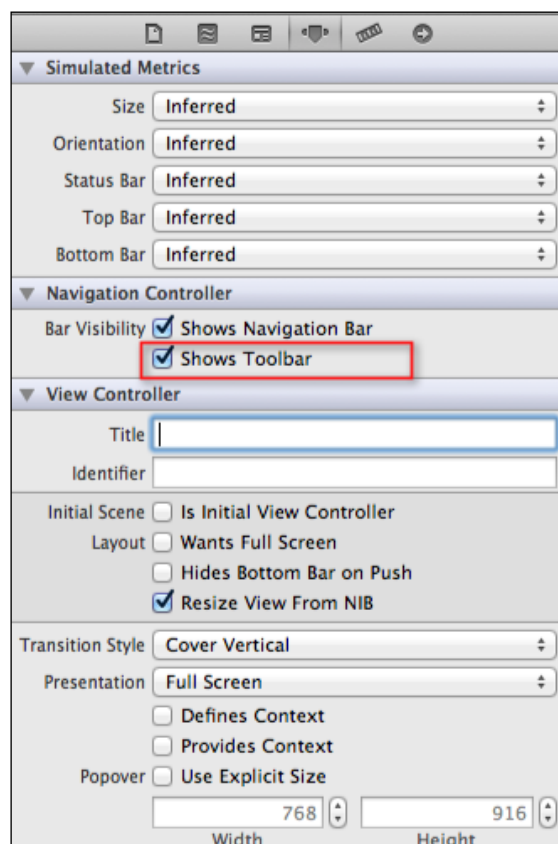
In the **Simulated Metrics** section, you will notice that **Inferred** has been set up as the default setting for each of the options; this is the default setting for storyboards. Inferred means that the scene will show a navigation bar when the **Table View Controller** control is inside a **Navigation Controller** control.

You have the ability to override any of these settings if you want to, but keep in mind that these are here only to help you when designing your screens. These aren't used during runtime, and are only available to show you how your screen will end up looking like when it is run on the iOS device.

Our next step is to connect these scenes to our **Tab Bar Controller** control, so that the **Table View Controller** control will be the first screen to be displayed when it is run:

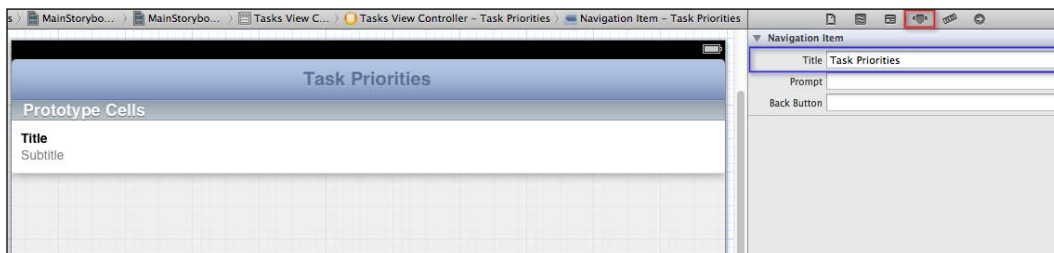


1. Select the **Tab Bar Controller** control, then hold down the *Ctrl* key, drag from **Tab Bar Controller** to **Navigation Controller**, and release the mouse.
2. Choose **Relationship – viewControllers** from the Storyboard segues pop up.
You will notice that when we made this connection between the two view controllers, a new tab was added to the **Tab Bar Controller** control, named **Item**.
3. Next, we want to show the bottom toolbar within our **Navigation Controller** control. Select **Navigation Controller**, and from the **Attributes Inspector** dialog box, tick the **Shows Toolbar** option:

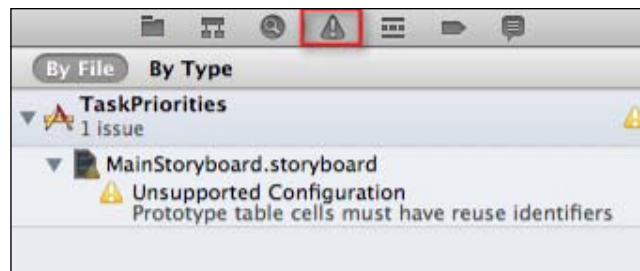


So far, we have linked up our **Tab Bar Controller** and **Navigation Controller** controls, and have configured the properties required for **Navigation Controller**; our next step is to set up the properties on our **Table View Controller** control. Follow these simple steps:

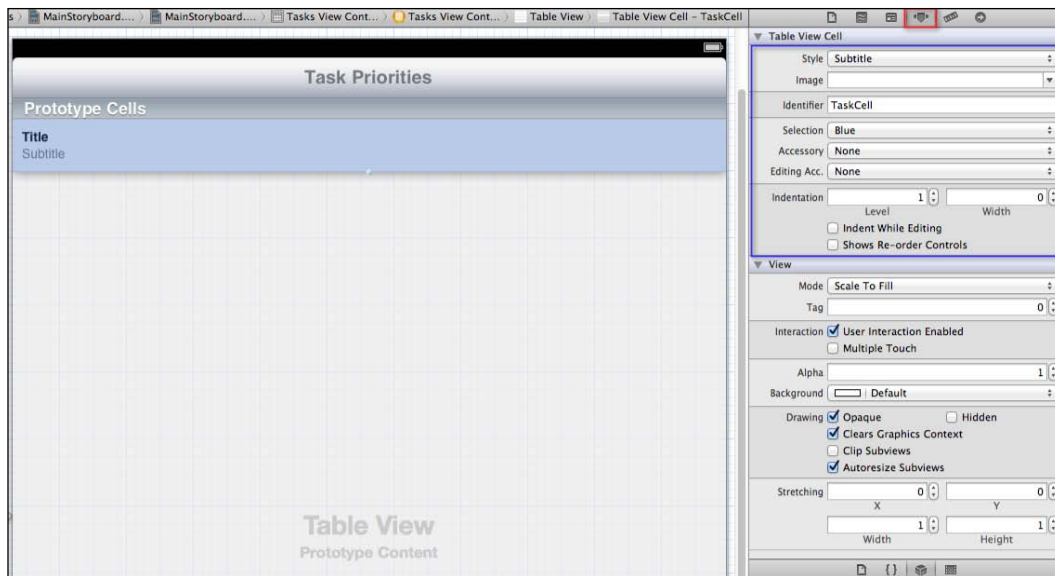
1. Select the **Table View Controller** that we just added previously.
2. Next, click on the toolbar located at the top of the **View Controller** control.
3. Then, from **Attributes Inspector**, change the value of **Title** to read Task Priorities, as shown in the following screenshot:



If you prefer, you can also double-click the navigation bar and change its title. You may have noticed that since we added our **Table View Controller** control, Xcode gave us a warning. This is shown in the following screenshot:



This warning message comes up whenever you add a **Table View Controller** control to a Storyboard, and this is because it wants to use prototype cells as the default, but we haven't correctly configured this control yet. Let's take a look at the following screenshot:

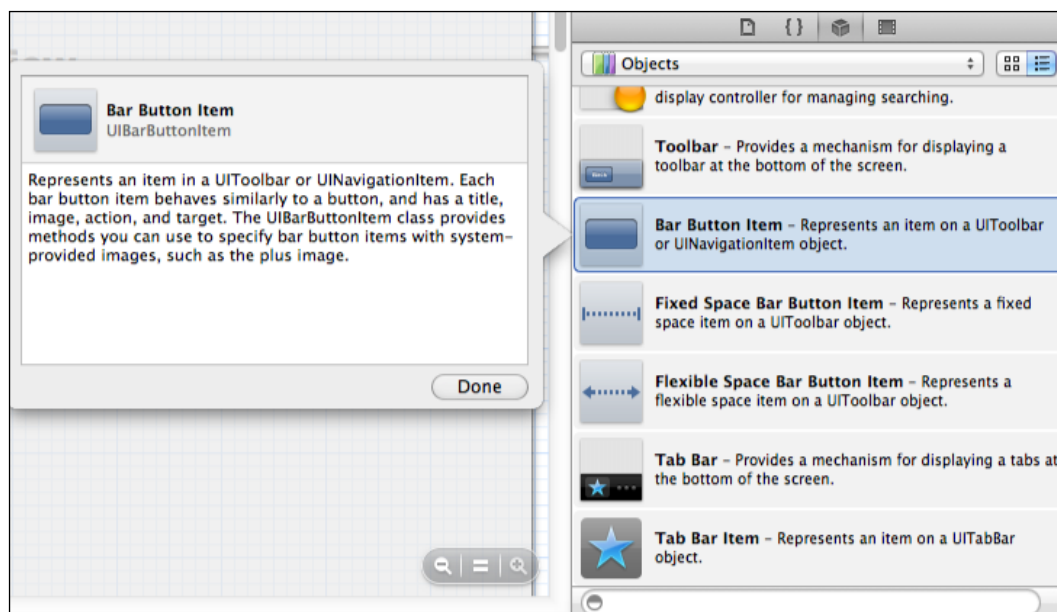


4. Click on the **Prototype** cell from the **Prototype Cells** section.
5. From the **Attributes Inspector** section, change the value of **Style** to **Subtitle**. This will change the cell's appearance to contain two labels.
6. Select the **Identifier** item and enter in `TaskCell` as its unique identifier. This is used to look up and reuse existing cells without the need of having to release and reallocate memory for new cells each time. You will notice that once this has been entered in, Xcode will stop complaining about the warning message we received earlier.
7. Set the **Accessory** attribute to show **None**.

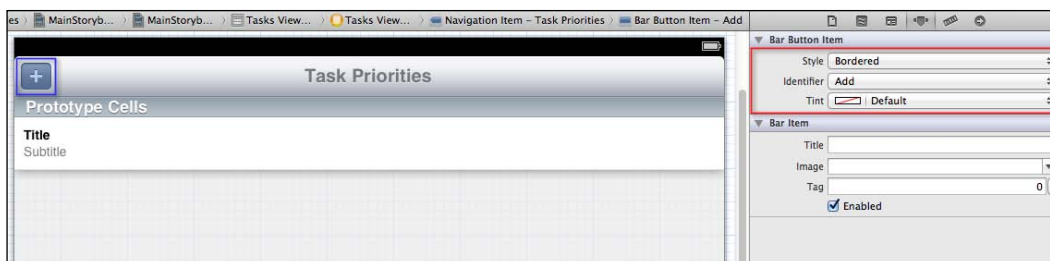
Adding the Add button

Our next step is to add a button to our `UITableViewController`; this will be responsible for displaying an additional screen where we can create additional tasks. This can be achieved by following these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top-left corner of the navigation bar (`UITableViewController`) **Table View Controller** screen of the **Task Priorities** window that we added previously:



3. From the **Attributes Inspector** section, change the **Identifier** to **Add**.
4. Then, change the value of **Style** to **Bordered**:

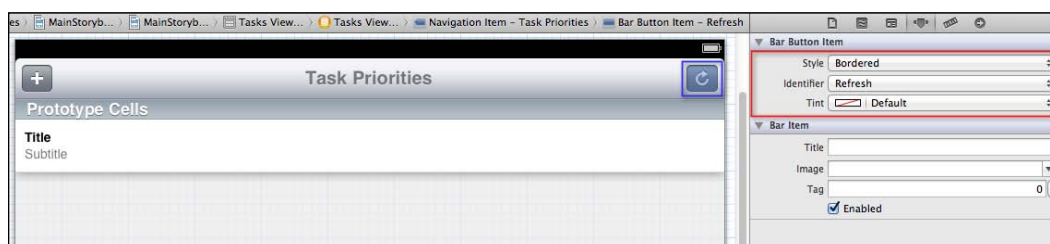


Now that we have added our **Add** button to our **Task Priorities View Controller**, our next step is to add the **Refresh** button, which will be responsible for refreshing our table view when the button is clicked. So let's proceed with the next section.

Adding the Refresh button

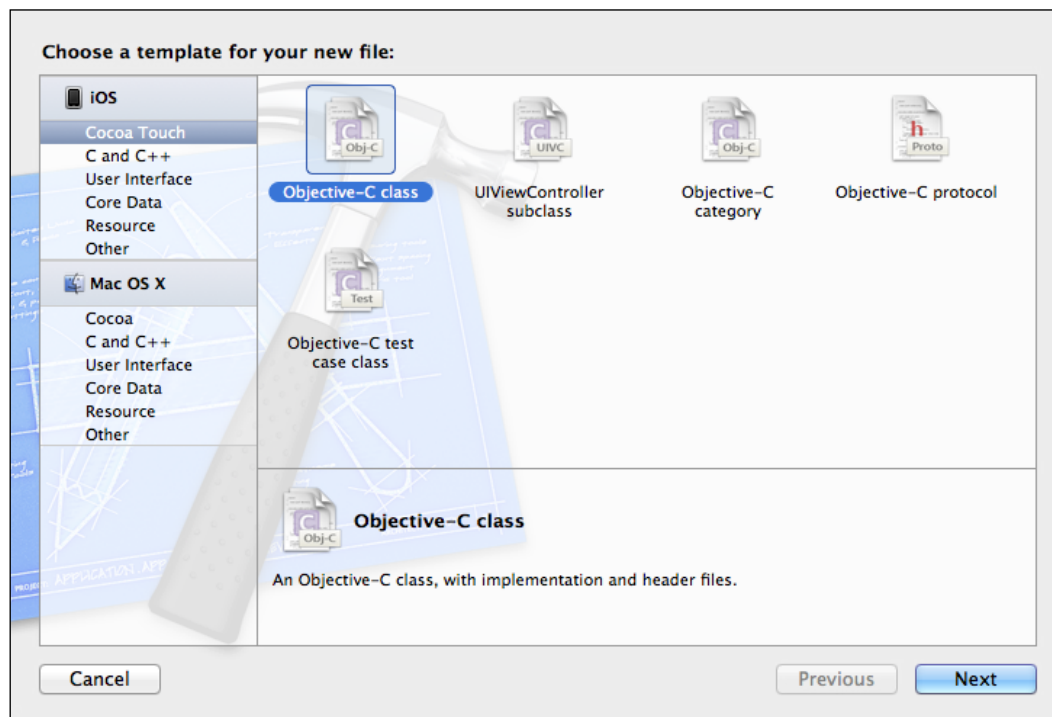
Now that we have added our button to add a new record, our next step is to add another button to our `UITableViewController`; this will be responsible for refreshing the table view. This can be achieved by following these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top-right corner of the navigation bar (`UITableViewController`) **Table View Controller** screen of the **Task Priorities** window that we added previously.
3. From the **Attributes Inspector** section, change the value of **Identifier** to **Refresh**.
4. Then, change the value of **Style** to **Bordered**:

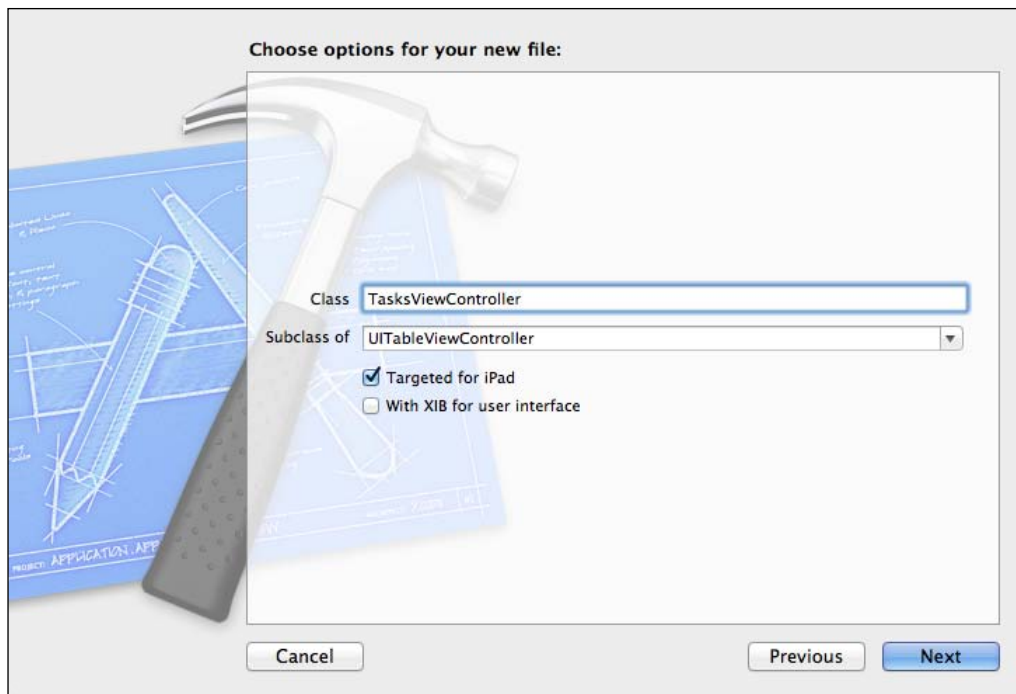


Now that we have added our **Add** and **Refresh** buttons, as well as properly configured our **Table View Controller**, our next step is to create our very own custom `UITableViewController` subclass that will act as the data source for our table, so that it will know how many rows to display:

1. Select the `TaskPriorities` folder; choose **File | New | New File...** or **Command+ N**.
2. Select the **Objective-C** class template from the list of templates:



3. Click on the **Next** button to proceed to the next step within the wizard.
4. Enter in `TasksViewController` as the name of the file to create.
5. Ensure that you have selected `UITableViewController` as the type of subclass to create from the **Subclass** of dropdown.
6. Ensure that you have selected the **Targeted for iPad** option:

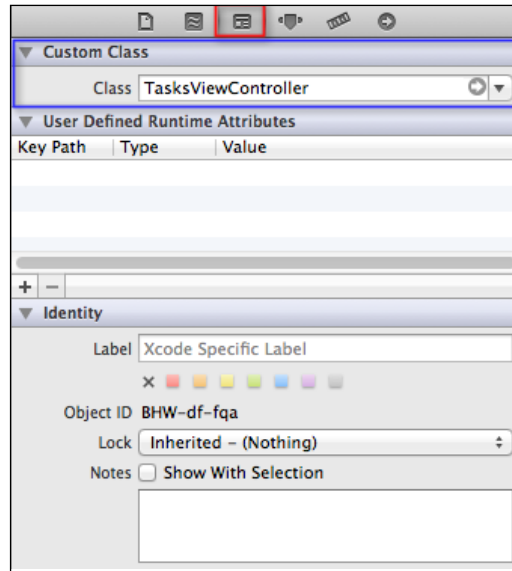


7. Click on the **Next** button to proceed with the next step of the wizard.
8. Click on the **Create** button to save the file to the folder location specified.

Now that we have added our View Controller class to our **TaskPriorities** application, our next task is to update the class of `UITableViewController` to use this class, instead of the default `UITableViewController` class:

1. Select the **MainStoryboard.storyboard** file from **Project Navigator**.
2. Click and select our **Task Priorities** (`UITableViewController`) controller.

3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `TasksViewController`:



Our next step is to create an `NSMutableArray` array property within our `TasksViewController` interface file:

1. Open the `TasksViewController.h` interface file located within the `TaskPriorities` folder, and enter in the following code snippet:

```
// TasksViewController.h
// TaskPriorities
// Created by Steven F Daniel on 30/12/11.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>

@interface TasksViewController : UITableViewController

@property (nonatomic, strong) NSMutableArray *tasks;

@end
```

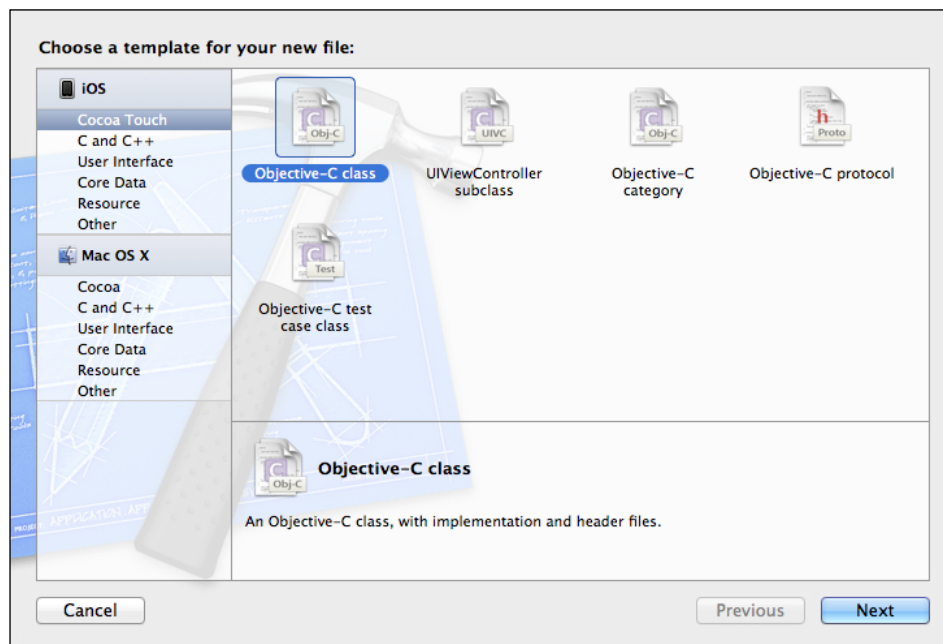
As you can see, all we have done is created an `NSMutableArray` object that will be used to hold each of our tasks objects that we will create, and this will also be used to act as a data source to our **Task Priorities** table view.



For more information about the NSMutableArray object, you can refer to the Apple Developer documentation at the following URL: https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/Reference/Reference.html.

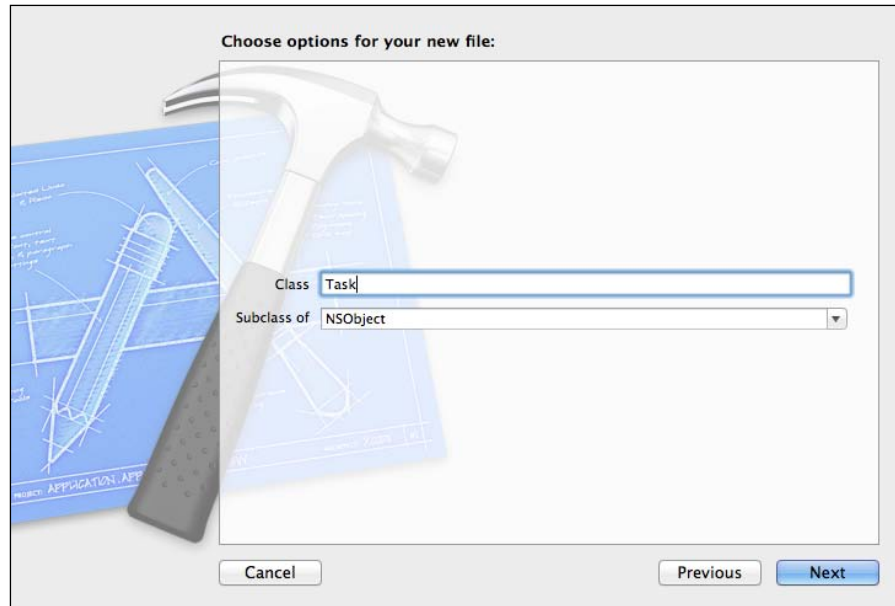
Our next step is to create the `Tasks` object class, which will act as a data object, containing the necessary properties for the task name, the priority, when it is due, and a description of the task:

1. From the `TaskPriorities` folder, choose **File | New | New File...**, or *Command + N*.
2. Select the **Objective-C** class template from the list of templates.



3. Click on the **Next** button to proceed with the next step within the wizard.
4. Enter in the value of **Task** as the name of the file to create.

5. Ensure that you have selected `NSObject` as the type of subclass to create from the **Subclass of** drop-down:



The `NSObject` class provides us with a framework for creating, initializing, de-allocating, copying, comparing, and archiving objects.

6. Click on the **Next** button to proceed with the next step of the wizard.
7. Click on the **Create** button to save the file to the folder location specified.
8. Next, open the `Task.h` interface file located within the `TaskPriorities` folder, and enter in the following code snippet:

```
// Task.h
// TaskPriorities
// Created by Steven F Daniel on 30/12/11.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

@interface Task : NSObject

@property (nonatomic, copy) NSString *taskName;
@property (nonatomic, copy) NSString *description;
@property (nonatomic, copy) NSString *priority;
@property (nonatomic, copy) NSString *dueDate;

@end
```

What we have created in our `Task.h` interface file is an object class that will be used to store each task and the information associated with it. This will make it easier when we want to pass this information back and forward, between the various screens in our Storyboard.

9. Next, open the `Task.m` implementation file located within the `TaskPriorities` folder. Enter in the following highlighted code snippet:

```
// Task.m
// TaskPriorities
// Created by Steven F Daniel on 30/12/11.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "Task.h"

@implementation Task

@synthesize taskName;
@synthesize description;
@synthesize priority;
@synthesize dueDate;

@end
```

We have added the preceding code snippet to make the outlet properties visible, so that we can start using them when we store these into our array from within our `TasksViewController` implementation file.

10. Next, we need to modify the `viewDidLoad` method located within the `TasksViewController.m` implementation file. Enter in the following highlighted code snippet:

```
#pragma mark - View lifecycle
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Initialize and reload our Tasks Information
    [self initTasksDetails];
}
```

In this code snippet, we need to call the `initTasksDetails` method that will be responsible for populating our table view with default task information.

11. Next, open the `TasksViewController.m` implementation file, and enter in the following code snippet for the `initTasksDetails` method, as follows:

```
#pragma mark Populate our UITableView Controller with default Task
Details.
- (void) initTasksDetails
{
    tasks = [NSMutableArray arrayWithCapacity:99];
    Task *task = [[Task alloc] init];

    task.taskName = @"Build the GUI Screen";
    task.description = @"Create Main Application GUI Screen.";
    task.priority = @"Medium";
    task.dueDate = @"22/12/11";
    [tasks addObject:task];
    task = [[Task alloc] init];
    task.taskName = @"Add the Save Record button";
    task.description = @"Implement the functionality to save the
record to the grid.";
    task.priority = @"High";
    task.dueDate = @"3/01/12";
    [tasks addObject:task];
    task = [[Task alloc] init];
    task.taskName = @"Add the Delete Record button";
    task.description = @"Implement the functionality to Delete
the record from the grid.";
    task.priority = @"Low";
    task.dueDate = @"22/03/12";
    [tasks addObject:task];
}
```

In this code snippet, we simply declared and initialized our `NSMutableArray` `tasks` array, which is an extension of the `NSArray` object that allows us to store, remove, and modify array items. Next, we created some default tasks and add them to our `tasks` array.

12. Next, we need to change the `didFinishLaunchingWithOptions:` method located within the `AppDelegate.m` implementation file:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchin
gWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after the application
    // launches.
    return YES;
}
```

When using Storyboards we don't need to create a new `UIWindow` instance, as this will create another white window and place this on top of the Storyboard. So, we just need to clear out everything except the return `YES` statement.

Now that we have successfully populated our array with `Task` objects, our next step is to continue building our data source for `TasksViewController`. First, we need to import a reference to our `Tasks.h` interface file; otherwise, our class will not know anything about our `Task` object, or remember to synthesize the `tasks` property.

1. Open the `TasksViewController.m` implementation file, and enter in the following highlighted code snippets:

```
//
//  TasksViewController.m
//  TaskPriorities
//
//  Created by Steven F Daniel on 30/12/11.
//  Copyright (c) 2012 GenieSoft Studios. All rights reserved.
//
```

```
#import "TasksViewController.h"
#import "Task.h"
```

```
@implementation TasksViewController
@synthesize tasks;
```

2. Next, we need to change the table view data source methods that are located within the `TasksViewController.m` implementation file, and enter in the following highlighted code snippets:

```
#pragma mark - Table view data source
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection: (NSInteger)section
{
    // Return the number of rows in the section.
    return [self.tasks count];
}
```

As you can see, we initialize the number of table sections that our table will contain, and then use the `numberOfRowsInSection` method to work out how many rows need to be displayed within each section. This is achieved by using the `count` property of our `tasks` array object.

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

    Task *task = [self.tasks
    objectAtIndex:indexPath.row];

    cell.textLabel.text = [[NSString alloc]
    initWithFormat:@"Task: %@\tPriority: %@",
    task.taskName, task.priority];

    cell.detailTextLabel.text = [[NSString alloc]
    initWithFormat:@"%@\tDue on: %@", task.description,
    task.dueDate];

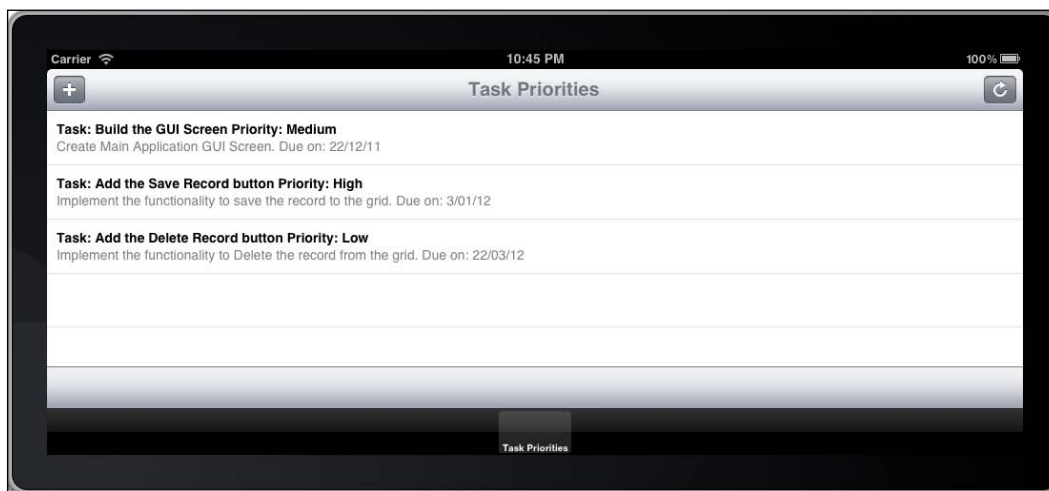
    // Configure the cell...
    return cell;
}
```

Finally, as you can see in the preceding code snippet, we supply the reuse identifier of the cell of `TableViewController` that we set up previously, then assign each of the properties from our `task` array, and write it to the cell labels.



When you reference the reuse identifier as a parameter to the following method called `dequeueReusableCellWithIdentifier`, this will automatically make a new copy of the prototype, and return the object back to you.

- Now that we have set up the data source correctly for our `TableViewController`, we can now run our application by choosing **Product | Run** from the **Product** menu, or alternatively by pressing *Command + R* to see the `TaskPriorities` application running within the iOS Simulator, as shown in the following screenshot:



Now that we have successfully configured our data source for our list of `Tasks`, we will see how we can navigate between screens within the Storyboard. We will learn about segues, and the different types of views they can take on. We will look into static table view cells, as well as how to go about providing the ability for additional tasks to be added to the `tasks` list.

Navigating between screens using Storyboards

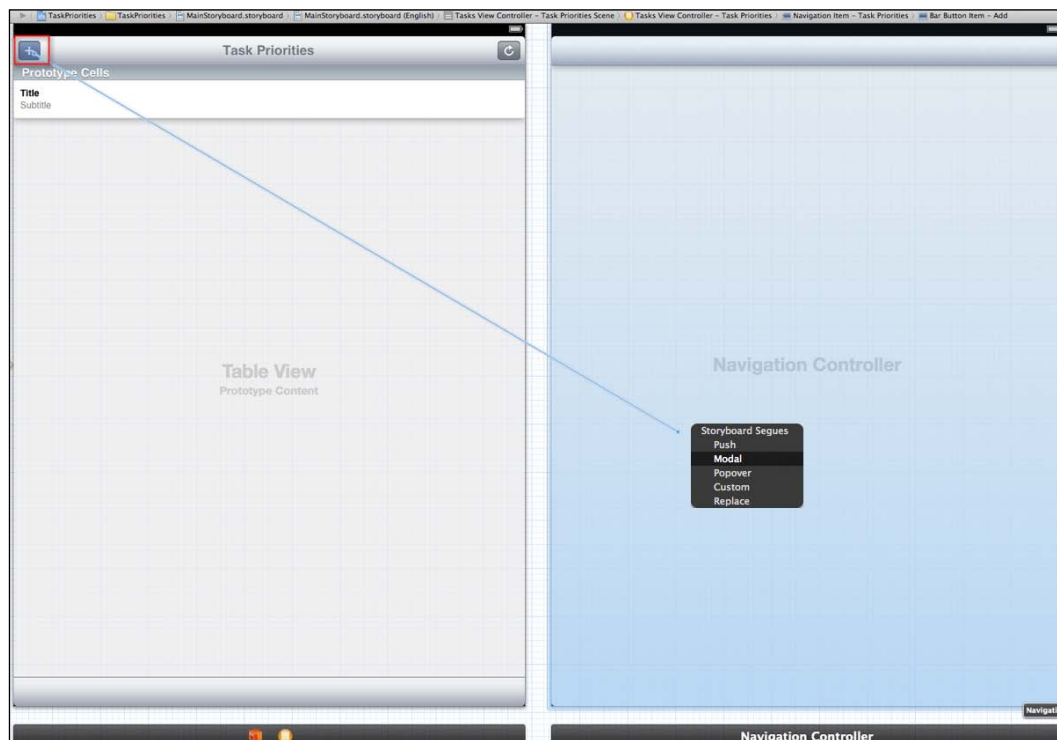
In this section, we will be adding more view controllers to our Storyboard to allow the flexibility of adding new tasks to our existing table view.

In order for us to transition between screens within our Storyboard, we need to create a connection, known as a segue. Segues are defined as having the ability to only go one way; they cannot go back to the previous screen, unless a delegate class has been set up.

For our new screen, we will be creating a "modal" segue. A **modal segue** is a screen that becomes the active screen and prevents the user from interacting with the underlying screen until they close the modal screen first.

To begin creating the **Add new tasks** screen, follow these simple steps:

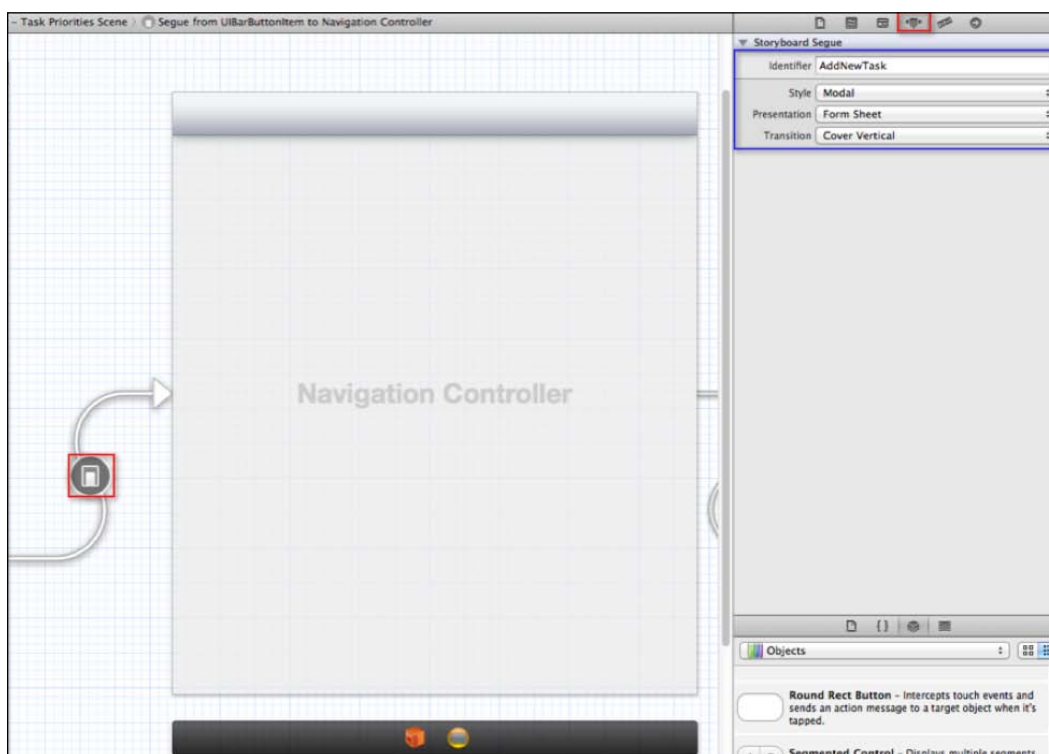
1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a new (`UITableViewController`) **Table View Controller** control, and add this to our Storyboard to the right of the **Task Priorities** screen.
3. Next, select `UITableViewController` that we just added, and then choose **Editor | Embed In | Navigation Controller** from the **Editor** menu.
4. Next, select the **+** button that we added previously, hold down the *Control* key while dragging it to the new **Navigation Controller** control, and release the mouse button.
5. Finally, select **Modal** from the pop-up list of choices:



When you select **Modal** from the list of **Storyboard Segues**, a new arrow gets placed between the **Tasks** screen and the **Navigational Controller** control. So, when you press the **+** button, a new Table View will slide up from the bottom of the screen.

Next, we need to specify an identifier for our Storyboard Segue. This will be responsible for handling the canceling and saving methods when the **Add New Task** form is closed.

1. Select the segue relationship that is located between the **Task Priorities** screen and the **Navigation Controller** control for the **Add New Task** screen.
2. Click on the **Attributes Inspector** button.
3. Change the **Identifier** property to `AddNewTask`.
4. Change the **Style** property to **Modal**.
5. Change the value of **Presentation** to **Form Sheet**.
6. Change the value of **Transition** to **Cover Vertical**:

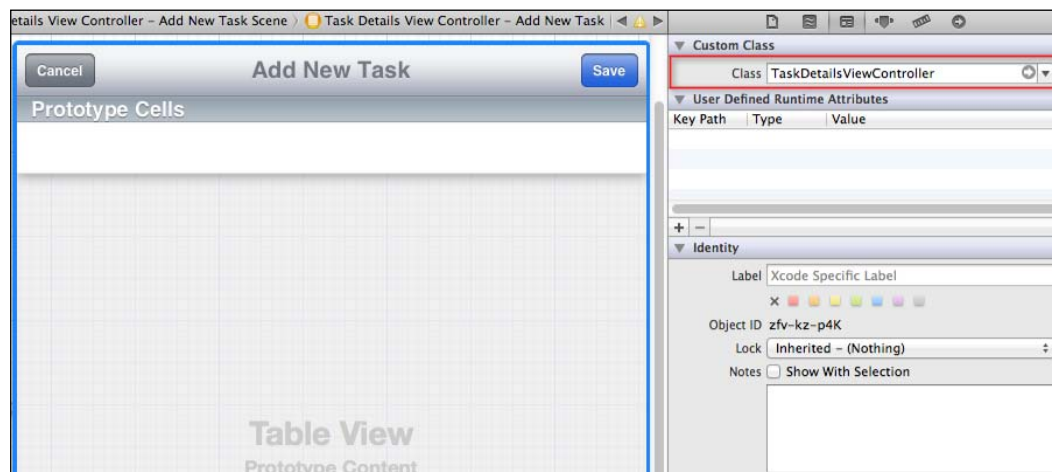


Unfortunately, you won't be able to go back to the previous screen until we create a `UITableViewController` subclass, in the same way as what we did for our `TaskViewController`.

1. From the `TaskPriorities` folder, choose **File | New | New File...** or press the **Command + N** keys.
2. Select the `UIViewController` subclass template from the list of templates.
3. Click on the **Next** button to proceed with the next step within the wizard.
4. Enter in `TaskDetailsViewController` as the name of the file to create.
5. Ensure that you have selected `UITableViewController` as the type of subclass to create from the **Subclass** of drop-down.
6. Ensure that you have selected the **Targeted for iPad** option.
7. Click on the **Next** button to proceed with the next step of the wizard.
8. Then, click on the **Create** button to save the file to the folder location specified.

Once you have done this, we need to update the class method of our previously added **Table View Controller** to use our new Table View subclass. Follow these simple steps:

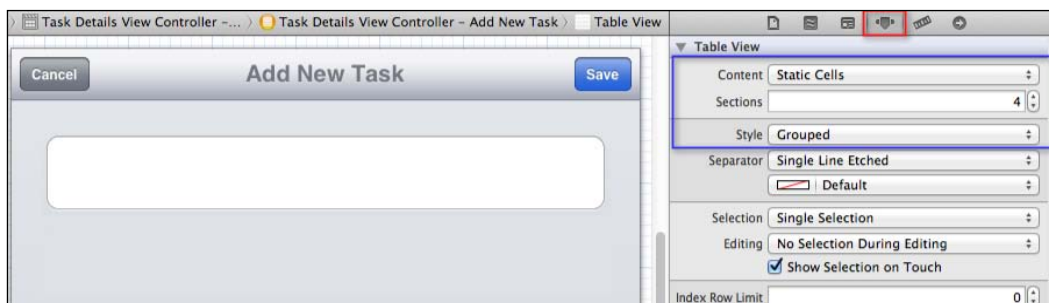
1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Click-and-select our newly added (`UITableViewController`) to the right of the **Task Priorities** table.
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `TaskDetailsViewController`.



4. Next, from the **Attributes Inspector** section, change the **Title** property to read **Add New Task**.
5. From **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control to the top-left corner of the navigation bar on the **Add New Task** (UITableViewController) window of the **Table View Controller** screen that we added previously.
6. From the **Attributes Inspector** section, change the value of **Identifier** to **Cancel**.
7. Then, change the value of **Style** to **Bordered**.
8. Next, from **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control to the top-right corner of the navigation bar on the **Add New Task** (UITableViewController) window of the **Table View Controller** screen that we added previously.
9. From the **Attributes Inspector** section, change the value of **Identifier** to **Save**.
10. Change the value of **Style** to **Bordered**.

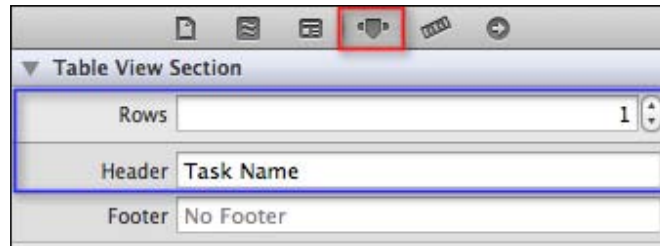
Our next step is to start building the screen that will allow us to record the task information, so that it can be saved to the `TaskPriorities` list:

1. Select the **Add New Task** table view controller from within our Storyboard.
2. Change the **Content** field property to read **Static Cells**.
3. Change the **Style** field property to read **Grouped**.
4. Modify the **Sections** property to display as **4**:

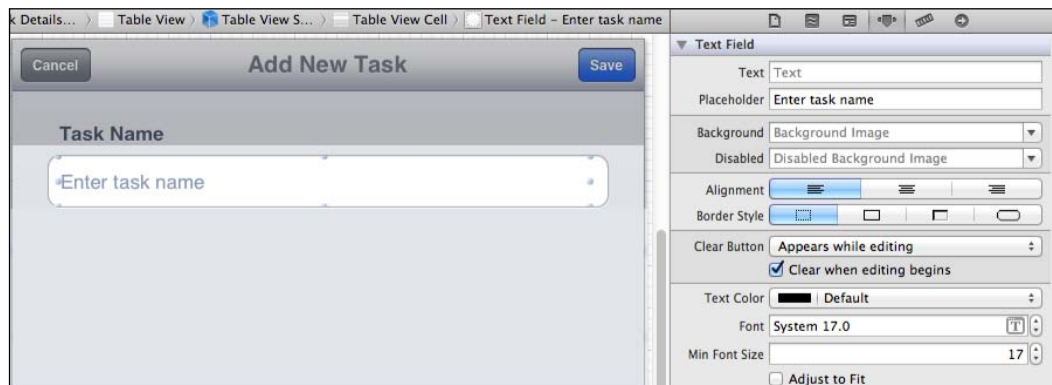


5. Select the first section, and then select **Attributes Inspector**.
6. Change the **Rows** section to **1**.

7. Change the value of **Header** to read Task Name:



8. Repeat steps 5 to 7 to apply the same for each section to add the values for **Description**, **Priority**, and **Due Date**.
9. Next, drag a (UITextField) **Text Field** control into the **Task Name** cell.
10. Select the value of **Attributes Inspector** for the **Text** field.
11. Set the **Placeholder** field property to read Enter task name.
12. Set the **Alignment** field property to left justify.
13. Set the value of **Border Style** to none.
14. Set the value of **Font** to **System 17.0**.
15. Ensure that the **Adjust to Fit** checkbox is unchecked:



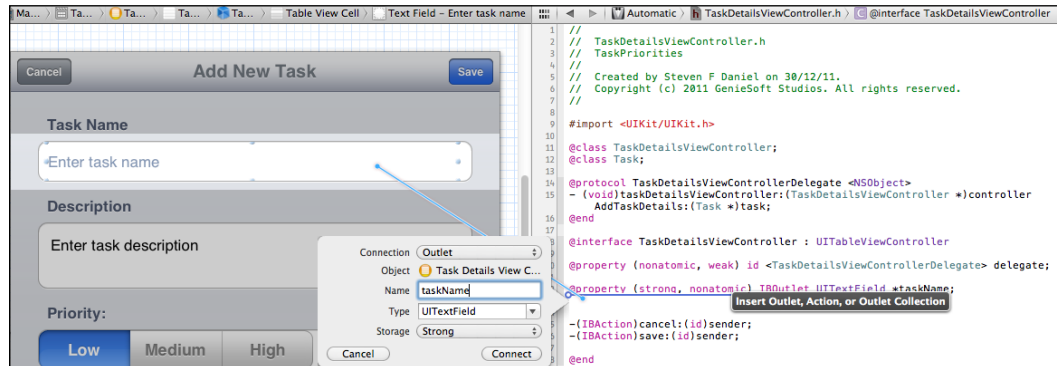
16. Repeat steps 9 to 15 to apply the same for each section and add the **Description** (UITextView), **Priority** (UISegmentedControl), and **Due Date** (UIDatePicker) values.

If you have followed the steps correctly, the completed **Add New Task** screen should look similar to the following screenshot. Feel free to adjust yours accordingly if it doesn't.

The next step is to create the outlets for the **Task Name**, **Description**, **Priority**, and **Due Date** properties:

1. Open **Assistant** Editor by choosing **Navigate | Open In Assistant Editor** or press the *Option + Command + ,* keys.
2. Ensure that the `TaskDetailsViewController.h` interface file gets displayed.
3. Select the **Task Name** (`UITextField`), hold down the *Control* key, and drag it into the `TaskDetailsViewController.h` interface file.
4. Enter in `taskName` for the **Name** of the property to be created.

5. Choose **Strong** from the **Storage** drop-down:



6. Repeat steps 3 to 5 and create the properties for the **Description**, **Priority** and **Due Date** fields.

Since we have set up our table view to use **Static Cells**, we no longer need the data source and delegate sections within our `TaskDetailsViewController.m` implementation file, so we will need to remove these. If we leave these in, it will just overwrite our controls and display a table with a series of rows, which is not what we want to happen.

1. Open the `TaskDetailsViewController.m` implementation file, located within the `TaskPriorities` folder.
2. Comment out or delete anything that is between the Table view data source and Table view delegate `#pragma mark` marks:

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

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    ..
    ..
}
```

```
...
...
```

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

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

```

        ..
        ..
    }

@end

```

Now that we have properly set up our screen that will be responsible for adding new tasks to our TaskPriorities table view, we need to start modifying our TaskDetailsViewController.h interface file:

1. Open the TaskDetailsViewController.h interface file, located within the TaskPriorities folder, and enter in the following highlighted code snippets:

```

// TaskDetailsViewController.h
// TaskPriorities
//
// Created by Steven F Daniel on 30/12/11.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>

@class TaskDetailsViewController;
@class Task;

@protocol TaskDetailsViewControllerDelegate <NSObject>
- (void)taskDetailsViewController:(TaskDetailsViewController *)
controller AddTaskDetails:(Task *)task;
@end

@interface TaskDetailsViewController : UITableViewController

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

@property (strong, nonatomic) IBOutlet UITextField *taskName;
@property (strong, nonatomic) IBOutlet UITextView *description;
@property (strong, nonatomic) IBOutlet UISegmentedControl
*priority;
@property (strong, nonatomic) IBOutlet UIDatePicker *dueDate;

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

@end

```


In the preceding code snippet, we declare a new delegate object that will allow other view controllers within the Storyboard to communicate with each other using this delegate object, and communicate back to our **Task Priorities** screen when the user either cancels or saves the **Add New Task** screen.

2. Next, we need to connect the **Cancel** and **Save** buttons to their respective action methods.
3. Click on the **Cancel** button while holding down the *Control* key and dragging your mouse to the view controller.
4. From the pop-up menu, choose the **cancel** method.
5. Repeat steps 5 to 6, hook up the **Save** button, then choose the **save** method from the pop-up menu:



In the next section, we will take a look at building the functionality for our **TaskPriorities** application, as well as implementing the methods that will be used for our **Cancel** and **Save** buttons. These will be responsible for adding new tasks to our **TaskPriorities** list and returning us back to the **Task Priorities** screen.

Implementing the Save record method

We are now ready to start implementing the method that will be responsible for saving the record when the user presses the **Save** button.

Open the `TaskDetailsViewController.m` implementation file located within the `TaskPriorities` folder, and enter in the following code snippet:

```
-(IBAction) save:(id) sender
{
    NSDateFormatter *dateFormat;

    dateFormat = [[NSDateFormatter alloc] init];
    [dateFormat setDateFormat:@"MMMM d, yyyy"];

    Task *task = [[Task alloc] init];
    task.taskName = self.taskName.text;
    task.description = self.description.text;

    task.priority = [[NSString
                     alloc] initWithFormat:@"%@", [priority
                     titleForSegmentAtIndex:self.priority.selectedSegmentIndex]];

    task.dueDate = [[NSString
                    alloc] initWithFormat:@"%@", [dateFormat
                    stringFromDate:[self.dueDate date]]];

    [self.delegate taskDetailsViewController:self
    AddTaskDetails:task];
}
```

In the preceding code snippet, we created an `NSDateFormatter` object, and used the `setDateFormat` method to set up and initialize the correct date format that we would like our **Due Date** field to be in. We then created a new `Task` object, and then assigned the object properties with the values from our **Add New Task** screen. Finally, we notified the delegate object that we have added a new task item, so that it can update the `TaskPriorities` table view.

Implementing the Add a record to the table method

In the previous section, we added some code to our `save` method that created a new `Task` instance and sent this information to the delegate object, located within our `taskDetailsViewController`. Next, we need to create the `AddTaskDetails` method.

Open the CIT implementation file, located within the TaskPriorities (CIT) folder and enter in the following code snippet:

```
- (void)taskDetailsViewController:(TaskDetailsViewController *)
controller AddTaskDetails:(Task *)task
{
    [self.tasks addObject:task];
    [self.tableView reloadData];
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

In the preceding code snippet, we add the new `task` object to our existing list of tasks priorities. We then refresh the table view, using the `reloadData` method to show that the new item was added, and then close the **Add New Task** screen.

Implementing the Cancel method

Next, we need to implement the **Cancel** button. This will be responsible for closing the screen, and returning you back to the TaskPriorities table view when pressed.

Open the `TaskDetailsViewController.m` implementation file located within the TaskPriorities folder, and enter in the following code snippet:

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

In the preceding code snippet, we use the `dismissViewControllerAnimated:completion:` method, which is only made available in iOS 5 and later. This method is used to close the current modal screen that was sent by our TaskPriorities table view screen.

Implementing the Refresh button method

Next, we need to implement the **Refresh** button. This will be responsible for refreshing our TaskPriorities table view when pressed.

Open the `TasksViewController.m` implementation file, located within the `TaskPriorities` folder, and enter in the following code snippet:

```
-(IBAction) refresh:(id) sender
{
    [self.tableView reloadData];
}
```

In the preceding code snippet, we use the `reloadData` method of the table view that will be used to refresh the data source property of the `TaskPriorities` table view to reload the contents from our tasks array.

Implementing the Delete row method

Next, we need to implement the **Delete** method. This will be responsible for removing an item from the `TaskPriorities` table view.

Open the `CIT` implementation file, located within the `TaskPriorities` folder, and enter in the following highlighted code:

```
// Override to support editing the table view.
- (void) tableView:(UITableView *)tableView commitEditingStyle:(UITab
leTableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)
indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        // Delete the row from the data source
        [self.tasks removeObjectAtIndex:indexPath.row];
        [self.tableView reloadData];
    }
}
```

In the preceding code snippet, we determine the type of action that we are performing within the table view, and this is determined by the `UITableViewCellEditingStyle` class. We then perform a comparison against the `UITableViewCellEditingStyleDelete` constant variable, and if the condition is met, we remove the selected task at the selected row from our tasks array, and refresh the table view data source.

Finishing up

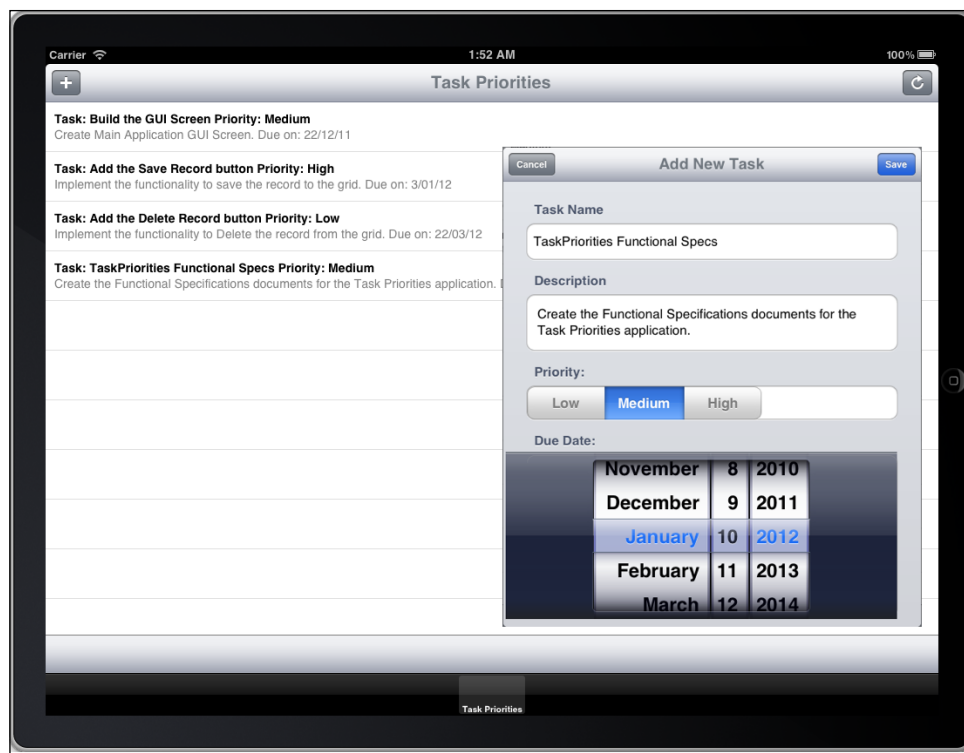
We just have a few more things to implement before we have a complete working application. We will need to implement a couple more methods that will be used to handle the transition between our **Task Priorities** and **Add New Task** screens when the **+** button has been pressed from the **Task Priorities** screen. We also need to let the **Tasks Priorities** screen notify the `TaskDetailsViewController` that it is now a delegate, so that it can communicate with other view controllers within the Storyboard using this delegate object, by calling the navigation controller whenever the segue is called.

1. Open the `TasksViewController.m` implementation file, located within the `TaskPriorities` folder, and enter in the following code snippet:

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id) sender
{
    if ([segue.identifier isEqualToString:@"AddNewTask"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        TaskDetailsViewController *taskDetailsViewController =
            [[navigationController viewControllers] objectAtIndex:0];
        taskDetailsViewController.delegate = self;
    }
}
```

In the preceding code snippet, we use the `prepareForSegue:` method to determine whenever a transition to segue takes place; a check is made on the identifier of the segue to determine if we are calling the **Add New Task** screen.

2. Next, we set the `navigationController` of the segue to be the navigation controller of the destination screen, and then cycle through each of the view controllers within the navigation controller properties to get the `TaskDetailsViewController` instance.



Congratulations, we have finally implemented the methods for our `TaskPriorities` application. Next, we are ready to build and run our application by choosing **Product | Run** from the **Product** menu, or alternatively by pressing `Command+ R` to have this run within the iOS Simulator, as shown in the preceding screenshot.

Summary

In this chapter, we learned how to create a `TaskPriorities` application, making use of the Storyboard feature, as well as creating relationships between the Tab Bar controllers and Navigation controllers using segues. We looked at how to set up delegates between table views, so that information can be passed back and forward, before finishing up learning about table views' Static Cells, and how to add object controls to the cells within each group.

In the next chapter, we will look at how to create an application that will give you the ability to record your voice, and have this played back to you or e-mailed as an attachment, to be played back later.

3

VoiceRecorder App – Audio Recording and Playback

The `VoiceRecorder` application is a small application that allows the user to record audio sounds using the iPad's built-in microphone, and then plays back the saved audio content at a later stage.

This voice recorder application can be very useful, and it can be used for sending simple and short voice messages, or even used more in business to take dictation notes during meetings, and eventually have this e-mailed and distributed amongst your peers.

In this chapter, we will be taking a closer look at how we can use both the `AVFoundation` and `MessageUI` frameworks to record and play back audio content using the iOS's device built-in microphone, and then start to design the user interface for our app.

We will also look at how we can create a new `UIViewController` instance to create our custom `VoiceVisualizer` class that will be used to measure our voice sound levels, and familiarize ourselves with the `MFMailComposeViewController` class, to see how we can add an audio file as an e-mail attachment, so that it can be e-mailed to one or more people.

In this chapter we will:

- Get an overview of the technologies that we will be using
- Learn how to add the `AVFoundation` and `MessageUI` frameworks
- Walk through the steps to build the `VoiceRecorder` application
- Implement the `VoiceVisualizer` class to measure our voice levels

- Implement a method to record our voice and save to a file
- Implement a method to play back our voice recording
- Implement a method to e-mail our voice recording

We have an exciting project ahead of us, so let's get started.

Overview of the technologies

The `VoiceRecorder` application makes reference to two very important frameworks to allow for audio recording and playback, as well as sending e-mail messages, directly within the application.

In this chapter, we will use the `AVFoundation` framework to handle recording and playback of our audio content, and the `MessageUI` framework for composing and sending our recording as an e-mail attachment directly within our app.

We will use the `AVAudioRecorder` class to record audio using the iOS device built-in microphone, and then save this to an audio file within the app. The `AVAudioPlayer` is used to play the previously saved audio when the **Play** button is pressed.

We make use of the `NSSearchPathForDirectoriesInDomains` class to create a list of path strings for each directory and then obtain the root directory, determined as the first object instance, before appending our attachment filename to the end of the path. We use the `NSData` object to convert the audio recording, and then pass this as an attachment to an `MFMailComposeViewController` class object. This class opens up the e-mail dialog-boxes directly within the application.



For more information on the `NSSearchPathForDirectoriesInDomains` class, refer to the Apple Developer Documentation located at the following location: https://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation_Functions/Reference/reference.html#//apple_ref/c/func/NSSearchPathForDirectoriesInDomains.

Building the VoiceRecorder app

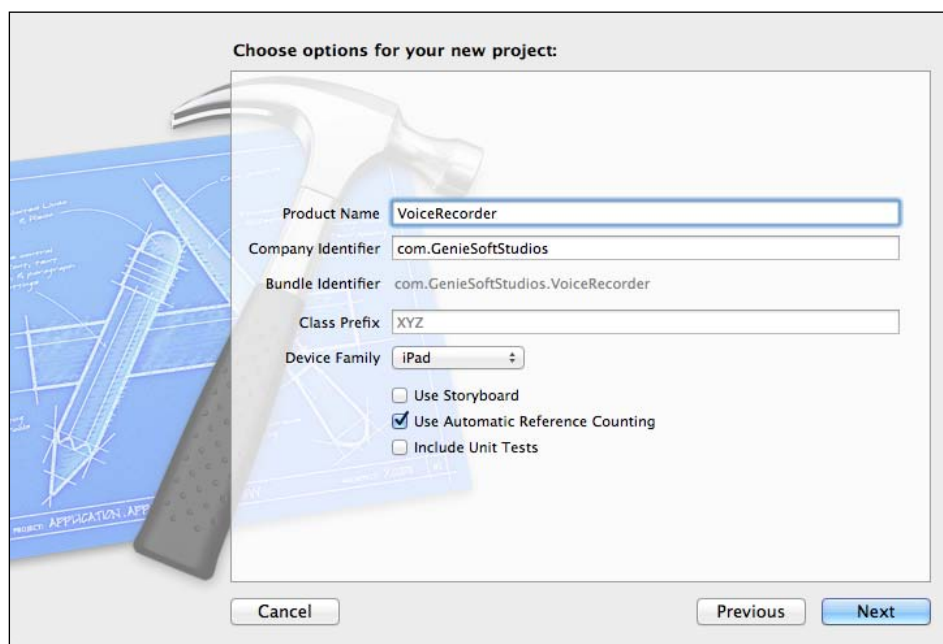
Leaving voice messages is one of the most common things that we do in our everyday lives; we call someone and leave a voice mail message when their phone is left unattended, so that they can get back to us.

In this section, we will take a look at how to create an application that we can use to run on an iOS device that will enable us to record and playback voice recordings, with the view of having these e-mailed and sent as an e-mail attachment.

Before we can proceed, we first need to create our `VoiceRecorder` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 2, Task Priorities – Building a TaskPriorities iOS App*, under the section named *Building the TaskPriorities App*.

It is very simple to create this in Xcode. Just follow the steps listed here:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the **Single View Application** template from the list of available templates.
4. Select **iPad** from under the **Device Family** drop-down.
5. Ensure that the **Use Storyboard** checkbox has not been selected.
6. Select the **Use Automatic Reference Counting** checkbox.
7. Ensure that the **Include Unit Tests** checkbox has not been selected.
8. Click on the **Next** button to proceed with the next step in the wizard:



9. Enter in `VoiceRecorder` as the name for your project.
10. Click on the **Next** button to proceed to the next step of the wizard.
11. Specify the location where you would like to save your project.
12. Then, click on the **Save** button to continue and display the Xcode workspace environment.

Now that we have created our `VoiceRecorder` project, we now need to add the `AVFoundation` and `MessageUI` frameworks that will handle the recording, playback, and e-mailing of our audio content.

Adding the AVFoundation and MessageUI frameworks

As we mentioned previously, we need to add the `AVFoundation` and `MessageUI` frameworks to our project to enable us to record and play back audio files, as well as enable us to have these e-mailed with attachments, all directly within our iOS application.

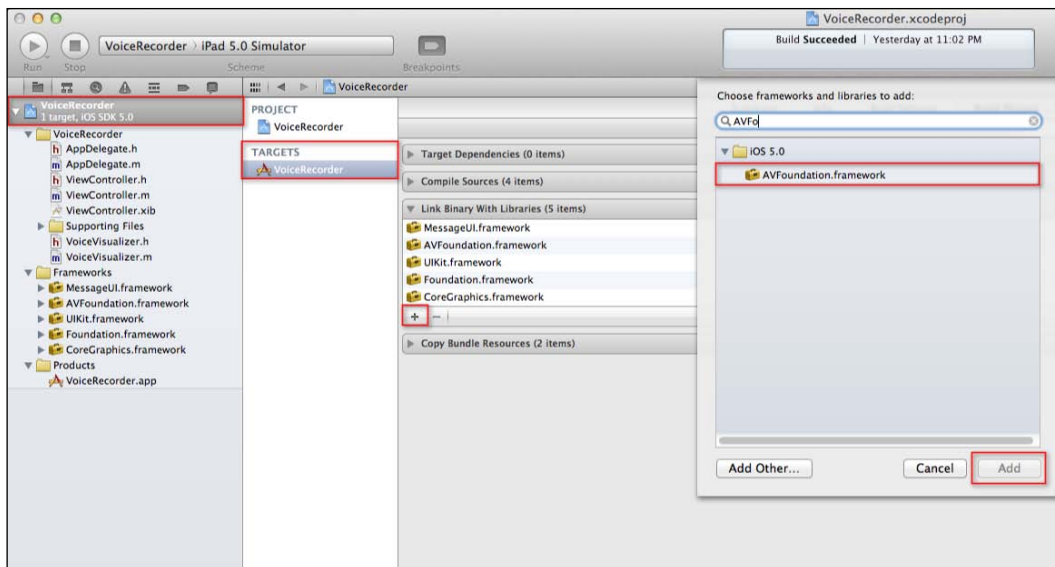
To add the `AVFoundation` framework, select **Project Navigator Group**, and follow these simple steps as outlined here:

1. Click and select your project from **Project Navigator**.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Binary With Libraries** disclosure triangle.
5. Finally, use the **+** to add the library you want.
6. Select the `AVFoundation.framework` from the list of available frameworks.



If you can't find the framework you are looking for, there is also the added ability to search for this directly, right from within the list of available frameworks.

If you are still confused how to go about adding the framework, follow this screenshot, which highlights the areas that you need to select (surrounded by a rectangles):

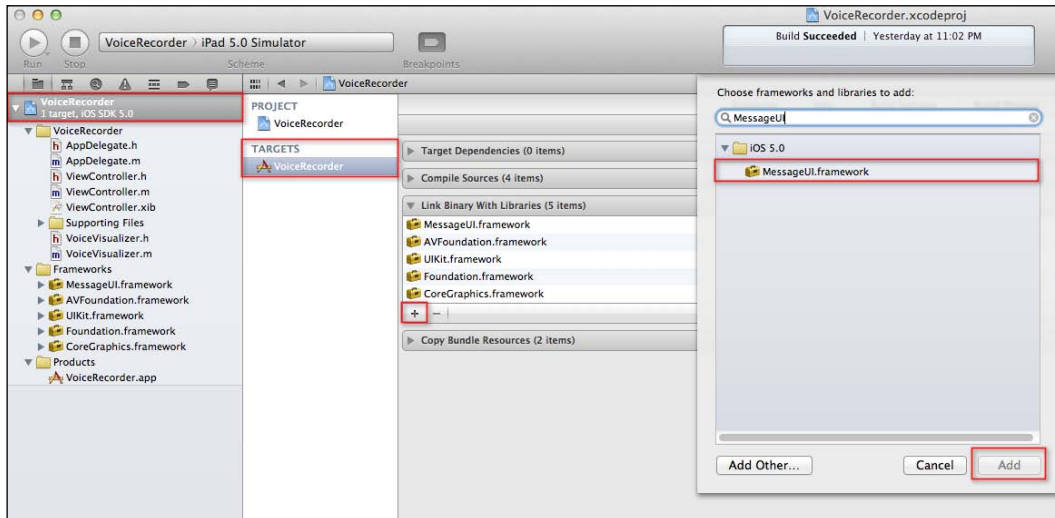


Next, we need to add the `MessageUI.framework` framework to our project that will allow us to compose and send our sound recording as an attachment, directly within our iOS application.

To add the `MessageUI` framework, select **Project Navigator Group**, and follow these simple steps as outlined here:

1. Click and select your project from **Project Navigator**.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Binary With Libraries** disclosure triangle.
5. Finally, use the **+** button to add the library you want.

6. Select the `MessageUI` framework from the list of available frameworks:

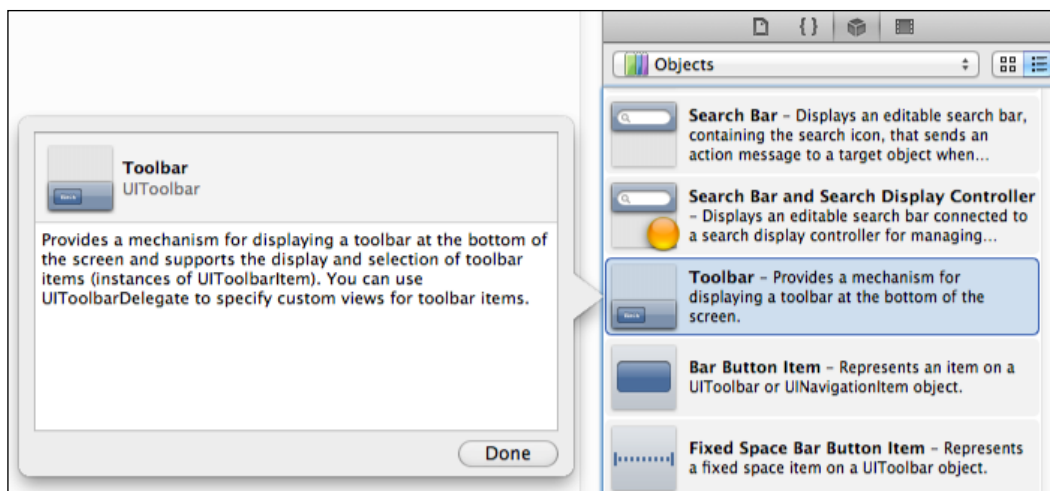


Now that we have added the `MessageUI` framework into our project, we need to start building our user interface that will be responsible for allowing us to record, playback, and e-mail our recording.

Creating the main application screen

We have successfully created our project and added the `AVFoundation` and `MessageUI` frameworks to handle the record/playback and e-mailing of our audio recording. Our next step is to build the user interfaces for our `VoiceRecorder` application. This screen will be very simple and will consist of just a View controller and a toolbar.

1. Select the `ViewController.xib` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UIToolbar) **Toolbar** Controller, and add this to our view:

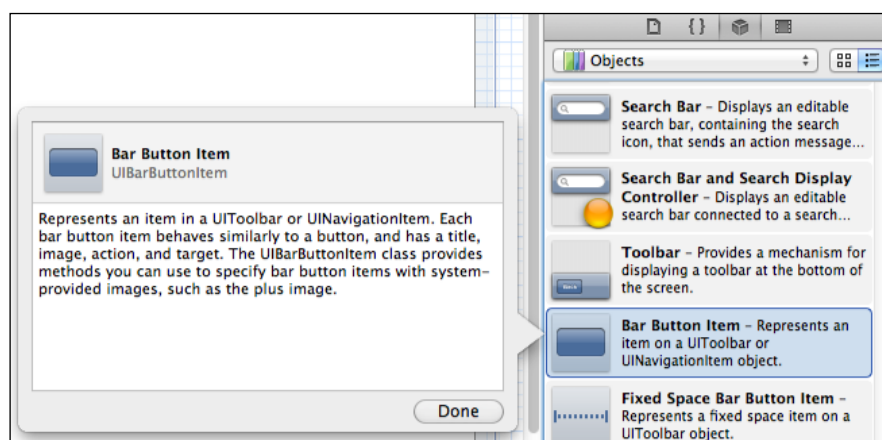


Now that we have added our `UIToolbar` toolbar control to our view controller, our next step is to start adding the **Start Recording**, **Play**, **Stop**, and **E-mail** buttons. So let's proceed to the next section.

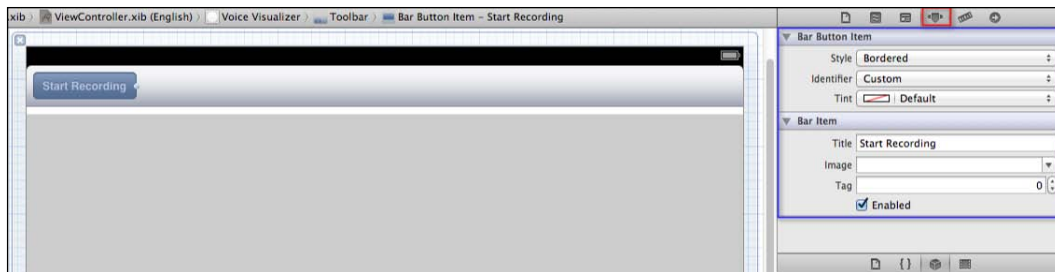
Adding the Start Recording button

Our next step is to add a button to our previously added toolbar; this button will be responsible for handling the recording of our audio content and storing this content within a file, to be used for playback later. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top-left corner of the **Voice Recorder** screen:



2. From the **Attributes Inspector** section, change the value of **Identifier** to **Custom**.
3. Change the value of **Style** to **Bordered**.
4. Then, change the value of **Title** to **Start Recording**:



Now that we have added our **Start Recording** button to our **Voice Recorder View Controller**, our next step is to add the **Play** button that will be responsible for playing back our recorded voice message when this has been clicked. So, let's proceed with the next section.

Adding the Play button

Now, we need to add another button on our `UIToolbar` that will be responsible for playing back our recorded audio, once the user has finished recording their message. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag another (`UIBarButtonItem`) **Bar Button Item** control after the **Start Recording** button, located within our `UIToolbar`.
2. From the **Attributes Inspector** section, change the value of **Identifier** to **Play**.
3. Change the value of **Style** to **Bordered**:

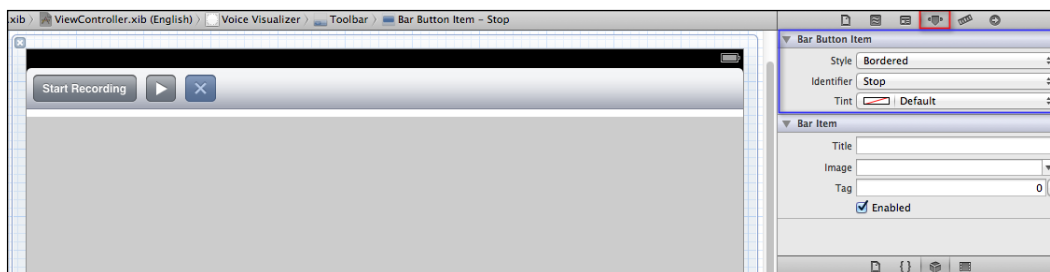


Now that we have added our **Play** button to our **Voice Recorder View Controller**, our next step is to add the **Stop** button that will be responsible for halting the playback our recorded voice message when this has been clicked.

Adding the Stop button

Now, we need to add another button to our **UIToolbar** that will be responsible for stopping audio recording, once the user has recorded what they wanted to say. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag another (**UIButtonItem**) **Bar Button Item** control after the **Play** button, located within our **UIToolBar**.
2. From the **Attributes Inspector** section, change the value of **Identifier** to **Stop**.
3. Change the value of **Style** to **Bordered**:

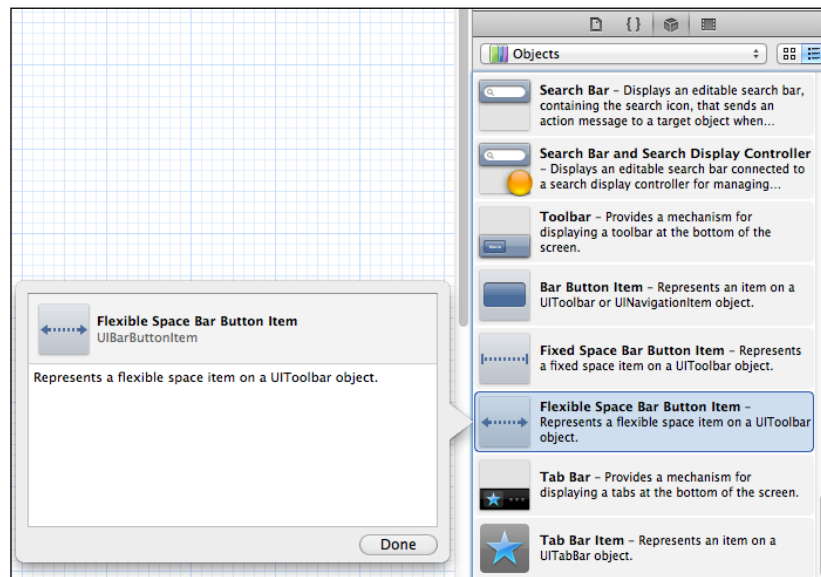


Now that we have added our **Stop** button to our **Voice Recorder View Controller**, our next step is to add a (**UIBarButtonItem**) **Flexible Space Bar Button Item** control that will be used to fill in the space between the **Stop** button and the **Compose** e-mail button, which will be responsible for composing and attaching our voice recording when it has been clicked.

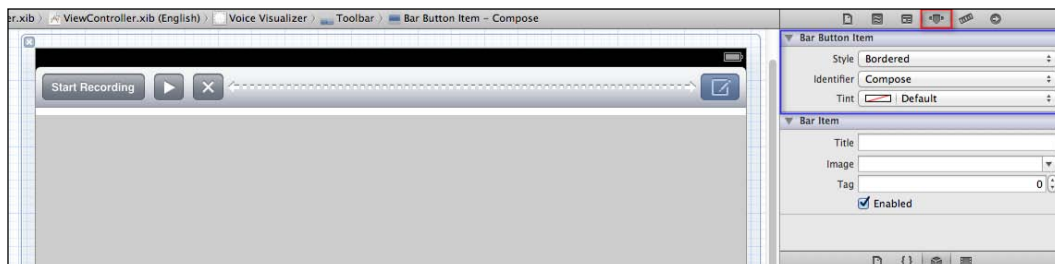
Adding the E-mail button

Now, we need to add our final button to our `UIToolbar` that will be responsible for allowing the user to e-mail their attached recording to an individual or a group of recipients. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Flexible Space Bar Button Item** control after the **Stop** button within our **View Controller**:

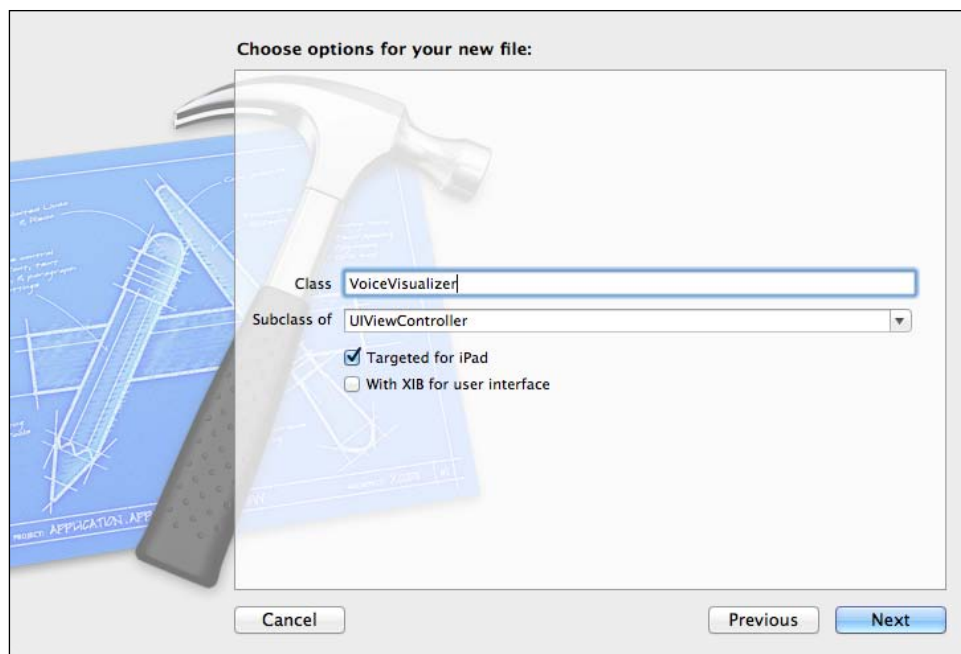


2. Next, from **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control after the `UIBarButtonItem` control, located within our `UIToolbar`.
3. From the **Attributes Inspector** section, change the value of **Identifier** to **Compose**.
4. Change the value of **Style** to **Bordered**:



Now that we have added our buttons and have built our user interface, our next step is to create our very own custom `UIViewController` subclass that will be used to display a visual representation of our voice:

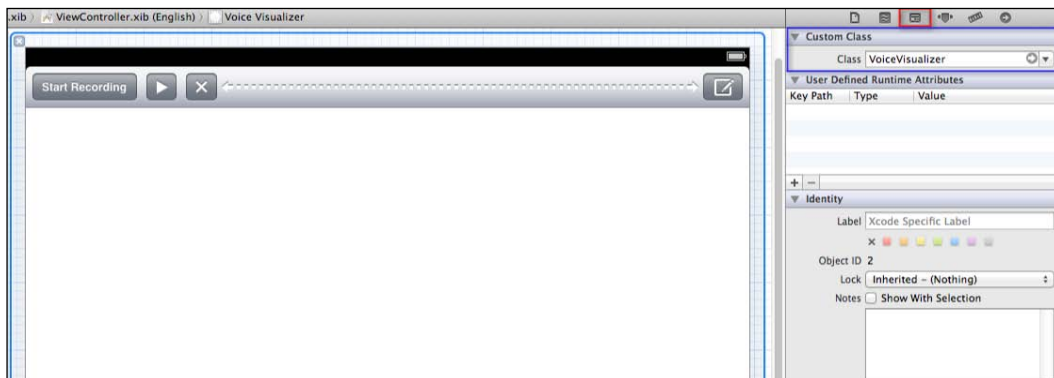
1. Select the `VoiceRecorder` folder, then choose **File | New | New File...** or press *Command + N*.
2. Select the **Objective-C** class template from the list of available templates.
3. Click on the **Next** button to proceed to the next step within the wizard.
4. Enter in `VoiceVisualizer` as the name of the file to be created.
5. Ensure that you have selected `UIViewController` as the type of subclass to create from the **Subclass** of dropdown.
6. Ensure that you have selected the **Targeted for iPad** option. This will ensure that the view is created using the iPad dimensions, not the iPhone dimensions:



7. Click on the **Next** button to proceed with the next step of the wizard.
8. Then, click on the **Create** button to save the file to the folder location specified.

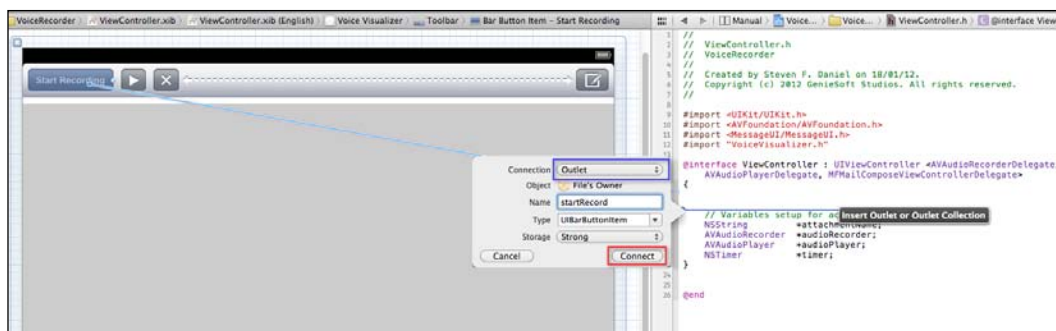
Now that we have created our voice visualizer class, we need to update our **Voice Recorder View Controller** to use our `VoiceVisualizer` class, rather than the default `UIViewController` class:

1. Select the `ViewController.xib` file from the `VoiceRecorder` folder.
2. Click and select our `VoiceRecorder (UIViewController)` controller.
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `VoiceVisualizer`:



Our next step is to create the `Outlet` events for the **Start Recording**, **Play**, **Stop**, and **Compose E-mail** buttons. Creating these will allow us to access these controls within our code and make modifications to the control properties. To create an `Outlet`, follow these simple steps:

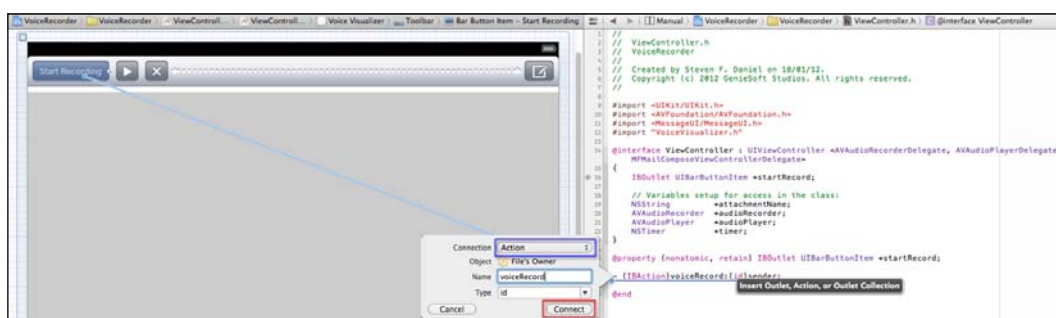
1. Open **Assistant Editor** by choosing **Navigate | Open in Assistant Editor**, or press *Option + Command + ,*.
2. Ensure that the `ViewController.h` interface file is displayed to the left of `ViewController.xib`.
3. Select the **Start Recording** (`UIBarButtonItem`) control, then hold down the *Control* key, and drag it into the `ViewController.h` interface file.
4. Choose **Outlet** from the **Connection** dropdown for the connection to be created.
5. Enter in `startRecord` as the name of the **Outlet** property to be created.
6. Choose **Strong** from the **Storage** dropdown:



7. Repeat steps 3 to 6 to create the IBOutlets for the **Play**, **Stop**, **Compose**, and **VoiceVisualizer** controls, while providing the following naming for each as `startPlayback`, `stopPlayback`, `composeE-mail`, and `visualizer`, respectively.

Now that we have created the `Outlet` events for our controls, we need to create the associated `Action` events for those `Outlets`. Creating these actions allows an event to be fired when the button has been pressed. To create an **Action**, follow these simple steps:

1. The `ViewController.h` interface file should still be displayed to the left of the `ViewController.xib` View Controller.
2. Select the **Start Recording** (`UIBarButtonItem`) control, then hold down the **Control** key, and drag it into the `ViewController.h` interface file.
3. Choose **Action** from the **Connection** dropdown for the connection to create.
4. Enter in `voiceRecord` for the **Name** field, which is the name of the method to be created:



5. Repeat steps 2-4 to create the IBActions for the **Play**, **Stop**, and **Compose** controls, while providing the naming for each as `voicePlayback`, `voicePlaybackStop`, and `e-mailRecording`, respectively.

Now that we have successfully finished building the user interface for both the **Voice Recorder** and **Voice Visualizer** screens, connected up each of our controls, and created the required outlets and associated the action methods. Our next step is to start implementing the methods that will be used by our **Start Recording**, **Play**, **Stop** and **Compose** buttons. These buttons will be responsible for handling the recording and audio playback, as well as for e-mailing our audio recording as a file attachment.

Implementing the View Controller class

We are now ready to start adding additional content to our `ViewController` class. We need to import some interface header files and declare some objects that we will be using throughout our application. We will also need to extend our class, so that we can use the `AVFoundation` and mail composition functionality.

1. Open the `ViewController.h` interface file, located within the `VoiceRecorder` folder, and enter in the following highlighted code sections:

```
// ViewController.h
// VoiceRecorder
//
// Created by Steven F. Daniel on 18/01/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>
#import <MessageUI/MessageUI.h>
#import "VoiceVisualizer.h"

@interface ViewController : UIViewController
<AVAudioRecorderDelegate, AVAudioPlayerDelegate,
MFMailComposeViewControllerDelegate>
{
    IBOutlet UIBarButtonItem *startRecord;
    IBOutlet UIBarButtonItem *startPlayback;
    IBOutlet UIBarButtonItem *stopPlayback;
    IBOutlet UIBarButtonItem *composeE-mail;
    IBOutlet VoiceVisualizer *visualizer;

    // Variables setup for access in the class:
    NSString *attachmentName;
    AVAudioRecorder *audioRecorder;
```

```

        AVAudioPlayer    *audioPlayer;
        NSTimer          *timer;
    }

    @property (nonatomic, strong) IBOutlet VoiceVisualizer
    *Visualizer;
    @property (nonatomic, strong) IBOutlet UIBarButtonItem
    *startPlayback;
    @property (nonatomic, strong) IBOutlet UIBarButtonItem
    *startRecord;
    @property (nonatomic, strong) IBOutlet UIBarButtonItem
    *composeE-mail;

    - (IBAction)voiceRecord:(id)sender;
    - (IBAction)voicePlayback:(id)sender;
    - (IBAction)voicePlaybackStop:(id)sender;
    - (IBAction)e-mailRecording:(id)sender;

```

In the preceding code snippet, we import the interface file header information for our `AVFoundation.h`, `MessageUI.h`, and `VoiceVisualizer.h` interface files, so that we can access their class methods. We extended our class so that we can include the `AVAudioRecorderDelegate`, `AVAudioPlayerDelegate`, and `MFMailComposeViewControllerDelegate` class protocols, as well as their methods.

We then declared a new outlet to our `VoiceVisualizer` to display the intensity of the user's voice during recording, and then declared an `NSTimer` object that will be used to generate the events required to redraw the visualizer based on the voice input.

2. Next, open the `ViewController.m` implementation file, located within the `VoiceRecorder` folder, and modify the `viewDidLoad` method as shown in the following code snippet:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Do any additional setup after loading the view,
    // typically from a nib.
    NSArray *dirPaths;
    NSString *docsDir;

    attachmentName = @"VCOR_Recording.caf";
    dirPaths = NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES);

```

```
docsDir = [dirPaths objectAtIndex:0];
NSString *soundFilePath = [docsDir
stringByAppendingPathComponent:attachmentName];

NSURL *soundFileURL = [NSURL
fileURLWithPath:soundFilePath];

// Initialize the recorder with default settings.
NSDictionary *recordSettings = [NSDictionary
dictionaryWithObjectsAndKeys:

[NSNumber
                                numberWithInt:AVAudioQualityMin],

AVEncoderAudioQualityKey,

[NSNumber numberWithInt:16],

AVEncoderBitRateKey,

[NSNumber numberWithInt:2],

AVNumberOfChannelsKey,

[NSNumber
                                numberWithFloat:44100.0],

AVSampleRateKey, nil];

// Initialize our audioRecorder settings and
// the default recording filename
audioRecorder = [[AVAudioRecorder alloc]
initWithURL:soundFileURL settings:recordSettings
error:nil];

[audioRecorder setDelegate:self];
[audioRecorder prepareToRecord];
audioRecorder.meteringEnabled = YES;

composeE-mail.enabled = NO;
stopPlayback.enabled = NO;
startPlayback.enabled = NO;

// Initialize our visualizer and timer.
```

```

[Visualizer clearOSCLevels];
[timer invalidate];
timer = nil;

// Set the background color of our view to black
self.view.backgroundColor = [UIColor blackColor];
}

```

In the preceding code snippet, we need to create an instance of the `AVAudioRecorder` class when the application is first launched. This method is then initialized with a URL pointing the default filename to which the recorded audio is saved, and declare an `NSDictionary` object to initialize and set up the recording settings for the audio.

3. Next, we use the `NSSearchPathForDirectoriesInDomains` class to identify the application's document directory, then construct a URL to a file in that location named `VCOR_Recording.caf`, and prepare the `audioRecorder` instance to begin recording when the user requests it.
4. We then set the `meteringEnabled` property of our `audioRecorder` object to monitor the level of audio that is coming in. Finally, we disable our **Play**, **Stop**, and **E-mail** buttons, since no audio has been recorded yet, call the `clearOSCLevels` method to reinitialize our `Visualizer` class, and stop the timer from firing the events used to redraw our voice visualizer.



For more information on the `AVFoundation` class, refer to the Apple Developer Documentation located at the following location: http://developer.apple.com/library/mac/#documentation/AVFoundation/Reference/AVAudioRecorder_ClassReference/Reference/Reference.html#//apple_ref/occ/cl/AVAudioRecorder.

Implementing the `voiceRecord` method

Now that we have set up our `VoiceRecorder` View Controller and have initialized everything correctly, we are ready to start implementing the method that will be responsible for recording the audio when the user presses the **Start Recording** button:

1. Open the `ViewController.m` implementation file, located within the `VoiceRecorder` folder, locate the `voiceRecord` method, and enter in the following code snippet:

```

//=====
// Handles recording of the audio
//=====

```



```
- (IBAction)voiceRecord:(id)sender
{
    // Check to see if we are already recording.
    if (!audioRecorder.recording)
    {
        // Initialize our Timer event for the Voice Visualizer
        timer = [NSTimer scheduledTimerWithTimeInterval:0.02
                target:self selector:@selector(timerFired:)
                userInfo:nil repeats:YES];

        [startRecord setTitle:@"Stop Recording" ];
        startPlayback.enabled = NO;
        stopPlayback.enabled = NO;
        [audioRecorder record];
    }
    else
    {
        [startRecord setTitle:@"Start Recording" ];
        startPlayback.enabled = YES;
        stopPlayback.enabled = NO;
        composeE-mail.enabled = YES;

        // Stop the recorder and reset our visualizer.
        [audioRecorder stop];
        [Visualizer resetOSCLevels];
        [timer invalidate];
        timer = nil;
    }
}
```

In the preceding code snippet, we use the `audioRecorder` object to determine if we are currently recording. If we have determined that we are not recording, we initialize our timer to start generating the events and redraw our voice visualizer, then change the text of our `startRecord` button to display **Stop Recording**, disable the `startPlayback` and `stopPlayback` methods, and then set up our `audioRecorder` object to begin the recording.

If we are currently recording, we change the text of our `startRecord` method to display **Start Recording** and disable the `stopPlayback` method, stop our `audioRecorder` object from recording, and call the `clearOSCLevels` method to reinitialize our `Visualizer` class. Finally, we need to tell our timer object to stop firing the events used to redraw our voice visualizer.

Implementing the voicePlayback method

Now, we need to start implementing the method that will be responsible for playing back our previously recorded audio when the user presses the **Play** button:

1. Open the `ViewController.m` implementation file, located within the `VoiceRecorder` folder, locate the `voicePlayback` method, and enter in the following code snippet:

```
//=====
// Handles playback of our recording.
//=====
- (IBAction)voicePlayback:(id)sender
{
    [Visualizer resetOSCLevels];

    // Check to see if we are already playing.
    if (!audioPlayer.playing)
    {
        // Grab the recorded file from the url location.
        audioPlayer = [[AVAudioPlayer alloc]
                       initWithContentsOfURL:audioRecorder.url
                       error:nil];

        // Next, play our audio file
        audioPlayer.delegate = self;
        [audioPlayer prepareToPlay];
        [audioPlayer play];

        // Enable the stop playback button
        stopPlayback.enabled = YES;
    }
}
```

In the preceding code snippet, we call our `clearOSCLevels` method of our `Visualizer` class to reset the sound levels recording array, and perform a check to ensure that our `audioPlayer` object is not in the process of already playing the audio. Next, we initialize the `audioPlayer` object to play the file recording using the `URL` method of the `audioRecorder` object.

2. Finally, we then set up our `audioPlayer` object to use the `prepareToPlay` method, and call the `play` method to start the playback. Finally, we enable our `stopPlayback` method, so that we can stop recording the playback, if required.

Implementing the voicePlaybackStop method

Now, we need to start implementing the method that will be responsible for handling the stopping of the audio playback currently being played, when the user presses the **Stop** button.

Open the `ViewController.m` implementation file, located within the `VoiceRecorder` folder, locate the `voicePlaybackStop` method, and enter in the following code snippet:

```
//=====
// Stops playback of our recording
//=====
- (IBAction)voicePlaybackStop: (id) sender
{
    // Check to see if our audio is playing prior to stopping.
    if (audioPlayer.playing)
    {
        [audioPlayer stop];
        stopPlayback.enabled = NO;
    }
}
```

In the preceding code snippet, we perform a check on our `audioPlayer` object to see if we are currently playing our voice recording. If we are, we make a call to the `stop` method on our `audioPlayer` object, and then disable the `stopPlayback` button.

Implementing the e-mailRecording method

Now, we need to implement a method that will be responsible for allowing you to attach a voice-recording file to an e-mail message, so that you can send this to an individual person or a group of people. This will be displayed when the user presses the **Compose** button.

1. Open the `ViewController.m` implementation file, located within the `VoiceRecorder` folder, locate the `e-mailRecording` method, and enter in the following code snippet:

```
//=====
// Sends an e-mail with our recording attached
//=====
- (IBAction)e-mailRecording: (id) sender
{
    NSArray *arrayPaths =
    NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
```

```

NSString *docDir = [arrayPaths objectAtIndex:0];
NSString *Path = [docDir stringByAppendingString:@"/"];

Path = [Path stringByAppendingFormat:attachmentName];

// Create an NSData object with the selected recording
NSData *audioData = [NSData dataWithContentsOfFile:Path];

MFMailComposeViewController *controller =
[[MFMailComposeViewController alloc] init];
[controller.navigationBar setTintColor:[UIColor
purpleColor]];

controller.mailComposeDelegate = self;

// Add the recording as an attachment
[controller setSubject:@"Voice Recorder File"];
[controller setMessageBody:@"Recorded File Recording
attached" isHTML:NO];

[controller addAttachmentData:audioData
mimeType:@"audio/mp4" fileName:attachmentName];

// show the MFMailComposerVewController
[self presentModalViewController:controller animated:YES];
}

```

In the preceding code snippet, we use the `NSSearchPathForDirectoriesInDomains` class to identify our root directory folder location to where our audio recording filename `VCOR_Recording.caf` is located. We then create an `NSData` object that contains the contents of the audio recording file using the `dataWithContentsOfFile:` method of `NSData`.

2. Next, we create a new object instance of the `MFMailComposeViewController` class, which controls the mail dialog view, thus allowing the user to compose and send an e-mail without leaving the application. We then change the color of the mail composition sheet using the `navigationBar:setTintColor:` method of the controller to purple, and then set the subject heading and body of our e-mail message, before using the `addAttachmentData:mimeType:fileName:` method to add the audio recording file as an e-mail attachment.

3. We then set the value of the mailComposeDelegate controller to self so that our controller receives the mailComposeController:didFinishWithResult:error: message from the MFMailComposeViewControllerDelegate protocol when the user finishes with the e-mail dialog.
4. Next, we call the controller's presentViewController:animated: method to display the e-mail dialog.



For more information on the MFMailComposeViewController class, refer to the Apple Developer Documentation located at the following location: https://developer.apple.com/library/ios/#documentation/MessageUI/Reference/MFMailComposeViewControllerDelegate_protocol/Reference/Reference.html.

```
//=====
// Dismiss our Mail view controller when the user finishes
//=====
- (void) mailComposeController:(MFMailComposeViewController *)
controller didFinishWithResult:(MFMailComposeResult)result
error:(NSError *)error
{
    NSString *e-mailMessage = nil;

    // Notifies users about errors associated with
    // the interface
    switch (result)
    {
        case MFMailComposeResultCancelled:
            e-mailMessage = @"E-mail sending has been cancelled";
            break;
        case MFMailComposeResultSaved:
            e-mailMessage = @"E-mail draft saved successfully";
            break;
        case MFMailComposeResultSent:
            e-mailMessage = @"E-mail sent successfully.";
            break;
        case MFMailComposeResultFailed:
            e-mailMessage = @"E-mail sending failure.";
            break;
        default:
            e-mailMessage = @"Problem sending the e-mail.";
            break;
    }
}
```

```

        // Display the alert dialog based on the message derived
        // from the above case statement.
        UIAlertView *alert = [[UIAlertView alloc]
                               initWithTitle: @"Voice
Recorder E-mail"
                               message: e-mailMessage
                               delegate: nil
                               cancelButtonTitle:@"OK"
                               otherButtonTitles:nil];

        [alert show];

        // make the MFMailComposeViewController disappear
        [self dismissModalViewControllerAnimated:YES];
    }

```

In the preceding code snippet, we declare our `mailComposeController:mailComposeController:didFinishWithResult:error: method`, which will be responsible for notifying the user when the user finishes with the e-mail dialog box, either by sending an e-mail or by dismissing out of this view.

5. Next, we determine the type of error that was received by the delegate, and then assign this to an `NSString` object variable `e-mailMessage`.
6. In our final step, we declare an instance of the `UIAlertView` dialog box to display the error message, before calling the `dismissModalViewControllerAnimated: method` of our view controller.

Implementing the VoiceVisualizer class

In our final step, we need to implement the class that will be used to display a graphical visual representation of our voice intensity during the recording of our audio, using the iOS device's built-in microphone:

1. Open the `VoiceVisualizer.h` interface file, located within the `VoiceRecorder` folder, and enter in the following code snippet:

```

//
//  VoiceVisualizer.h
//  VoiceRecorder
//
//  Created by Steven F. Daniel on 18/01/12.
//  Copyright (c) 2012 GenieSoft Studios. All rights reserved.
//

#import <UIKit/UIKit.h>

```

```
@interface VoiceVisualizer : UIView
{
    NSMutableArray *OSCLevel;           // Levels in the recording
    float minOSCLevel;                 // Minimum recorded level
}

- (void) setOSCLevel:(float)level; // set the OSCLevel
- (void) clearOSCLevels;           // clear all OSC levels

@end
```

In the preceding code snippet, we declare a new instance of our `UIView` sub-class, as our `VoiceVisualizer` will act as a view within our `ViewController` on our main screen. Next, we declare an `NSMutableArray` object variable `OSCLevel` that will hold the intensity levels from the `ViewController`.

2. Next, we declare a `minOSCLevel` variable to store the lowest recorded level for the current recording. We then declare the `setOSCLevel:` method to basically add a new level to the `OSCLevel` `NSMutableArray` object, and update the `minOSCLevel` variable accordingly. Next, we declare the `clearOSCLevels` method to remove all the previously added items from the `OSCLevel` array, thus removing the visualizer representation from appearing on screen.
3. Open the `VoiceVisualizer.m` implementation file, located within the `TaskPriorities` folder, and enter in the following code snippet:

```
// initialize the Visualizer
- (id)initWithCoder:(NSCoder *)aDecoder
{
    // if the superclass initializes properly
    if (self = [super initWithCoder:aDecoder])
    {
        // initialize powers with an entry for every
        // other pixel of width
        OSCLevel = [[NSMutableArray alloc]
                    initWithCapacity:self.frame.size.width / 2];
    }

    return self;
}
```

In the preceding code snippet, we create an `initWithCoder:` method object to initialize the voice visualizer, and check to ensure that it has been initialized correctly, prior to initializing our `OSCLevel` array with half of the screen's width using the `initWithCapacity:` method of `NSMutableArray`:

```
// sets the current power in the recording
- (void)setOSCLevel:(float)level
{
    [OSCLevel addObject:[NSNumber numberWithFloat:level]];

    // Need to ensure that our current level is
    // not less than the last one that was recorded.
    if (level < minOSCLevel) minOSCLevel = level;
}
```

In the preceding code snippet, we add the current recording voice intensity level to the voice visualizer.

4. We then perform a check to see if the current recording level is lower than what was previously recorded; if the comparison returns `TRUE`, we update the `minOSCLevel` variable to the given value:

```
// Resets the objects contained within our Array.
- (void)resetOSCLevels
{
    [OSCLevel removeAllObjects];
} // clears all the points from the visualizer
```

In the preceding code snippet, we perform a call to the `removeAllObjects` method of `NSMutableArray` to remove all array elements from the `OSCLevel` array, resulting in the voice visualizer graphic being cleared.

```
// draws the visualizer
- (void)drawRect:(CGRect)rect
{
    // Get the current graphics context
    CGContextRef context;
    CGSize size;
    CGFloat cHeight;

    context = UIGraphicsGetCurrentContext();
    size = self.frame.size;
    cHeight = size.height;

    // Cycle through each of the levels and plot a line.
    for (int i = 0; i < OSCLevel.count; i++)
    {
```



```
// Get the next determined sound wave level and
// calculate the height.
float newLevel = [[OSCLLevel objectAtIndex:i]
                  floatValue];

float height = (1-newLevel/minOSCLLevel)*(cHeight/ 2);

// Move to a point located within the center of the
// screen and plot a line.
CGContextMoveToPoint(context, i*2, cHeight/2-height);
CGContextAddLineToPoint(context, i*2,
cHeight/2+height);

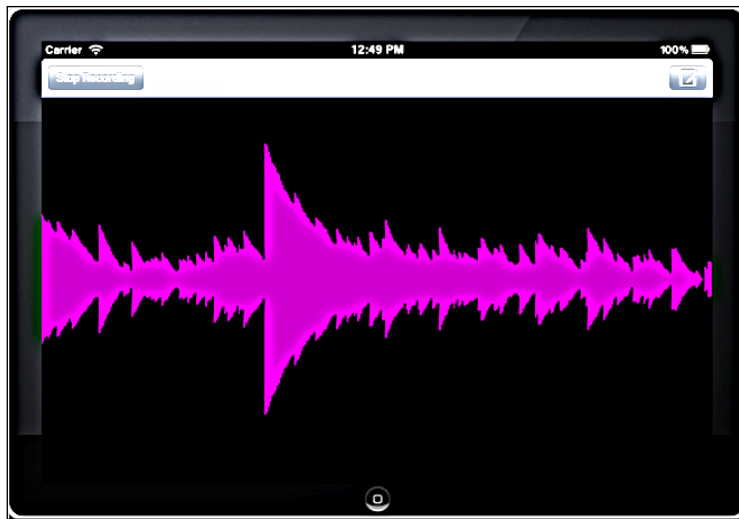
// Set the line color and then draw it to the view.
CGContextSetRGBStrokeColor(context, 1, 0, 1, 1);
CGContextStrokePath(context);
}
}
```

In the preceding code snippet, we use the `drawRect:` method to draw each of our voice intensity levels from our `NSMutableArray` object to `VoiceVisualizer`. We first need to obtain the current graphics context using the `UIGraphicsGetCurrentContext` function, and then work out the size and height of our `VoiceVisualizer`, using the controller's frame property.

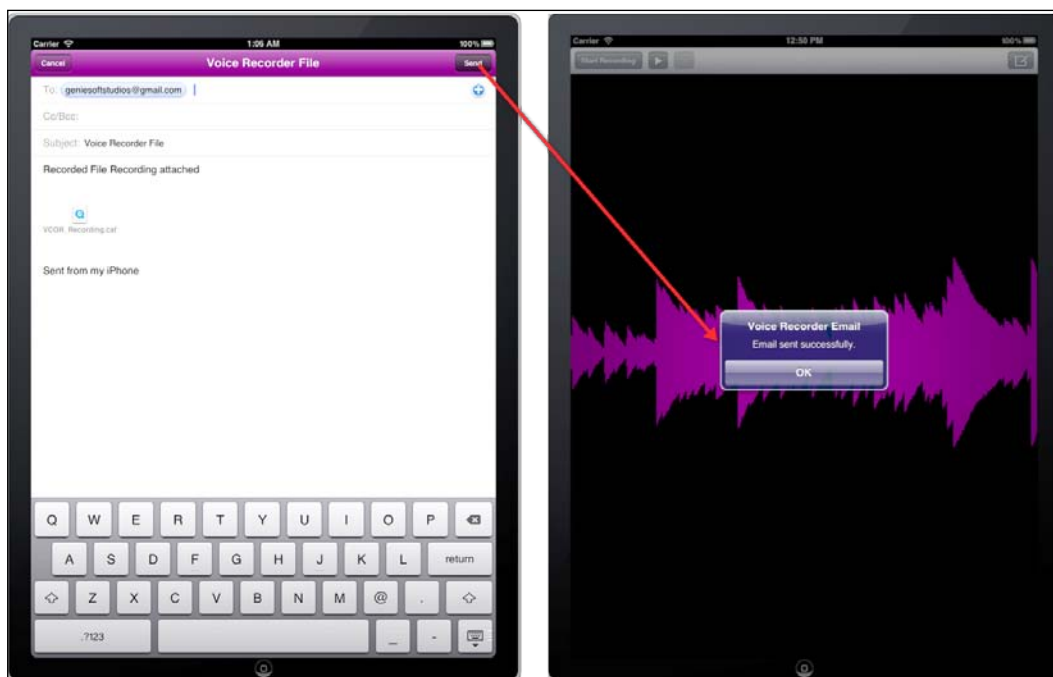
5. Next, we perform a loop to cycle through each of the intensity levels within the `OSCLLevel` array, calculate the height of the line, and use `CGContextMoveToPoint` and `CGContextAddLineToPoint` to work out and draw the line in the middle of the screen.
6. Finally, we use the `CGContextSetRGBStrokeColor` function to set the line color to purple, and use the `CGContextStrokePath` function to draw our line to the graphics window.

Finishing up

Congratulations, we have successfully implemented the methods for our `VoiceRecorder` application. Next, we are ready to build and run our application by choosing **Product | Run** from the **Product** menu, or alternatively pressing *Command* + *R*. The following screenshot shows the `VoiceRecord` application currently within the iOS Simulator in the middle of a voice recording:



In the following screenshot, we display the e-mail composition sheet when the **Compose** button has been pressed on the toolbar. As you can see, it contains the pre-populated **Subject** and body message, as well as the audio recording being attached as a file attachment:



From this screenshot, you can amend the subject header, and include additional content within the body of the message. Once you have finished composing your e-mail, click on the **Send** button to have your e-mail sent. You will receive a confirmation, as shown in the preceding screenshot.

Summary

In this chapter, we learned how to use the `AVFoundation` framework and `AVAudioRecorder` to record sounds using the iOS device's built-in microphone. We learned about the `AVAudioPlayer` framework, and how we can use this framework to play back audio content that was previously recorded.

We then looked at how to create a custom `UIViewController` sub-class to display a graphical representation of our voice level intensity, using the `meteringEnabled` method of the `AVAudioRecorder` class. To end the chapter, we looked at how we can use the `MFMailComposeViewController` class to allow the user to send e-mails directly within the app, as well as looking at how we can use the `NSData` object to convert the saved audio files, so that these can then be added as attachments to e-mails.

In the next chapter, we will look at how we can create an enhanced address book application that will store contact information into an SQLite database using the Core Data framework. We will also learn about the `GameKit` framework, and how we can use this to send a contact from one device to another using Bluetooth.

4

Enhanced AddressBook App – Core Data

The iPad comes with a built-in Bluetooth functionality, allowing it to communicate with other Bluetooth-capable devices, such as other iOS devices or Bluetooth-compatible headsets. In this chapter, we will take a look at how to create a simple `AddressBook` application, making use of Apple's powerful Core Data framework that will allow you to directly interface with a SQLite database to create and store client information using a form.

We will then take a look at how you can incorporate the Bluetooth functionality within your application, so that you can send this information by communicating with another iOS device, and have this information received wirelessly and stored within the database at the other end.

This may sound all a bit confusing at first, but you will soon come to see that by using the iOS SDK, Bluetooth programming is actually quite simple and this functionality is nicely encapsulated within the Game Kit framework.

In this chapter we will:

- Build the `AddressBook` application using Storyboards
- Build the `AddressBook` Core Data Model and create the table schema
- Learn how to navigate between screens using Storyboards
- Implement a functionality to populate `UITableView` from a database
- Implement a method to save a record to the database
- Implement a method to send the record using Bluetooth
- Implement a method to delete table view items
- Implement an ability to perform searches within `UITableView`

We have an exciting project ahead of us; so let's get started.

Overview of the Core Data technologies

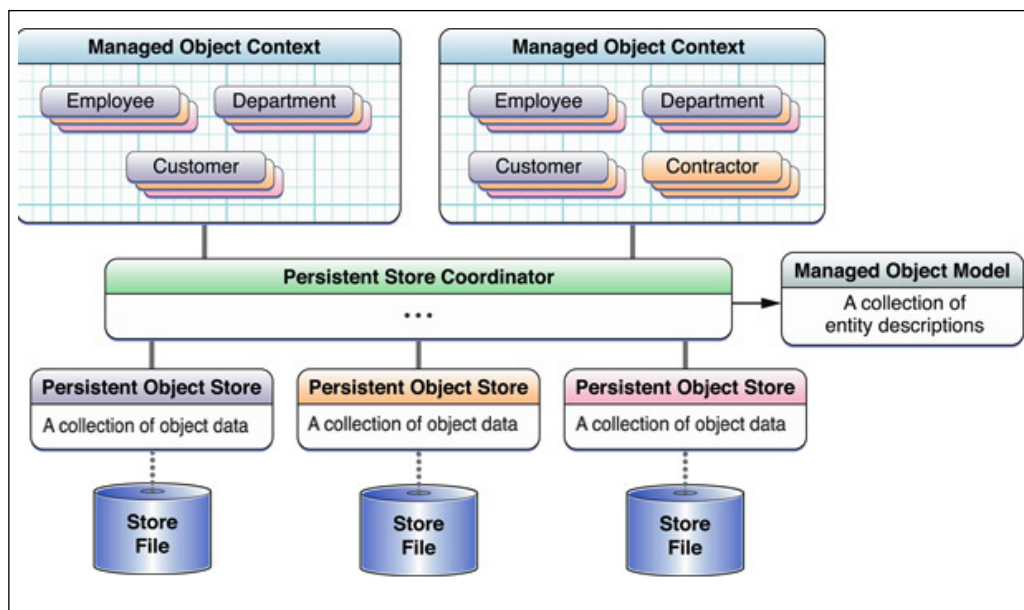
The enhanced AddressBook application makes reference to two very important frameworks: the **Game Kit framework** and the **Core Data framework**. The Game Kit framework allows for multiple iOS devices to communicate with each other over the Bluetooth network, to allow information to be sent and received from one device to another.

The Core Data framework is described as an abstraction layer that sits on top of a SQLite database, and enables developers to easily implement data-centric applications by modeling your data storage around entities (which are known as **classes**), which contain the relationships between them. If you are familiar with the Entity-Framework that comes as part of the Microsoft .NET framework, then this one is of a similar nature.

This framework can be described as a "Schema-driven object graph management and persistence framework" that was first released as part of the iOS 3.0 SDK, and helps manage where data is stored, how it is stored, how it is cached, and how it handles the management of memory.

Since Core Data is a large topic, we will be covering a small aspect of some of the features that come with Core Data. There are many books that already exist on the market, which go into more depth than what will be covered within this chapter.

The following screenshot shows the simplest and most common configuration of the stack. The objects that you will most likely work directly with are located at the top of the stack, including the managed object context and the managed data objects that it contains.



The following table shows you the three main management object models the Core Data framework contains.

Object models	Description
Managed Object Context	This associates the in-memory object with their associated in-storage counterparts
Managed Object	This is the in-memory representation of the data-model object and is saved in a persistent store (table)
Managed Object Model	This is the object-relational schema that contains the entity descriptions that are required to build the managed objects



For more information on the Core Data Framework, refer to the Core Data Programming Guide located within the Apple Developer Documentation at https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html//apple_ref/doc/uid/TP30001200-SW1.

Building the AddressBook application

The ability to create a new contacts record is one of the most common things that we do in our everyday lives. This includes adding the contact details of family members, friends, colleagues, or even business contacts.

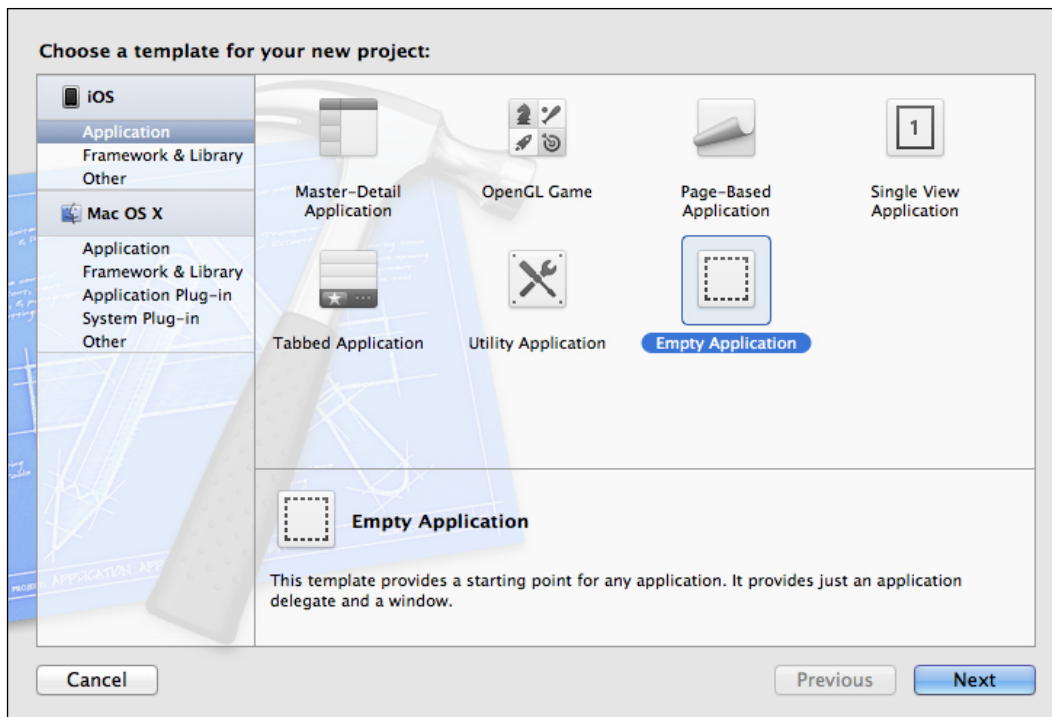
In this section, we will take a look at how to create an application, which will do just that, so it can run on an iOS device, enabling us to create new contacts, assign company details, address information, and additional notes or comments.

We will also be storing this information within a database using Core Data, and then have this information populated within a `UITableView` control that will provide us with the added ability to handle the management and presentation of data a lot more cleanly, and provide an added functionality of being able to delete items that have been previously added to the list.

Before we can proceed, we first need to create our AddressBook project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 2, Task Priorities – Building a TaskPriorities iOS App*, under the section named *Building the TaskPriorities app*.

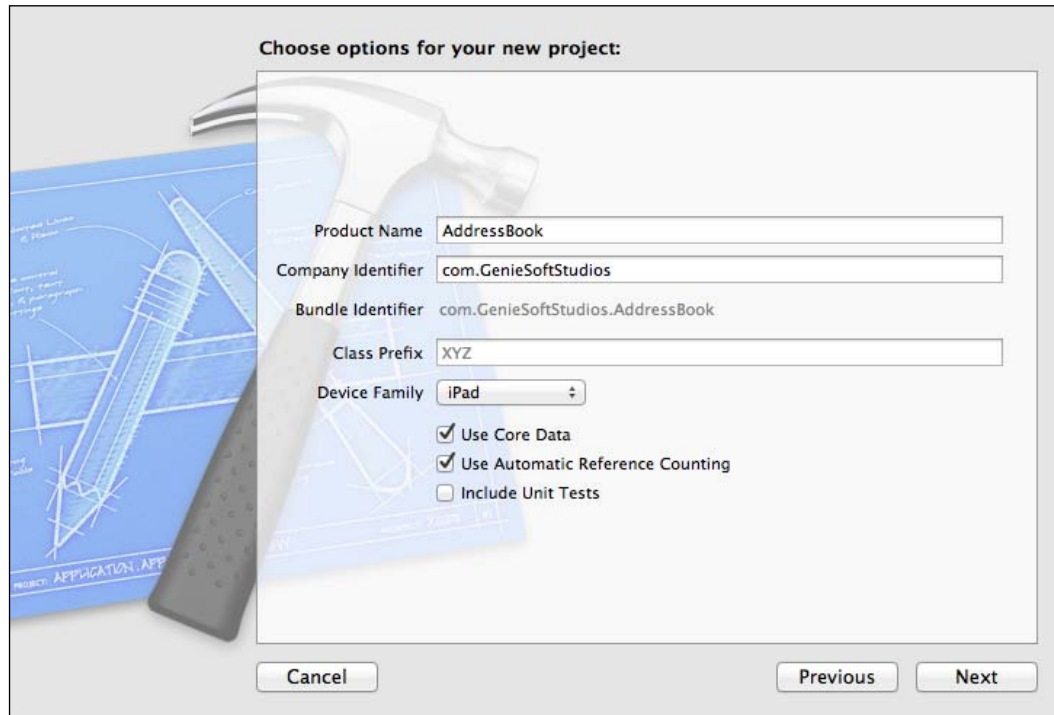
It is very simple to create this in Xcode. Just follow the steps listed here.

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode** project, or **File | New Project**.
3. Select the **Empty Application** template from the list of available templates.



4. Select **iPad** from under the **Device Family** drop-down.
5. Select the **Use Core Data** checkbox.
6. Select the **Use Automatic Reference Counting** checkbox.
7. Ensure that the **Include Unit Tests** checkbox has not been selected.

- Click on the **Next** button to proceed with the next step in the wizard.




- Enter in AddressBook as the name for your project.
- Click on the **Next** button to proceed with the next step of the wizard.
- Specify the location where you would like to save your project.
- Then, click on the **Save** button to continue and display the Xcode workspace environment.

Now that we have created our AddressBook project, we need to add the Game Kit framework to our project. This will enable us to communicate with other iOS devices over a Bluetooth network to send and receive information between them.

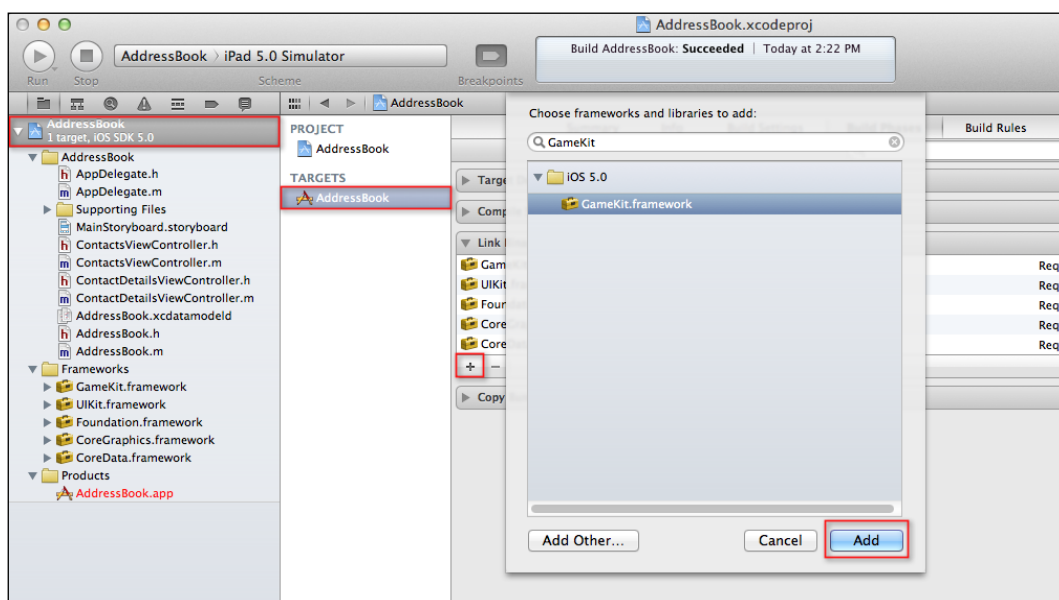
Adding the GameKit framework

As we mentioned previously, we need to add the Game Kit framework to our project that will enable us to perform to have the ability of transmitting information over a Bluetooth network to other iOS devices. To add the Game Kit framework, select **Project Navigator Group**, and then follow these simple steps as outlined here:

1. Click and select your project from **Project Navigator**.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Binary With Libraries** disclosure triangle.
5. Finally, use **+** to add the library you want.
6. Select `GameKit.framework` from the list of available frameworks.

 If you can't find the framework you are looking for, there is also the added ability to search for this directly, right from within the list of available frameworks.

If you are still confused how to go about adding the framework, follow this screenshot, which highlights the areas that you need to select (surrounded by a rectangle):

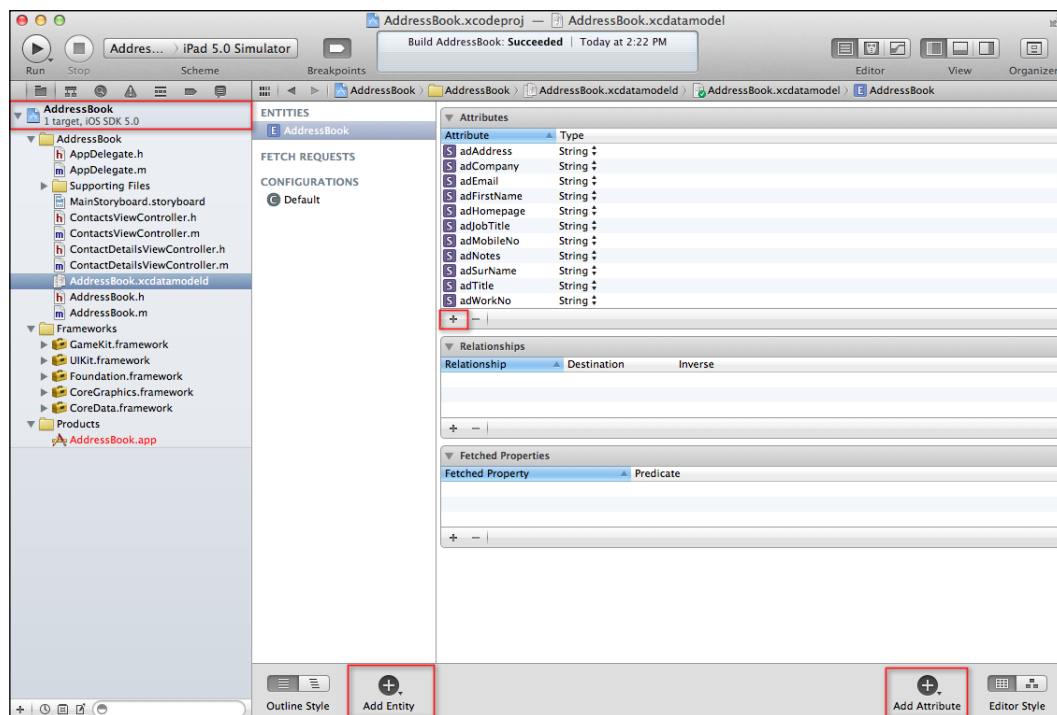


Now that we have added `GameKit.framework` into our project, we need to start building our user interface that will be responsible for allowing us to create and add new contacts directly into our list. One thing you will also notice is that we don't need to include `CoreData.framework` as it has been automatically added for us.

Building the Core Data model

The Core Data database model is stored within `AddressBook.xcdatamodel`, located within the `AddressBook` group within the **Project Navigator** window. This file will be used to define the database schema for our SQLite database, as we will be defining the entities (table) and the attributes (fields) that make up our address book.

Since we have selected the **Use Core Data** option, Xcode has automatically set up some important variables, and has created the file for us within our project.



Our next step is to create an entity and add the necessary attributes that will enable our application to write to these fields, hence storing this information within the database, so that it can be queried later. To create a new entity, follow the steps listed here:

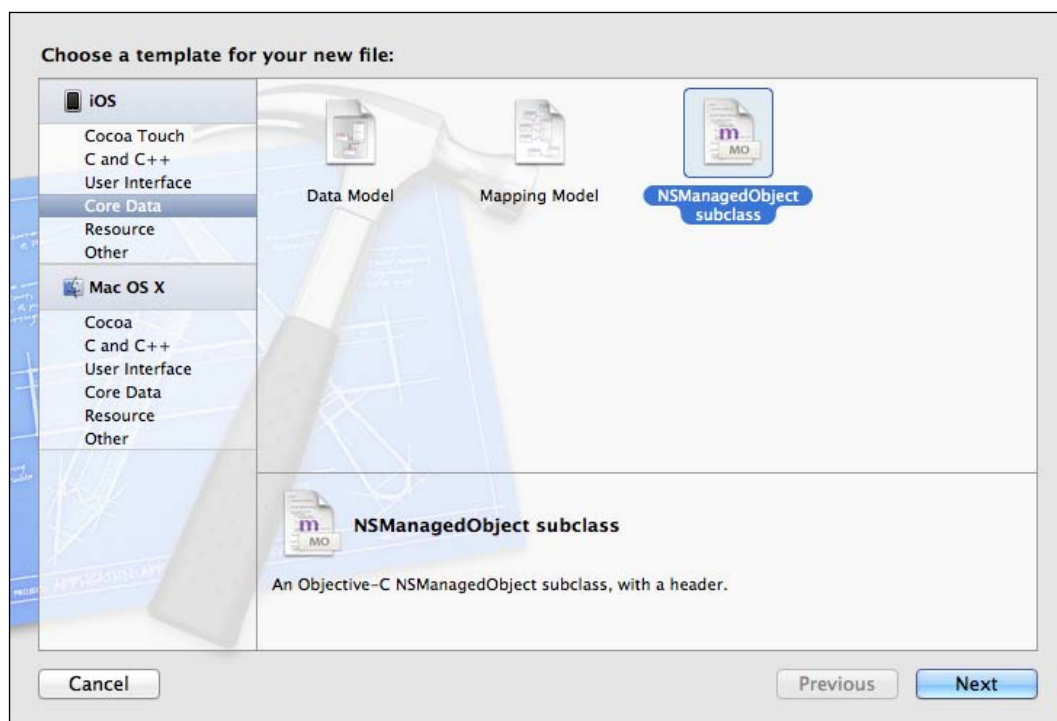
1. Select the `AddressBook.xcdatamodel` file from **Project Navigator**.
2. Click on the **+ Add Entity** button located at the bottom left-hand corner of the entity panel, and name this entity `AddressBook`.
3. Next, click on the **+ Add Attribute** button located at the bottom right-hand corner of the entity panel, or alternatively from the **Attributes** pane, and enter in `adTitle` as the value for the attribute.
4. Change the attribute type to **String** from the **Type** selection box.
5. Click on the **+ Add Attribute** button located at the bottom right-hand corner of the entity panel, and enter in `adFirstName` as the value for the attribute.
6. Change the attribute type to **String** from the **Type** selection box.
7. Click on the **+ Add Attribute** button located at the bottom right-hand corner of the entity panel, and enter in `adSurName` for the attribute.
8. Change the attribute type to **String** from the **Type** selection box.
9. Repeat steps 7 to 8 to add the remaining attributes for `adAddress`, `adCompany`, `adEmail`, `adHomepage`, `adJobTitle`, `adMobileNo`, `adNotes`, and `adWorkNo`.
10. Save your project using **File | Save**, as we have finished defining our database table schema.

So far we have created our `AddressBook` database model. Our next step is to take a look at how we can integrate and use the database within our application. In the next section, we will look at how to create the core data model files that will allow us to access the table definitions.

Creating our Core Data model files

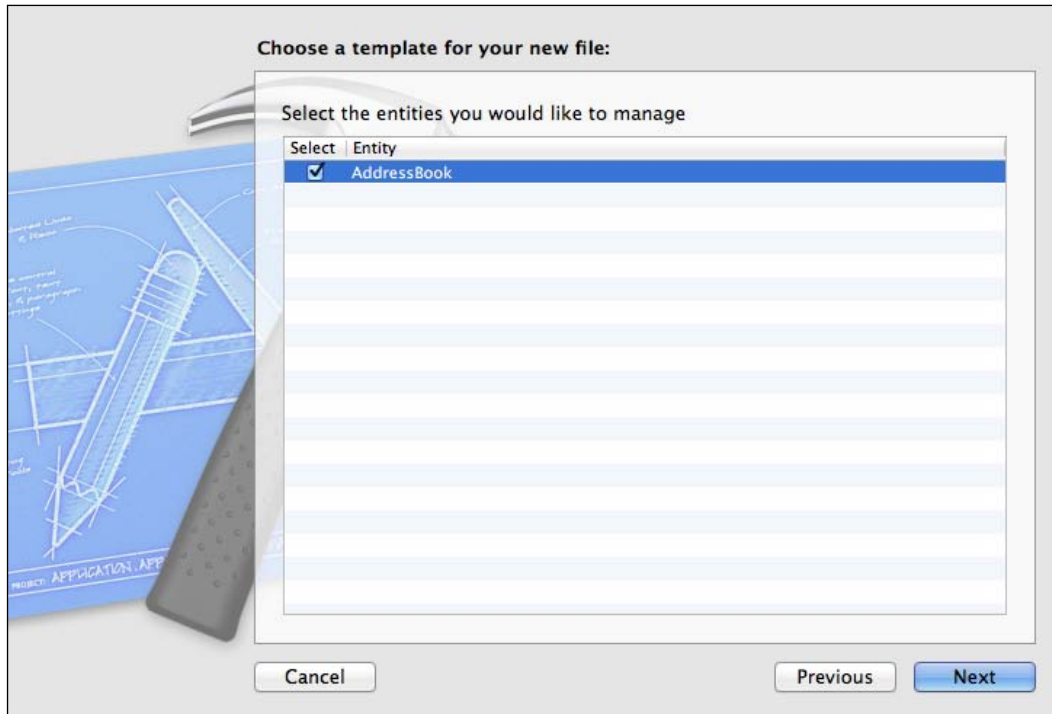
Before our application can start to use our AddressBook database, we need to create the entity class definitions that will define the variables the database store contains, so that we can access these through code.

1. From the AddressBook folder, select the AddressBook.xcdatamodel file from **Project Navigator**.
2. Choose **File | New | New File...** or press *Command + N*.
3. Select the `NSObject` subclass from the list of available templates.



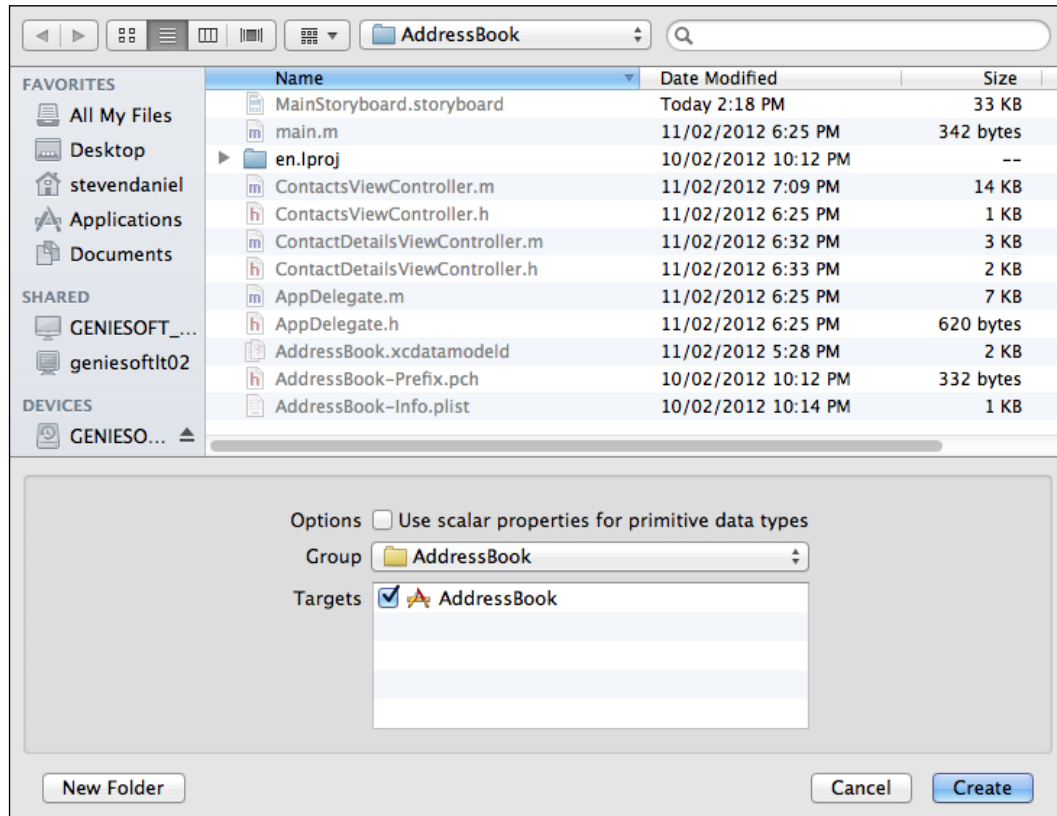
4. Click on the **Next** button to proceed with the next step within the wizard.
5. Click on the **Create** button to save the file to the folder location specified.

- Next, we need to define the entities for which we want to create the `NSManagedObject` classes.



- Select the `AddressBook` entity from the **Select the entities you would like to manage** list.

- Click on the **Next** button to proceed with the next step in the wizard.



- Ensure that the **Use scalar properties for primitive data types** option is not selected.
- Click on the **Create** button to generate the `NSManagedObject` class files.

You will notice that the wizard has created two new files for us — `AddressBook.m` and `AddressBook.h`. These files define the `NSManagedObject` class for the `AddressBook` entity that we created in the Core Data store.

They define the table schema fields, so that when we want to use the `AddressBook` class, we can access the attributes at runtime.

Let's take a quick look at the AddressBook class files, to see what the wizard generated for us.

1. Open the AddressBook.h interface file located within the AddressBook folder.

```
// AddressBook.h
// AddressBook
// Created by Steven F Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface AddressBook : NSObject

@property (strong, nonatomic) NSString * adFirstName;
@property (strong, nonatomic) NSString * adSurName;
@property (strong, nonatomic) NSString * adAddress;
@property (strong, nonatomic) NSString * adCompany;
@property (strong, nonatomic) NSString * adEmail;
@property (strong, nonatomic) NSString * adHomepage;
@property (strong, nonatomic) NSString * adJobTitle;
@property (strong, nonatomic) NSString * adMobileNo;
@property (strong, nonatomic) NSString * adNotes;
@property (strong, nonatomic) NSString * adTitle;
@property (strong, nonatomic) NSString * adWorkNo;

@end
```

From the preceding code snippet, we can see that the wizard has generated an AddressBook.h interface file containing each of our entity attribute fields with each being declared as an NSString object.

2. Next, open the AddressBook.m implementation file located within the AddressBook folder.

```
// AddressBook.m
// AddressBook
// Created by Steven F Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "AddressBook.h"

@implementation AddressBook

@dynamic adFirstName;
```



```
@dynamic adSurName;  
@dynamic adAddress;  
@dynamic adCompany;  
@dynamic adEmail;  
@dynamic adHomepage;  
@dynamic adJobTitle;  
@dynamic adMobileNo;  
@dynamic adNotes;  
@dynamic adTitle;  
@dynamic adWorkNo;  
  
@end
```

From the preceding code snippet, we can see that the wizard has generated an `AddressBook.m` implementation file that contains each of our entity attribute fields with each being declared as dynamic. This defines the entity attribute properties, so that they can be used when data is being written or retrieved from Core Data.



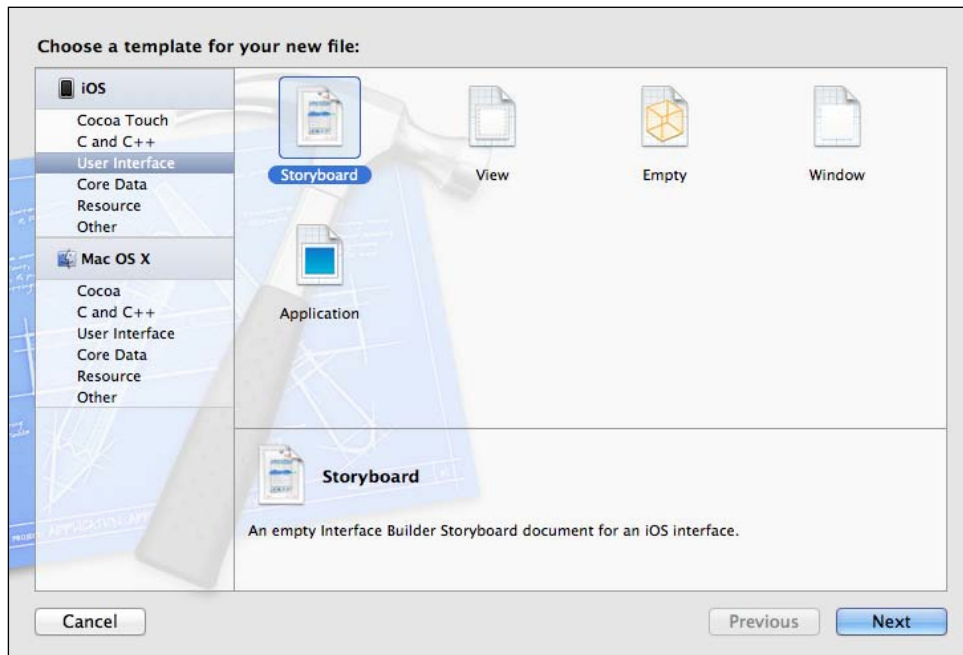
For more information about the dynamic data type, you can refer to the Apple Developer documentation at the following URL: https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CoreData/Articles/cdAccessorMethods.html#//apple_ref/doc/uid/TP40002154-SW9.

Adding the Storyboard screen

Now that we have created our AddressBook project, added the GameKit framework, and created the CoreData database, our next step is to include the Storyboard template as part of our AddressBook project. Unfortunately, the Storyboard template is not added as part of the **Empty Application** project template. This template provides you with a starting point for any application, and comes with an application delegate and a window.

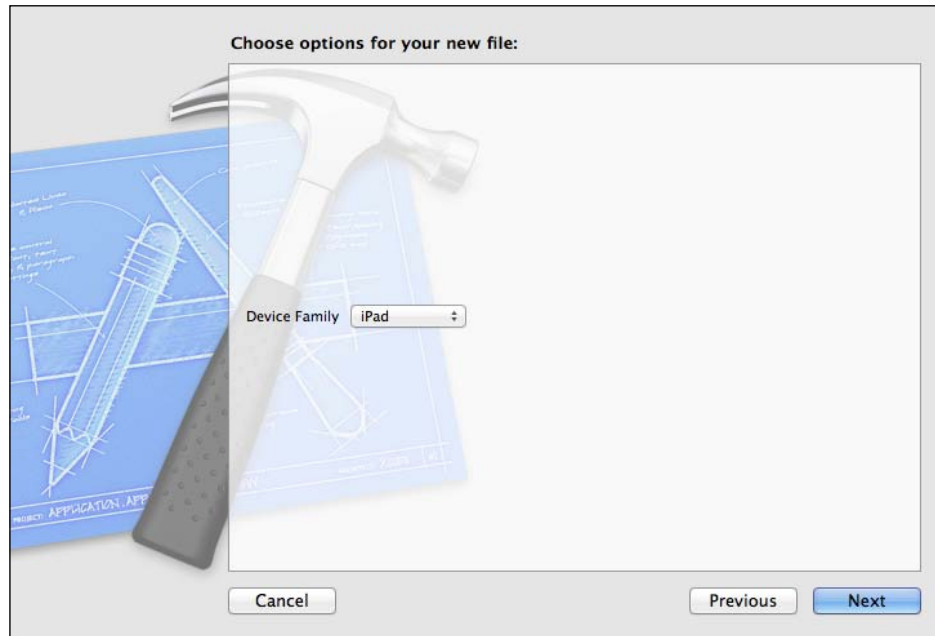
Follow these simple steps to see how to add the Storyboard template into your application.

1. From the **Project Navigator** window, select the `AddressBook` folder.
2. Choose **File | New | New File...** or press *Command + N*
3. Select **Storyboard** from the list of available templates.



4. Click on the **Next** button to proceed with the next step in the wizard.
5. Choose the **Storyboard** template from the list of available templates, located under the **User Interface** option within the **iOS** section.

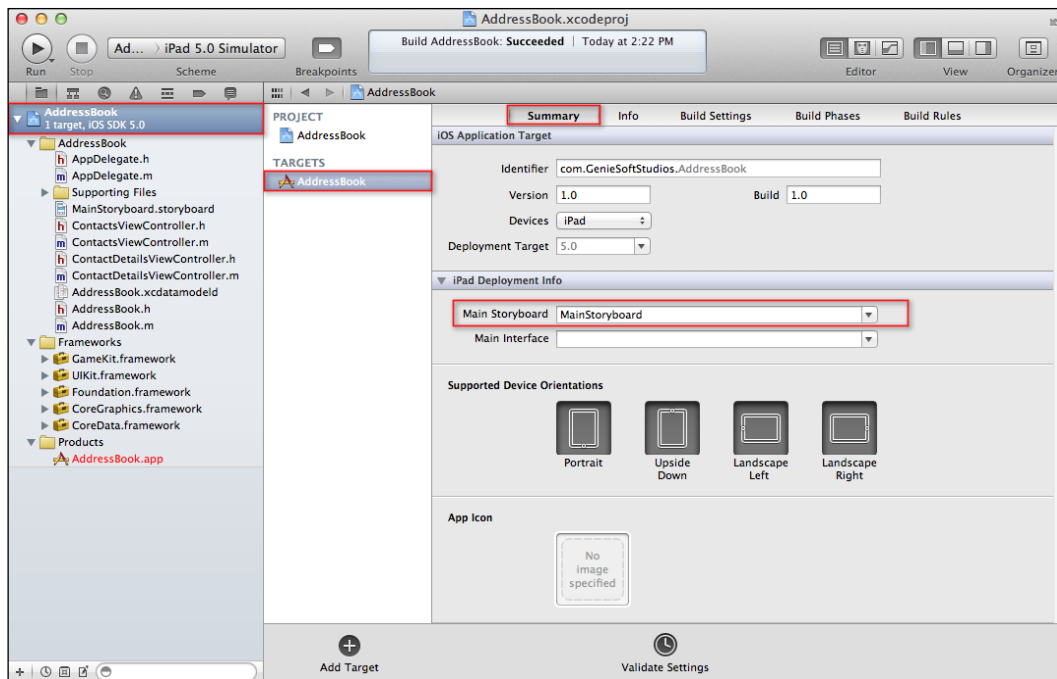
6. Click on the **Next** button to proceed with the next step in the wizard.



7. Ensure that you have selected **iPad** from under the **Device Family** dropdown.
8. Click on the **Next** button to proceed with the next step of the wizard.
9. Enter in MainStoryboard within the **Save As** field as the name of the file to be created.
10. Click on the **Create** button to save the file to the folder specified.

Now that we have created and added our Storyboard to our AddressBook application, our next task is to modify our project so that it is configured to use the Storyboard that we just created.

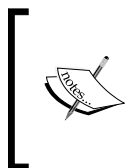
1. Click and select your project from the **Project Navigator** window.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Summary** tab.
4. Ensure that you select MainStoryboard from the **Main Storyboard** dropdown box.



- Next, open the `AppDelegate.m` implementation file from within **Project Navigator**, and modify the application's `didFinishLaunchingWithOptions` method, as shown in the following code snippet.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    return YES;
}
```

When using Storyboards, we don't need to create a new `UIWindow`, as this will create another white window and place this on top of the Storyboard. Now that we have added our Storyboard to our AddressBook application, our next step is to start building our main application.



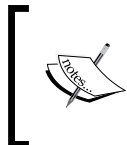
For more information on using Storyboards, you can refer to the Apple Developer Documentation located at the following URL: <https://developer.apple.com/library/mac/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/InterfaceBuilder/InterfaceBuilder>.

Creating the main application screen

Our next step is to build the user interface for our AddressBook application. The screens will consist of a Tab Bar controller, a Navigational controller, and View controllers. The Navigational controller enables us to create relationships between the other screens within the Storyboard and sets up the required connections, known as segues. A segue represents a transition from one screen to another.

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UITabBarController) **Tab Bar Controller** control, and add this to your view.

The Tab Bar controller comes pre-configured with two other view controllers, one for each tab that are represented by each image button at the bottom of the **Tab Bar Controller** control. The container relationship between each screen is represented within the Storyboard editor between the Tab Bar controller and the View controllers that it contains.



To refresh your memory on how to go about adding a UITabBarController to your Storyboard, you can refer to the section that we covered in **Chapter 2, Task Priorities – Building a TaskPriorities iOS App**, under the section named *Creating the main application screen*.

So far we have added a Tab Bar controller consisting of two view controllers that don't provide any functionality as yet. In this section, we will start building our user interface and add the required controls that will be used to process and hold our task items.

Adding a table control to hold the item data

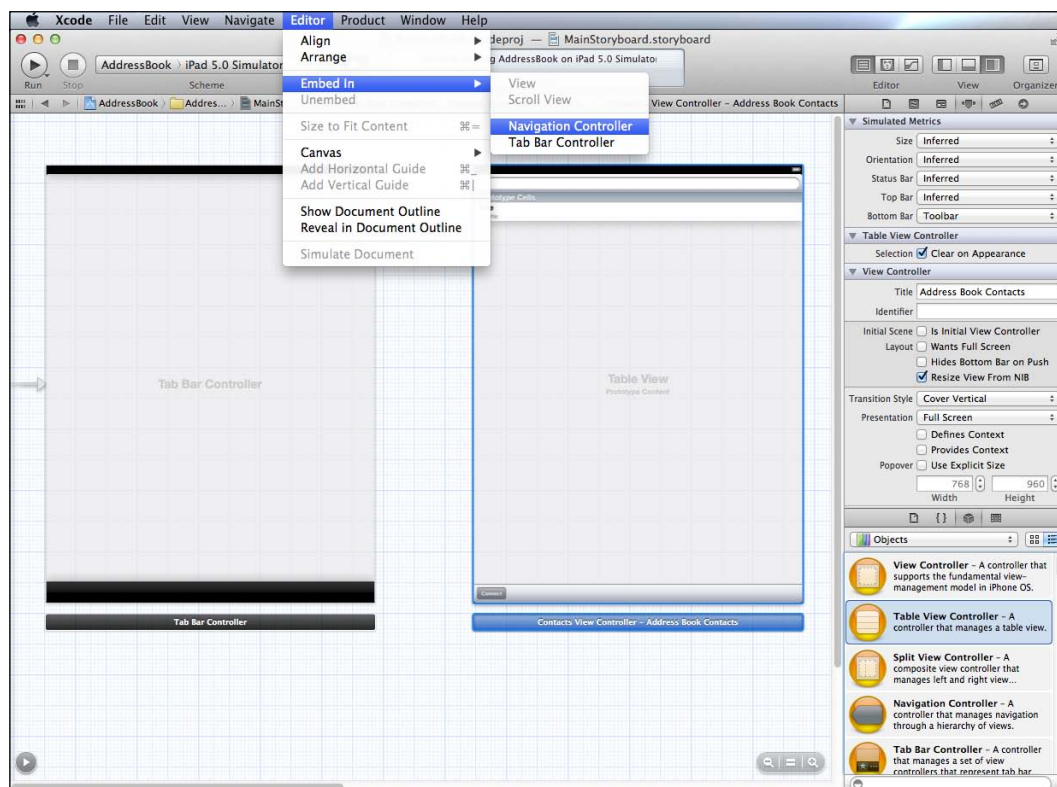
Our next step is to add a UITableViewController that will be used to hold and list our task entries. We will need to include a Navigation controller that will be used to navigate back and forth between the UITableViewController and itself.

To implement the UINavigationController is very simple, and we will take a look at how this is done in a few moments. For a discussion on what happens when a view gets displayed, we covered this in *Chapter 2, Task Priorities – Building a TaskPriorities iOS App*, under the section named *Adding a table control to hold the item data*.

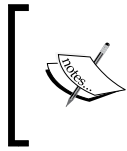
1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UITableViewController) **Table View Controller** control, and add this to our view.

- Next, we need to create a navigation controller between the Tab Bar controller and UITableViewController that we just added. There are two ways in which this can be achieved; you can either drag UINavigationController directly onto the view, or you can let Xcode do this for you automatically.
- Select UITableViewController that we just added, and then choose **Editor | Embed In | Navigation Controller** from the Editor menu.

If you have followed the steps correctly, your Storyboard should look similar to the following screenshot. If it doesn't look quite the same as mine, feel free to adjust yours.



You will notice that by embedding the Table View controller, our TableView controller automatically gets included within the navigation bar. The Storyboard editor automatically added the UINavigationController in there for us, because the scene will now be displayed inside the Navigation Controller's frame.



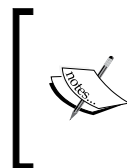
The `UINavigationController` screen is not a real `UINavigationController` object; the Storyboard Editor has simulated this for us. This can be seen from within **Attributes Inspector** as shown in the previous screenshot.

In the **Simulated Metrics** section, you will notice that **Inferred** has been set up as the default setting for each of the options; this is the default setting for storyboards. Inferred means that the scene will show a navigation bar when the Table View controller is inside a Navigation controller.

You have the ability to override any of these settings if you wanted to, but keep in mind that these are only here to help you when designing your screens. These aren't used during runtime, and are only available to show you how your screen will end up looking when it is run on the iOS device.

Our next step is to connect these scenes to our Tab Bar controller, so that the Table View controller will be the first screen to be displayed when it is run.

1. Select the **Tab Bar Controller** control, then hold down the *Ctrl* key and drag from the **Tab Bar Controller** control to the **Navigation Controller** control, and release the mouse.
2. Choose **Relationship – viewControllers** from the **Storyboard Segues** pop up.



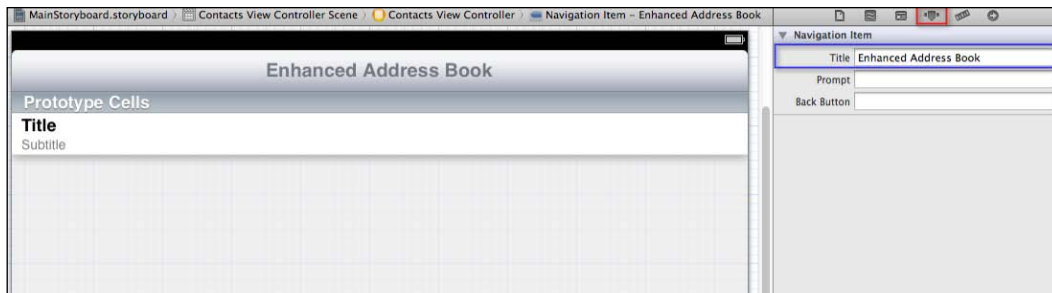
To refresh your memory on how to go about connecting the different scenes within your Storyboard, you can refer to the section that we covered in *Chapter 2, Task Priorities – Building a Task Priorities iOS App*, under the section named *Adding a table control to hold item data*.

You will notice that when we made a connection between the two view controllers, a new tab was added to the **Tab Bar Controller** control, named **Item**.

3. Next, we want to show the bottom toolbar within our Navigation Controller. Select the **Navigation Controller** control, and from the **Attributes Inspector** dialog, select the **Shows Toolbar** option.

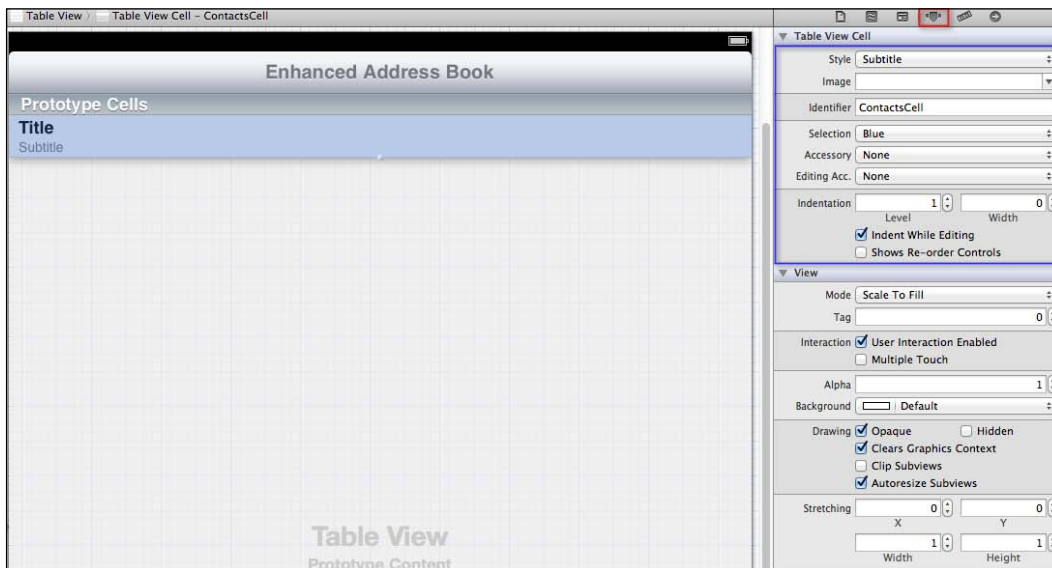
So far, we have linked up our Tab Bar controller and Navigation controllers, and have configured the properties required for the Navigation controller; our next step is to set up the properties on our Table View controller. Follow these simple steps:

1. Select the Table View controller that we just added previously.
2. Next, click on the toolbar located at the top of the View controller.
3. Then, from **Attributes Inspector**, change **Title** to read **Enhanced Address Book**, as shown in the following screenshot:



If you prefer, you can also double-click on the navigation bar and change its title. You may have noticed that since we added our Table View controller, Xcode gave us a warning.

As we have seen in the previous chapters, this warning message happens whenever you add a Table View Controller to a storyboard, and this is because it wants to use prototype cells as the default, but we haven't correctly configured this control yet. Let's take a look at the following screenshot:

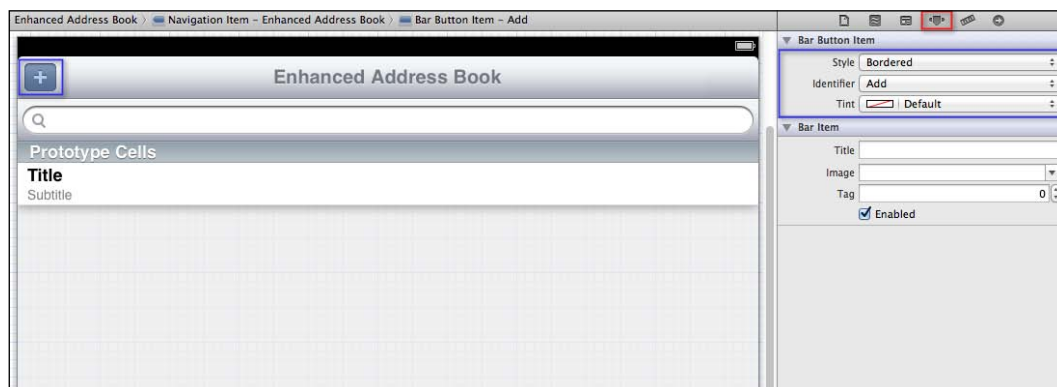


4. Click on the **Prototype** cell from the **Prototype Cells** section.
5. From the **Attributes Inspector** section, change **Style** to **Subtitle**.
This will change the cell's appearance to contain two labels.
6. Select the **Identifier** item and enter in `ContactsCell` as its unique identifier. You will notice that once this has been entered in, Xcode will stop complaining about the warning message that we received earlier on.
7. Set the **Accessory** attribute to show **None**.

Adding the Add button

Our next step is to add a button to our `UITableViewController`; this will be responsible for displaying an additional screen where we can create additional tasks. This can be achieved by following these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top left-hand corner of the navigation bar on the **Task Priorities** (`UITableViewController`) section of the **Table View Controller** screen that we added previously.
3. From the **Attributes Inspector** section, change **Identifier** to **Add**.
4. Then, change **Style** to **Bordered**.

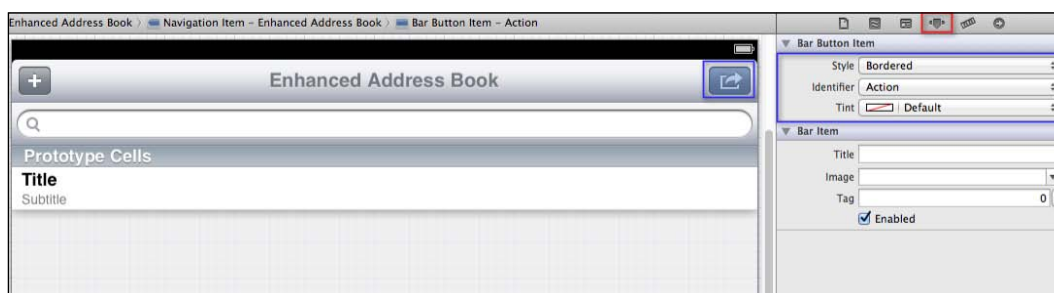


Now that we have added our **Add** button to our `Contacts View` controller, our next step is to add the **Action** button that will be responsible for sending our contact from one iOS device to another when the button is clicked. So let's proceed with the next section.

Adding the Action button

Now that we have added our button to add a new address contact record, our next step is to add another button to `UITableViewController`; this will be responsible for sending the contact information from one iOS device to another within the table view. This can be achieved by following these simple steps:

1. Select the **MainStoryboard.storyboard** file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top right-hand corner of the navigation bar on the **Enhanced Address Book** (`UITableViewController`) section of the **Table View Controller** screen that we added previously.
3. From the **Attributes Inspector** section, change the value of **Identifier** to **Action**.
4. Then, change the value of **Style** to **Bordered**.



Now that we have added our **Add** and **Action** buttons, as well as properly configured our Table View controller, we are ready to create our very own custom `UIViewController` subclass that will act as the data source for our table, so that it will know how many rows are to be displayed when it retrieves the address book information from our database.


1. Select the `AddressBook` folder, choose **File | New | New File...** or press *Command + N*.
2. Select the **Objective-C** class template from the list of templates.
3. Click on the **Next** button to proceed with the next step within the wizard.
4. Enter in `ContactsViewController` as the name of the file to create.
5. Ensure that you have selected `UITableViewController` as the type of subclass to create from the **Subclass of** dropdown.

6. Ensure that you have selected the **Targeted for iPad** option.
7. Click on the **Next** button to proceed with the next step of the wizard.
8. Click on the **Create** button to save the file to the folder location specified.

Now that we have added our `ViewController` class to our `AddressBook` application, our next step is to update the class of `UITableViewController` to use this class, instead of the default `UITableViewController` class.

1. Select the **MainStoryboard.storyboard** file from **Project Navigator**.
2. Select our **Enhanced Address Book** (`UITableViewController`) controller.
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `ContactsViewController`.

Our next step is to add a reference to the `NSManagedObjectContext` and `NSFetchedResultsController` objects that will provide us with all of the Core Data fetch-related functions we need to perform when populating our table view with data. These functions encapsulate the common functions that are associated with the table and the Core Data data-model. We will also create a `NSMutableArray` array property within our `ContactsViewController` interface file.

[ For more information about the `NSFetchedResultsController` object, you can refer to the Apple Developer documentation at the following URL: http://developer.apple.com/library/ios/#DOCUMENTATION/CoreData/Reference/NSFetchedResultsController_Class/Reference/Reference.html.]

1. Open the **ContactsViewController.h** interface file located within the `AddressBook` folder. Enter in the following code snippet:

```
// ContactsViewController.h
// AddressBook
// Created by Steven F Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import "AddressBook.h"

@interface ContactsViewController : UITableViewController
{
    NSManagedObjectContext *managedObjectContext;
}
```

```

    NSFetchedResultsController *fetchedResultsController;
    NSArray *fetchedObjects;
    IBOutlet UIBarButtonItem *btnSend;
}
@property (strong, nonatomic) NSManagedObjectContext
*managedObjectContext;
@property (strong, nonatomic) NSFetchedResultsController
*fetchedResultsController;
@property (strong, nonatomic) IBOutlet UIBarButtonItem *btnSend;

-(void)getContactDetails;

@end

```

As you can see, all we have done is created a reference to the `NSManagedObjectContext` and `NSFetchedResultsController` objects that will help us with managing the fetching, updating, and creating of records within the data store.

These objects also come with the added advantage and ability to handle validations and undo/redo functionality of records without having to write any additional code.

2. Next, open the `AppDelegate.m` implementation file, located within the `AddressBook` folder, and add the following highlighted code:

```

// AppDelegate.m
// AddressBook
//
// Created by Steven F Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "AppDelegate.h"
#import "ContactsViewController.h"

@implementation AppDelegate

@synthesize window = _window;

```

In the preceding code snippet, we need to import the `ContactsViewController.h` interface header file, as we will be referencing these when we set up our data source for `ContactsViewController` within our Storyboard.

3. Next, we need to change the `didFinishLaunchingWithOptions:` method located within the `AppDelegate.m` implementation file.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    UITabBarController *tabBarController =
        (UITabBarController *)self.window.rootViewController;

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

    contactsViewController.managedObjectContext =
        self.managedObjectContext;

    return YES;
}
```

In the preceding code snippet, we need to initialize the data source for `contactsViewController` using the `managedObjectContext` method. This will ensure that our controller has access to all of the required properties and methods required to add and retrieve the information from our data store.

Before this can happen, we must first cycle through each scene within our Storyboard in order to get a reference to `ContactsViewController`. This is so that we can initialize its data source, so that it points to our database. Next, we need to populate our address book information to our table view.

4. Open the `ContactsViewController.m` implementation file, and enter in the following highlighted code snippets:

```
// ContactsViewController.m
// AddressBook
//
// Created by Steven F Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "ContactsViewController.h"

@implementation ContactsViewController

@synthesize fetchedResultsController;
@synthesize managedObjectContext;
@synthesize btnSend;
```

- Next, we need to modify the `viewDidLoad` method located within the `ContactsViewController.m` implementation file. Enter in the following highlighted code snippet:

```
#pragma mark - View lifecycle
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Initialize and reload contacts.
    btnSend.enabled = NO;
    [self getContactDetails];
}
```

In the preceding code snippet, we need to initialize and set the action button to disabled and then call the `getContactDetails` method to populate the database object items to our table view.

- Next, open the `ContactsViewController.m` implementation file, and enter in the following code snippet for the `getContactDetails` method:

```
#pragma mark Populate our UITableView Controller with all records
in our database.
- (void)getContactDetails
{
    // Define our table/entity name to use
    NSEntityDescription *entity = [NSEntityDescription
        entityForName:@"AddressBook"
        inManagedObjectContext:managedObjectContext];

    // Set up the fetch request
    NSFetchRequest *fetchRequest=[[NSFetchRequest alloc]
        init];
    [fetchRequest setEntity:entity];

    // Define how we are to sort the records
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor
        alloc] initWithKey:@"adSurName" ascending:NO];
    NSArray *sortDescriptors = [NSArray
        arrayWithObject:sortDescriptor];
    [fetchRequest setSortDescriptors:sortDescriptors];

    // Define the FetchResults controller
    fetchedResultsController =
        [[NSFetchedResultsController
        alloc] initWithFetchRequest:fetchRequest
        managedObjectContext:managedObjectContext
        sectionNameKeyPath:nil cacheName:@"Root"];
```

```
// Fetch the records
NSError *error;
if (![self fetchedResultsController]
    performFetch:&error)
{
    // Something seriously went wrong, so notify the
    //user.
    NSLog(@"There was an error retrieving the address
        book contacts.");
}
// Return the number of rows to populate our
// Table View controller with.
fetchedObjects =
    fetchedResultsController.fetchedObjects;
[self.tableView reloadData];
}
```

In the preceding code snippet, we define the table entity that we want to use as our main data source and then create an instance to our `fetchRequest` object that will be used to hold the returned items. Next, we then specify that we would like to have the results sorted by surname in descending order.

7. Next, we initialize our `fetchResultsController` object in order for it to start retrieving the data from our database, then sort the result set returned by surname in descending order, then execute the record set, and then check for any errors that occurred, using the `performFetch` method.
8. Finally, we save the result set to our `fetchedObjects` property and then call the `reloadData` method on our table View control to redisplay the records in the table view.
9. Next, we need to modify the `viewDidAppear` method that is located within the `ContactsViewController.m` implementation file to refresh our Table View whenever the view is displayed. Locate the `viewDidAppear` method, and enter in the following highlighted code snippets:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    [self getContactDetails];
}
```

10. Next, we need to change the table view data source methods that are located within the `ContactsViewController.m` implementation file, and enter in the following highlighted code snippets:

```
#pragma mark - Table view data source
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
```

```

{
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
ion:(NSInteger)section
{
    // Return the number of rows in the section.
    return [fetchedObjects count];
}

```

From the preceding code snippet we can see that we set the number of table sections, and then had the `numberOfRowsInSection` method work out how many rows will exist in each section. This is achieved by using the `count` property of our `contactsArray` array object.

```

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
AtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"ContactsCell";
    AddressBook *address;

    // Check to ensure that we have items in our list.
    address = [contactsArray
                objectAtIndex:indexPath.row];

    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleSubtitle
                reuseIdentifier:CellIdentifier];
    }
    // Configure the cell...
    cell.textLabel.text = [NSString
                          stringWithFormat:@"%s %s, %s",
                          address.adTitle,
                          address.adSurName, address.adFirstName];
    cell.detailTextLabel.text = address.adCompany;

    return cell;
}

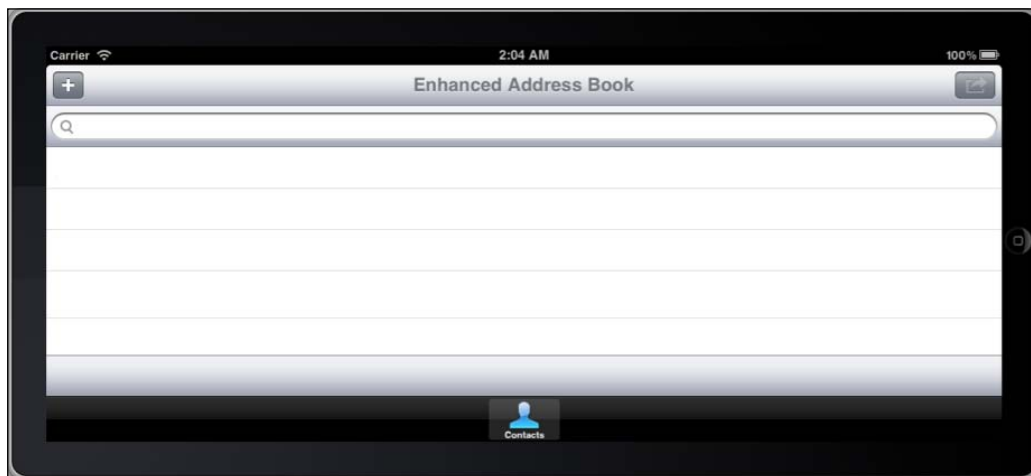
```


Finally, as you can see in the preceding code snippet, we supply the reuse identifier of the `TableViewController`'s cell that we set up previously, then assign each of the properties from our contacts array, and write it to the cell labels.



When you reference the reuse identifier as a parameter to the following method called `dequeueReusableCellWithIdentifier`, this will automatically make a new copy of the prototype, and return the object back to you.

11. Now that we have set up the data source correctly for our `TableViewController`, we can run our application by choosing **Product | Run** from the **Product** menu, or alternatively by pressing *Command + R* to see the `AddressBook` application running within the iOS Simulator, as shown in the following screenshot:



12. Now that we have successfully configured our data source for our list of contacts, we will see how we can navigate between screens within the Storyboard. We will learn about segues and the different types of views they can take on. We will look into static table view cells, as well as how to go about providing the ability for additional contact details to be added to the current list of contacts within our address book.

Navigating between screens using Storyboards

In this section, we will be adding more view controllers to our Storyboard to allow the flexibility of adding new contact detail information to our existing table view.

In order for us to transition between screens within our Storyboard, we need to create a connection, known as segue. Segues are defined as having the ability to only go one way; they cannot go back to the previous screen, unless a delegate class has been set up.

For our new screen, we will be creating a "modal" segue. A **modal segue** is a screen that becomes the active screen, which prevents the user from interacting with the underlying screen until they close the modal screen first.

To begin creating the **Add New Contact** screen, follow these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a new (`UITableViewController`) **Table View Controller** control, and add this to our Storyboard to the right of the **Enhanced Address Book** screen.
3. Next, select the `UITableViewController` that we just added, and then choose **Editor | Embed In | Navigation Controller** from the **Editor** menu.
4. Next, select the **+** button that we added previously, and hold down the *Control* key while dragging it to the new **Navigation Controller** and release the mouse button.
5. Finally, select **Modal** from the pop-up list of choices.

When you select **Modal** from the list of Storyboard Segues, a new connection will be placed between the **Contacts** screen and the Navigational controller. So, when you press the **+** button, a new Table View will slide up from the bottom of the screen.

Next, we need to specify an identifier for our Storyboard Segue. This will be responsible for handling the cancelling and saving methods when the **Add New Contact** form is closed.

1. Select the segue relationship that is located between the **Enhanced Address Book** screen and the Navigation controller for the **Add New Contact** screen.
2. Click on the **Attributes Inspector** button.
3. Change the **Identifier** property to `AddNewContact`.
4. Change the **Style** property to **Modal**.
5. Change the **Presentation** property to **Form Sheet**.
6. Change the **Transition** property to **Cover Vertical**.

Unfortunately, you won't be able to go back to the previous screen until we create a `UIViewController` subclass, same as what we did for `ContactsViewController`.

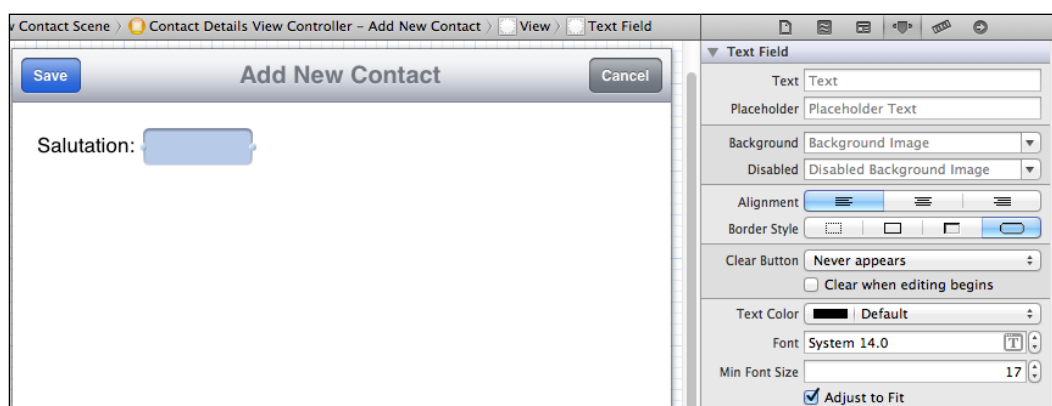
1. From the `AddressBook` folder, choose **File | New | New File...** or press *Command + N*.
2. Select the **Objective-C** class template from the list of templates.
3. Click on the **Next** button to proceed with the next step within the wizard.
4. Enter in `ContactDetailsViewController` as the name of the file to create.
5. Ensure that you have selected `UIViewController` as the type of subclass to create from the **Subclass** of dropdown list.
6. Ensure that you have selected the **Targeted for iPad** option.
7. Click on the **Next** button to proceed with the next step of the wizard.
8. Then, click on the **Create** button to save the file to the folder location specified.

Once you have done this, we need to update the class method of our previously added View controller to use our new `ViewController` subclass. Follow these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Click-and-select our newly added (`UIViewController`) to the right of the **Enhanced Address Book** table.
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `ContactDetailsViewController`.
4. Next, from the **Attributes Inspector** section, change the **Title** property to read **Add New Contact**.
5. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top left-hand corner of the navigation bar on the **Add New Contact** (`UIViewController`) section of the **View Controller** screen that we added previously.
6. From the **Attributes Inspector** section, change the value of **Identifier** to **Save**.
7. Then, change the value of **Style** to **Bordered**.
8. Next, from the **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top right-hand corner of the navigation bar.
9. From the **Attributes Inspector** section, change the value of **Identifier** to **Cancel**.
10. Change the value of **Style** to **Bordered**.

Our next step is to start building the screen that will allow us to record our contact details information, so that it can be saved to the **Enhanced Address Book** list.

1. Select the **Add New Contact** table view controller from within our Storyboard.
2. Next, drag a (UILabel) **Label Field** control onto the canvas.
3. Select **Attributes Inspector** for **Text Field**.
4. Set the **Text** field property to read **Salutation**:
5. Set the **Alignment** field property to **Left Justify**.
6. Next, drag a (UITextField) **Text Field** control next to the **Salutation** field that we added in the previous step.
7. Select **Attributes Inspector** for **Text Field**.
8. Set the **Alignment** field property to **Left Justify**.
9. Set the value of **Border Style** to **Rounded**.
10. Set the value of **Font** to **System 14.0**.

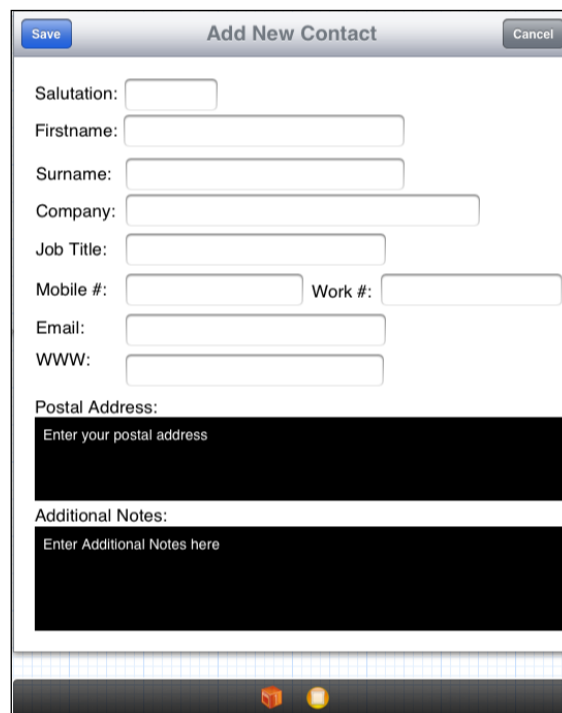


11. Ensure that you have unchecked the **Adjust to Fit** checkbox.
12. Repeat steps 2-11 to apply the same for each section, and create the following, along with their field types as shown in the table below:

FIELD NAME	FIELD TYPE
Firstname:	UITextField
Surname:	UITextField
Company:	UITextField
Job Title:	UITextField

FIELD NAME	FIELD TYPE
Mobile #:	UITextField
Work #:	UITextField
Email:	UITextField
WWW:	UITextField
Postal Address:	UITextView
Additional Notes:	UITextView

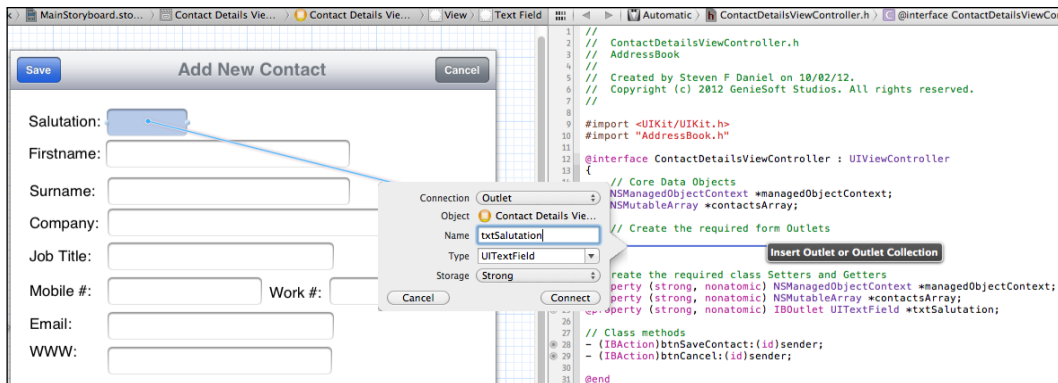
If you have followed the steps correctly, the completed **Add New Contact** screen should look similar to the following screenshot. Feel free to adjust yours accordingly if it doesn't.



The next step is to create the outlets for each of the fields that we previously added to our **Add New Contact** form.

1. Open **Assistant Editor** by choosing **Navigate | Open In Assistant Editor** or press *Option + Command + ,*.
2. Ensure that the `ContactDetailsViewController.h` interface file gets displayed.

3. Select the **Salutation** (UITextField), hold down the **Control** key, and drag it into the **ContactDetailsViewController.h** interface file.
4. Enter in `taskName` for the **Name** of the property to be created.
5. Choose **Strong** from the **Storage** dropdown.



6. Repeat steps 3 to 5, and add create the properties for the **Firstname**, **Surname**, **Company**, **Job Title**, **Mobile #**, **Work #**, **Email**, **WWW**, **Postal Address**, and **Additional Notes** fields.

Now that we have created the outlets and properties for each of our form fields, we need to start modifying our `ContactDetailsViewController.h` interface file.

1. Open the `ContactDetailsViewController.h` interface file, located within the `AddressBook` folder, and enter in the following highlighted code snippets:

```
// ContactDetailsViewController.h
// AddressBook
// Created by Steven F Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import "AddressBook.h"

@interface ContactDetailsViewController : UIViewController
{
    // Core Data Objects
    NSManagedObjectContext *managedObjectContext;

    // Create the required form Outlets
    IBOutlet UITextView *tvAdditionalNotes;
}
```

```
IBOutlet UITextView *tvPostalAddress;
IBOutlet UITextField *txtWebAddress;
IBOutlet UITextField *txtEmailAddress;
IBOutlet UITextField *txtWorkNo;
IBOutlet UITextField *txtMobileNo;
IBOutlet UITextField *txtJobTitle;
IBOutlet UITextField *txtCompany;
IBOutlet UITextField *txtSalutation;
IBOutlet UITextField *txtFirstname;
IBOutlet UITextField *txtSurname;
}

// Create the required class Setters and Getters
@property (strong, nonatomic) NSManagedObjectContext
*managedObjectContext;

@property (strong, nonatomic) IBOutlet UITextField *txtFirstname;
@property (strong, nonatomic) IBOutlet UITextField *txtSurname;
@property (strong, nonatomic) IBOutlet UITextField *txtSalutation;
@property (strong, nonatomic) IBOutlet UITextField *txtCompany;
@property (strong, nonatomic) IBOutlet UITextField *txtJobTitle;
@property (strong, nonatomic) IBOutlet UITextField *txtMobileNo;
@property (strong, nonatomic) IBOutlet UITextField *txtWorkNo;
@property (strong, nonatomic) IBOutlet UITextField
*txtEmailAddress;
@property (strong, nonatomic) IBOutlet UITextField *txtWebAddress;
@property (strong, nonatomic) IBOutlet UITextView
*tvPostalAddress;
@property (strong, nonatomic) IBOutlet UITextView
*tvAdditionalNotes;

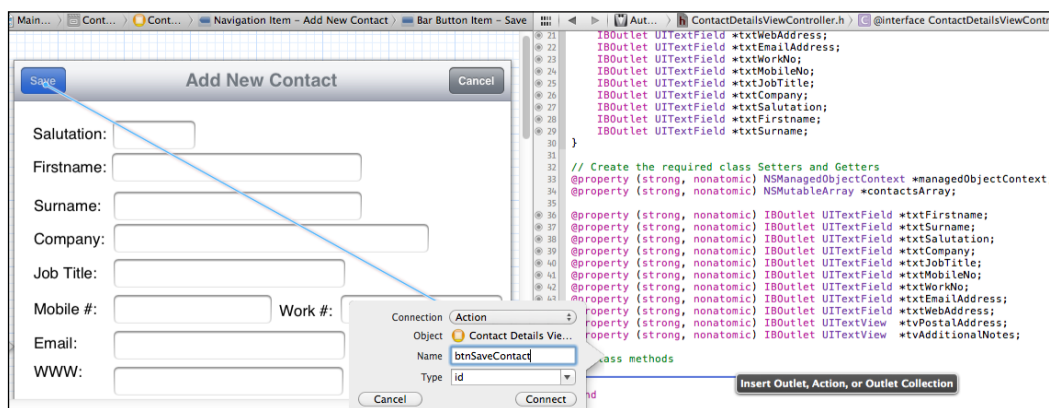
// Class methods
- (IBAction)btnSaveContact:(id)sender;
- (IBAction)btnCancel:(id)sender;

@end
```

In the preceding code snippet, we are setting up the Core Data objects that will enable us to save our new contact detail back to the data store when the user presses the **Save** button, as well as dismissing the view when the user presses the **Cancel** button to return us back to the **Enhanced Address Book** screen.

Now that we have created the `Outlet` events for each of our controls, we now need to create the associated `Action` events for those `Outlets`. Creating these actions allows an event to be fired when the button is pressed. To create an `Action`, follow these steps:

1. With the `ContactDetailsViewController.h` interface file still displayed to the right of the **Add New Contact** screen, select the **Save** (`UIBarButtonItem`) control, then hold down the **Control** key and drag it into the `ContactDetailsViewController.h` interface file.
2. Choose **Action** from the **Connection** dropdown for the connection to be created.
3. Enter in `btnSaveContact` for the **Name** of the method to be created.



4. Repeat steps 2 to 4 and hook up the **Cancel** button, creating the `Action` event `btnCancel`.

In the next section, we will take a look at building the functionality for our enhanced AddressBook application, as well as implementing the methods that will be used for our `Cancel` and `Save` buttons. These will be responsible for adding new contact information to our enhanced AddressBook list, and returning us back to the **Enhanced Address Book** screen.

Implementing the save record method

We are now ready to start implementing the method that will be responsible for saving the record when the user presses the **Save** button.

1. Open the **ContactDetailsViewController.m** implementation file, located within the AddressBook folder, and enter in the following code snippet:

```
- (IBAction)btnSaveContact:(id)sender {

    // Set a pointer to our AddressBook database table
    //schema
    AddressBook *address = (AddressBook
        *) [NSEntityDescription
        insertNewObjectForEntityForName:@"AddressBook"
        inManagedObjectContext:managedObjectContext];

    // Assign the form fields to each of their
    // managedObjectModel attributes.
    [address setAdTitle:txtSalutation.text];
    [address setAdFirstName:txtFirstname.text];
    [address setAdSurName:txtSurname.text];
    [address setAdCompany:txtCompany.text];
    [address setAdJobTitle:txtJobTitle.text];
    [address setAdMobileNo:txtMobileNo.text];
    [address setAdWorkNo:txtWorkNo.text];
    [address setAdEmail:txtEmailAddress.text];
    [address setAdHomepage:txtWebAddress.text];
    [address setAdAddress:tvPostalAddress.text];
    [address setAdNotes:tvAdditionalNotes.text];

    NSError *error;
    if (![managedObjectContext save:&error]) {
        // Display Error message stating that the record
        // could not be saved.
        UIAlertView *alertView = [[UIAlertView alloc]
            initWithTitle:@"Contact Details"
            message:@"There was a problem saving the contact
            details."
            delegate:self
            cancelButtonTitle:@"OK"
            otherButtonTitles:nil];

        [alertView show];
    }
    // Close our Contact Details view after we have
    //finished.
```

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

In the preceding code snippet, we create a managed object context that is then used to create a new managed object using the `AddressBook` entity description. The `setAd` getters and setters for each of the schema fields of the managed object are then called to set each of the attributes values of the managed object before finally the context is instructed to save the changes to the persistent store, with a call to the context's `save` method. Any errors detected during the save operation to our Core Data data model will be displayed within a `UIAlertView` dialog box.

2. We then add the new contact details object to our existing list of contacts and then refresh the table view, using the `reloadData` method to show that the new item was added, and then we close the **Add New Contact** screen.

Implementing the cancel method

Next, we need to implement the **Cancel** button. This will be responsible for closing the screen, and returning you back to the **Enhanced Address Book** table view when pressed.

Open the `ContactDetailsViewController.m` implementation file, located within the `AddressBook` folder, and enter in the following code snippet:

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

In the preceding code snippet, we use the `dismissViewControllerAnimated` method, which is only made available in iOS 5 and later. This method is used to close the current modal screen that was sent by our **Task Priorities** table view screen.

Implementing the delete row method

Next, we need to implement the delete method. This will be responsible for removing a contact detail record from the **Enhanced Address Book** table view.

Open the `ContactsViewController.m` implementation file, located within the `AddressBook` folder, and enter in the following highlighted code:

```
// Override to support editing the table view.
- (void)tableView:(UITableView *)tableView commitEditingStyle:(UITabl
eViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)
```

```
indexPath
{
    if (editingStyle ==
        UITableViewCellEditingStyleDelete)
    {
        // Get the item to delete from our row
        AddressBook *itemToDelete =
            [fetchResultsController
             objectAtIndex:indexPath];

        // Delete the item in Core Data
        [self.managedObjectContext
         deleteObject:itemToDelete];

        // Commit the deletion
        NSError *error;
        if (![self.managedObjectContext save:&error])
        {
            NSLog(@"There was a problem deleting the
                contact %@", [error domain]);
        }
        // Delete the row from the data source
        [tableView deleteRowsAtIndexPaths:[NSArray
            arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

In the preceding code snippet, we determine the type of action currently being performed within the table view, which is determined by the `UITableViewCellEditingStyle` class. We then compare against the `UITableViewCellEditingStyleDelete` constant variable, and if the condition is met, we remove the selected contact details at the selected row from our `AddressBook` database, then refresh the table view data source. If any errors have been detected during the removal process, are then logged out to the Debug window.

Implementing the `didSelectRowAtIndexPath` method

Next, we need to implement the `didSelectRowAtIndexPath:` method. This will be responsible for determining when a contact detail has been selected from the **Enhanced Address Book** table view.

In the preceding code snippet, we need to construct the contact detail information for the selected row. We use the row property of `indexPath` to seek inside our `contacts` array, and extract the record information. We then construct the record as a delimited string and assign it to our `itemSelected` variable, which will be used by the **Send** button.

One of the neat features that come as part of the iOS SDK is the Game Kit framework, which contains APIs that allow for communication over a Bluetooth network.

[97]

Open the `ContactsViewController.h` interface file, located within the AddressBook folder, and add the following highlighted code:

```
//
//  ContactsViewController.h
//  AddressBook
//
//  Created by Steven F Daniel on 10/02/12.
//  Copyright (c) 2012 GenieSoft Studios. All rights reserved.
//

#import <UIKit/UIKit.h>
#import <GameKit/GameKit.h>
#import "AddressBook.h"

@interface ContactsViewController : UITableViewController
<GKSessionDelegate, GKPeerPickerControllerDelegate,
UISearchBarDelegate>
{
    GKSession *currentSession;
    GKPeerPickerController *peerPicker;

    // Used for our selected table view item.
    NSString *itemSelected;
    NSArray *fetchedObjects;

    IBOutlet UIBarButtonItem *btnSend;
    IBOutlet UIBarButtonItem *btnConnect;
}

@property (nonatomic, strong) GKSession *currentSession;
@property (strong, nonatomic) IBOutlet UIBarButtonItem *btnConnect;
@property (strong, nonatomic) IBOutlet UIBarButtonItem *btnSend;

@property (strong, nonatomic) NSString *itemSelected;

- (IBAction)btnConnect:(id)sender;
- (IBAction)btnSend:(id)sender;

@end
```

In the preceding code snippet, we have declared a number of delegate objects that are responsible for handling the Bluetooth functionality.

We declared a `GKSessionDelegate` object that is used to represent an active session between two connected Bluetooth devices, and it allows for sending and receiving of data between the two. `GKPeerPickerControllerDelegate` provides a standard UI to let your application discover and connect to another Bluetooth device, and is the easiest way of connecting between devices.

The `UISearchBarDelegate` object is used to handle searches within our table view controller.

Implementing the connect method

Before you proceed to implement the `connect` method, ensure that you have added the **Connect** button to the `UITableViewController` view of **Enhanced Address Book**, and that you have created the necessary Outlets, Actions, and have synthesized these objects within the `ContactsViewController` interface and implementation files.

1. Open the `ContactsViewController.m` implementation file, located within the `AddressBook` folder, and enter the following code snippet:

```
- (IBAction)btnConnect:(id)sender {
    if ([btnConnect.title isEqualToString:@"Connect"])
    {
        [btnConnect setTitle:@"Disconnect"];
        peerPicker = [[GKPeerPickerController alloc] init];
        peerPicker.delegate = self;
        peerPicker.connectionTypesMask =
            GKPeerPickerConnectionTypeNearby;
        [peerPicker show];
        btnSend.enabled = YES;
    }
    else if ([btnConnect.title
        isEqualToString:@"Disconnect"])
    {
        [btnConnect setTitle:@"Connect"];
        [self.currentSession disconnectFromAllPeers];
        currentSession = nil;
        btnSend.enabled = NO;
    }
}
```

In the preceding code snippet, we check to see what is the current value of our button using the `isEqualToString` method. If the current state reads **Connect**, then we make a call to the `GKPeerPickerController` class, which displays a standard UI for which you can connect to another Bluetooth device.

Alternatively, if you click on **Disconnect**, we make a call to the `disconnectFromAllPeers` method from the **GKSession** object to close the connection between the two devices.

The following screenshot shows the process when two devices are trying to establish a connection:



The `connectionTypesMask` property indicates the types of connections that the user can choose from.

There are two types that are made available:

`GKPeerPickerConnectionTypeNearby` and `GKPeerPickerConnectionTypeOnline`. If you want to use Bluetooth communication, use the **`GKPeerPickerConnectionTypeNearby`** constant. Use of the **`GKPeerPickerConnectionTypeOnline`** constant indicates an internet-based connection.

When a Bluetooth connection has been detected between the two devices and the user has selected one of the items to connect to from the list of available devices, the `peerPickerController:didConnectPeer:toSession:` method is called.

2. Next, open the `ContactsViewController.m` implementation file located within the AddressBook folder, and enter the following code snippets:

```
#pragma mark Handle Bluetooth capabilities using the GameKit framework.
- (void)peerPickerController: (GKPeerPickerController *)picker
didConnectPeer: (NSString *)peerID toSession: (GKSession *)session
{
    self.currentSession = session;
    session.delegate = self;
    [session setDataReceiveHandler:self withContext:nil];
    picker.delegate = nil;
}
```

```

    [picker dismiss];
    btnSend.enabled = YES;
}

```

When the user has connected to the peer Bluetooth device, you save the GKSession object to the `currentSession` property, which enables you to use the GKSession object to communicate with the remote device.

```

- (void)peerPickerControllerDidCancel: (GKPeerPickerController *)
picker
{
    [btnConnect setTitle:@"Connect"];
    btnSend.enabled=NO;
    peerPicker = nil;
    peerPicker.delegate = nil;
    [peerPicker dismiss];
}

```

In the preceding code snippet, the `peerPickerControllerDidCancel:` method gets called whenever the user cancels out from the Bluetooth picker. Once this happens, we disable our **Send** button on our form, and close the `peerPicker` dialog box, releasing the memory allocated.

Whenever a device is connected or disconnected, there is a call made to the `session:didChangeState:` method. This method knows when a connection has been established or ended.

You then need to use the `state` property of the `GKPeerConnectionState` class and the constants to determine the type of connection.

```

- (void)session: (GKSession *)session peer: (NSString *)peerID didCha
ngeState: (GKPeerConnectionState) state
{
    NSString *GKPeerStateInfo;

    switch (state)
    {
        case GKPeerStateAvailable:
            GKPeerStateInfo = @"Wi-Fi is Available";
            break;

        case GKPeerStateUnavailable:
            GKPeerStateInfo = @"Wi-Fi is not Available";
            break;

        case GKPeerStateConnecting:
            GKPeerStateInfo = @"Establishing Connection";

```



```
        break;

        case GKPeerStateConnected:
            GKPeerStateInfo = @"Connection Successful";
            break;

        case GKPeerStateDisconnected:
            GKPeerStateInfo = @"Disconnected from Session";
            currentSession = nil;
            break;
    }

    // Display the current connection state.
    NSLog(@"Connection State: %@", GKPeerStateInfo);
}
```

The code preceding snippet shows how you can handle the different connection states, whenever a device connects or disconnects, using the state property. After we have determined the state type, we write the connection state to the console window using the NSLog statement.

In the next section, we will take a look at how to send a contact from one device to another over Bluetooth.

Please ensure that you have added the **Send** button to the UITableViewController view of **Enhanced Address Book**, and that you have created the necessary Outlets, Actions, and have synthesized these objects within the ContactsViewController interface and implementation files.

Implementing the Action button method

Once the two devices are connected via Bluetooth, you can start to send data between them. The data that is transmitted uses the NSData object, which is basically a bytes buffer, so you have the flexibility to define your own data format and send any given type of data.

1. Next, open the ContactsViewController.m implementation file located within the AddressBook folder, and enter the following code snippets:

```
- (IBAction)btnSend:(id)sender {
    // Convert an NSString object to NSData
    NSData *data;
    data = [itemSelected
            dataUsingEncoding:NSUTF8StringEncoding];
    [self.currentSession sendDataToAllPeers:data
     withDataMode:(GKSendDataReliable) error:nil];
}
```

In the preceding code snippet, we use the `sendDataToAllPeers:` method of the `GKSession` object to send data to the other device via the `NSData` object. We use a variable called `itemSelected`, which contains the contact address information to be sent.



When using the `GKSendDataReliable` constant, the `GKSession` object will continue to send the data until it successfully transmits the data or the connection times out. Alternatively, using `GKSendDataUnreliable` indicates that the `GKSession` object should send the data only once with no retry.

2. Next, we need to add a method that will be responsible for handling when data is received by the iOS device on the other end.
3. Next, open the `ContactsViewController.m` implementation file located within the `AddressBook` folder, and enter the following code snippet:

```
- (void) receiveData: (NSData *)data fromPeer: (NSString *)peer
inSession: (GKSession *)session context: (void *)context
{
    // Convert our NSData type to NSString
    NSString *strData;
    strData = [[NSString alloc] initWithData:data
        encoding:NSUTF8StringEncoding];

    // Split out our data array and place the contents
    // into an array.
    NSArray *stringComponents = [strData
        componentsSeparatedByString:@"~"];
    NSMutableArray *myArray = [[NSMutableArray alloc]
        initWithCapacity:1000];
    [myArray addObjectsFromArray:stringComponents];

    // Insert the passed record details into our database.
    AddressBook *address = (AddressBook
        *) [NSEntityDescription
        insertNewObjectForEntityForName:@"AddressBook"
        inManagedObjectContext:managedObjectContext];

    [address setAdTitle:[myArray objectAtIndex:0]];
    [address setAdFirstName:[myArray objectAtIndex:1]];
    [address setAdSurName:[myArray objectAtIndex:2]];
    [address setAdCompany:[myArray objectAtIndex:3]];
    [address setAdJobTitle:[myArray objectAtIndex:4]];
    [address setAdMobileNo:[myArray objectAtIndex:5]];
    [address setAdWorkNo:[myArray objectAtIndex:6]];
```

```
[address setAdEmail:[myArray objectAtIndex:7]];
[address setAdHomepage:[myArray objectAtIndex:8]];
[address setAdAddress:[myArray objectAtIndex:9]];
[address setAdNotes:[myArray objectAtIndex:10]];

NSError *error;
if (![managedObjectContext save:&error])
{
    // Display Error message stating that the record
    // could not be saved.
    UIAlertView *alertView = [[UIAlertView alloc]
        initWithTitle:@"Contact Details"
        message:@"There was a problem saving the contact
        details."
        delegate:self
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];

    [alertView show];
}
// Reload our contacts from our database
[self getContactDetails];
}
```

In the preceding code snippet, we use the `receiveData:fromPeer:inSession:context:` method of the `GKSession` object to handle the receiving of data that is sent. Since our data has been sent as an `NSData` delimited string, we first need to convert this into an `NSString` object before using the `componentsSeparatedByString` string class to split out each field individually, and place these into our `NSMutableArray` object variable, `myArray`. Next, we need to create a new `managedObjectContext` instance that will point to our `AddressBook` entity, and use the getter and setter methods of our `AddressBook` application's `NSManagedObject` to assign each array element from our `myArray` object, to each of the entity field attributes before the details are then written to the database. Any errors detected during the save operation to our Core Data data model will be displayed within a `UIAlertView` dialog box.

4. Finally, we reload our contacts from our database to repopulate our table view with the updated information.



For more information about the `NSMutableArray` object, you can refer to the Apple Developer documentation at the following URL: https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/Reference/Reference.html.

Finishing up

We just have a few more things to implement before we have a complete working application.

We will need to implement a couple more methods that will handle the transition between our **Enhanced Address Book** and our **Add New Contact** screens when the **+** button has been pressed, as well as adding the ability to search for records within our **Enhanced Address Book** Table View.

First, let's handle the transition between the **Enhanced Address Book** screen and the Navigation controller, to determine when a transition has been made on a segue within the Storyboard.

1. Open the **ContactsViewController.m** implementation file, located within the **TaskPriorities** folder, and enter in the following code snippet:

```
- (void)prepareForSegue: (UIStoryboardSegue *)segue sender: (id)
sender
{
    if ([segue.identifier
        isEqualToString:@"AddNewContact"])
    {
        UINavigationController *navigationController =
            segue.destinationViewController;
        ContactDetailsViewController
        *contactDetailsViewController =
            [[navigationController
            viewControllers] objectAtIndex:0];
        contactDetailsViewController.managedObjectContext =
            self.managedObjectContext;
    }
}
```

In the preceding code snippet, we use the `prepareForSegue:` method to determine whenever a transition to segue takes place, a check is required to be made on the identifier of the segue to determine if we are calling the **Add New Contact** screen.

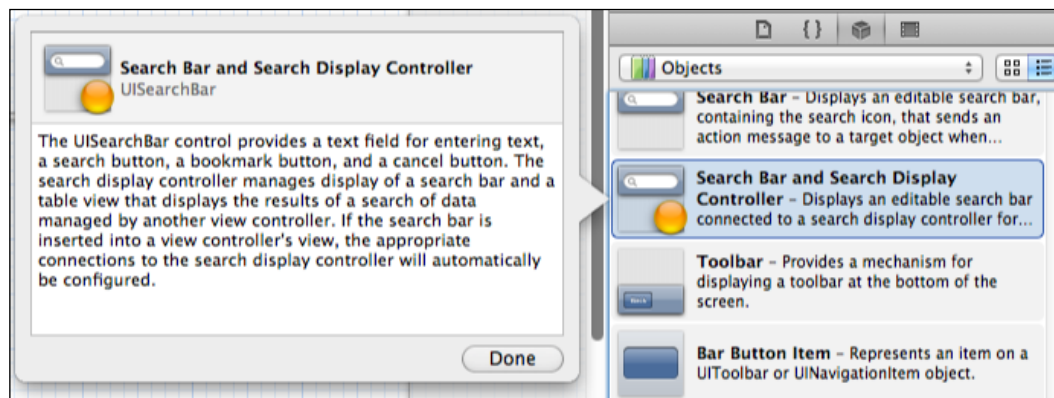
2. Next, we set `navigationController` of the segue to be the Navigation controller of the destination screen, and then cycle through each of the view controllers within the Navigation controller properties to get the `ContactDetailsViewController` instance, before finally setting the data source property of the form to be the currently active connection.

Implementing the search functionality

Now that we have sorted out our segue transition, we have one last feature to add to our enhanced AddressBook application. This will provide us with the ability to filter through our contacts list, and display only those names that have matching surnames.

Our next step is to add a **Search** bar to our `UITableViewController`; this will be responsible for filtering and narrowing down our contact list results. This can be achieved by following these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UISearchBar`) **Search Bar** and **Search Display Controller** control to the top of the navigation bar on the **Enhanced Address Book** (`UITableViewController`) section of the **Table View Controller** screen that we added previously.



Our next step is to create the Outlet for the Search bar. This will allow us to control what text is entered, and set or remove the control properties.

1. Open the **Assistant Editor** window by choosing **Navigate | Open in Assistant Editor**, or pressing *Option + Command + ,*.
2. Ensure that the **Enhanced Address Book** screen is displayed.

3. Select the **Search** bar (UISearchBar) control, then hold down the *Control* key and drag it into the `ContactsViewController.h` interface file.
4. Choose **Outlet** from the **Connection** dropdown for the connection to be created.
5. Enter in `destinationSearchBar` for the name of the **Outlet** property to be created.
6. Choose **Strong** from the **Storage** dropdown.

Now that we have created the outlet event for our `UISearchBar`, we need to start adding the additional content to our `ViewController` class that will provide us with the ability to filter our list.

1. Next, open the `ContactsViewController.h` interface file, located within the `AddressBook` folder, and enter the following highlighted code:

```
//
//  ContactsViewController.h
//  AddressBook
//
//  Created by Steven F Daniel on 10/02/12.
//  Copyright (c) 2012 GenieSoft Studios. All rights reserved.
//

#import <UIKit/UIKit.h>
#import <GameKit/GameKit.h>
#import "AddressBook.h"

@interface ContactsViewController : UITableViewController
<GKSessionDelegate, GKPeerPickerControllerDelegate,
UISearchBarDelegate>
{
    IBOutlet UISearchBar *destinationSearchBar;
}

@property (strong, nonatomic) IBOutlet UISearchBar
*destinationSearchBar;

@end
```

In the preceding code snippet, we have extended our `ContactsViewController` class to use the `UISearchBarDelegate` class protocols, so that we have access to its properties and methods.

2. Next, open the `ContactsViewController.m` implementation file located within the `AddressBook` folder, and enter the following code:

```
#pragma mark UISearchBar Delegates
- (void) searchBarTextDidBeginEditing: (UISearchBar *) searchBar
{
    // Only show the Search Bar's cancel button
    // while in edit mode.
    destinationSearchBar.showsCancelButton = YES;
    destinationSearchBar.autocorrectionType =
        UITextAutocorrectionTypeNo;
}
```

In the preceding code snippet, all we are doing is changing the appearance of the **Search** bar when a user taps in it. Next, we specify to show the **Cancel** button when the user is in **Edit** mode, and then turn off the **Auto Correction** feature.

```
- (void) searchBarTextDidEndEditing: (UISearchBar *) searchBar
{
    // Hide our Search Bar's cancel button when
    // not in edit mode.
    destinationSearchBar.showsCancelButton = NO;
}
```

In the preceding code snippet, we hide the **Cancel** button of the **Search** bar when the user has finished with editing.

```
- (void) searchBarCancelButtonClicked: (UISearchBar *) searchBar
{
    // Reload our contact details
    [self getContactDetails];
}
```

In the preceding code snippet, we call our `getContactDetails` method to get the updated records from the database, and populate this to our table view control.

```
- (void) searchBarSearchButtonClicked: (UISearchBar *) theSearchBar
{
    // We use an NSPredicate combined with the
    // fetchedResultsController to perform the search
    if (![destinationSearchBar.text isEqualToString:@""])
    {
        NSPredicate *predicate = [NSPredicate
            predicateWithFormat:@"adSurName contains[cd] %@",
            self.destinationSearchBar.text];
    }
```

```

        [fetchedResultsController.fetchRequest
        setPredicate:predicate];
    }
    else
    {
        // We have hit the cancel button, so just reload
        // our TableView
        [destinationSearchBar resignFirstResponder];
        [self.tableView reloadData];
        return;
    }
    NSError *error = nil;
    if (![self fetchedResultsController
        performFetch:&error])
    {
        // Handle the error that was caught by the exception
        NSLog(@"Unresolved error %@, %@", error, [error
            userInfo]);
        exit(-1);
    }
    // Return the number of rows to populate our
    // Table View controller with.
    fetchedObjects =
        fetchedResultsController.fetchedObjects;

    // reload the TableView Controller and hide the
    // keyboard.
    [destinationSearchBar resignFirstResponder];
    [self.tableView reloadData];

    NSString *searchResults = [[NSString alloc]
    initWithFormat:@"%d matching record(s) found.",
    [fetchedObjects count]];
    UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:@"Search Results"
    message:searchResults
    delegate:self
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil];

    [alertView show];
}

```


In the preceding code snippet, we cycle through our data source and select those objects that have the occurrence of the search string. We then reload our table view with the search data that was returned to be matching, and then display the total number of matching records within a UIAlertView dialog box.

If the **Search** criterion is empty, we perform a comparison using the `isEqualToString` method and check to see if the string is empty. We then resign the keyboard and reload all contacts from our data-model.

In our next part, we need to modify our `tableView:numberOfRowsInSection:section` method to handle the search facility in order to display the correct number of rows.

1. Open the `ContactsViewController.m` implementation file, located within the AddressBook folder, and enter the following code:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return [fetchedObjects count];
}
```

In the preceding code snippet, we need to determine whenever a search has been applied, and filter the list accordingly for those items. This is achieved by using the `fetchedObjects count` property.

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPathIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"ContactsCell";
    AddressBook *address;

    // Get each item from our resultset and add this to
    // the TableView.
    address = [fetchResultsController
        objectAtIndex:indexPath.indexPath];

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier];
    }
}
```

```

// Configure the cell...
cell.textLabel.text = [NSString
    stringWithFormat:@"%@@ %@, %@", address.adTitle,
    address.adSurName, address.adFirstName];
cell.detailTextLabel.text = address.adCompany;

return cell;
}

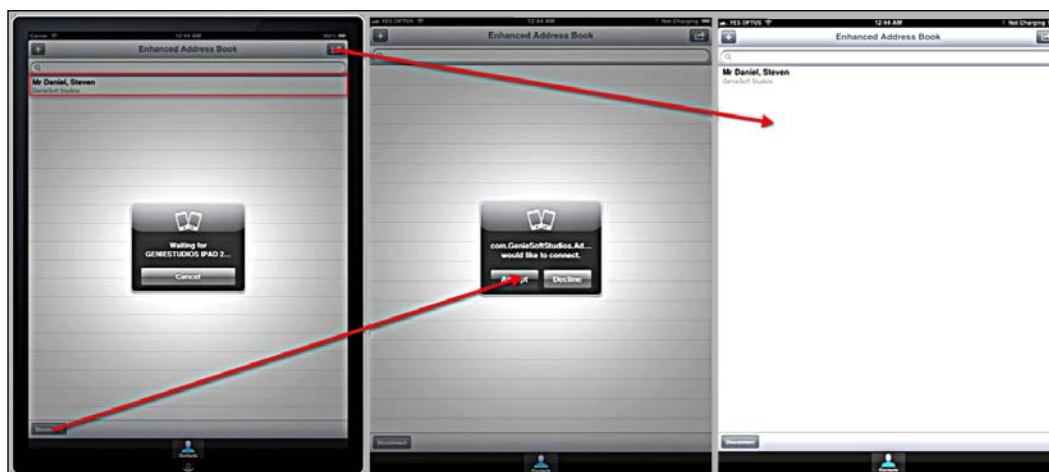
```

In the preceding code snippet, we again need to determine whenever a search has been applied, and filter the list accordingly for only those items. This is to ensure that when we select an item from the list, we are using the correct one.

Congratulations, we have successfully implemented the methods for our enhanced AddressBook application. Next, build and run the application by choosing **Product** | **Run** from the **Product** menu, or alternatively by pressing *Command + R* keys, and deploy the AddressBook application onto two different devices.

One thing to keep in mind when working with Core Data is that if you try to add a new field to the data-model schema, your application will crash. You will need to regenerate the `NSManagedObject` files, and then reset the simulator or delete the application from the iOS device.

Next, connect each of the devices using Bluetooth, select a contact from the **Contact** list, and click on the **Send** button to have it submitted to the other device. The following screenshot shows the application running within the iOS Simulator and the other running on an iOS device with an Internet connection and Bluetooth connectivity.



From the preceding screenshot, we can see that when the **Connect** button is pressed, a pop up is displayed that contains a list of nearby devices from which the user can choose from. Once the user has selected a device, confirmation is required from that device to allow the iOS Simulator to establish a connection with it.

Once a connection has been established, you are free to select an item from the list, and then click on the **Send** button to have this information transmitted to the other device, as shown in the last window to the right. As you can see, incorporating Bluetooth connectivity into your applications allows you to create some stunning games, so that you can play against your friends, or create some fantastic business applications and have documents or images transmitted between your colleagues.

Summary

In this chapter, we learned how to create an enhanced AddressBook application, using the Core Data framework to separate our data model from the rest of the application using the Model-View-Controller design. We visually designed our AddressBook entity, which contained the attributes representing each contact's name, address, job title, and so on, and programmatically interacted with the data model using the `NSManagedObject`, and the `NSFetchedResultsController` objects, which allowed us to fetch information from the data store and populate this within our `UITableView`.

We used the Game Kit framework to transfer the selected contact information among multiple iOS devices using Bluetooth, and used the `GKPeerPickerController` class that enabled the user to choose a nearby iOS device to which the contact should be transferred. We then used the `GKSession` object, which enabled us to transmit this information as an `NSData` object representing the contact information.

In the next chapter, we will look at how to create an application that will allow us to interact with the iOS device and determine its battery level, and use the Core Graphics framework to represent the battery level as a colored bar.

5

BatteryMonitor Application

The `BatteryMonitor` application allows you to monitor the state and battery levels of your iOS device using the APIs that come with the iOS SDK. Each iOS device represents a unique set of properties that include the device's current physical orientation, its model name, and its battery state. It also provides access to the onboard hardware.

In this chapter, we will be taking a closer look at how we can use the Core Graphics framework to create and draw a gauge that will be used to present and visualize the total amount of battery life remaining on the iOS device, and then start to design the user interface for our app.

We will look at how to create an instance of our `UIViewController` that will be used to create a custom `BatteryGauge` class. This class will be used to visually represent the total amount of battery remaining on the device. We will then take a look at how we can use the `MFMailComposeViewController` class to send an e-mail message when the total amount of battery life left is less than 20 percent full.

In this chapter we will:

- Get an overview of the technologies that we will be using
- Learn how to add the Core Graphics and `MessageUI` frameworks
- Walk through the steps to build the `BatteryMonitor` application
- Implement the `BatteryGauge` class to measure battery levels
- Implement a method to handle monitoring of the battery
- Implement a method to change the battery color
- Implement a method to change the number of battery bars displayed
- Implement a method to send an e-mail alert when the battery is low

We have an exciting project ahead of us; so let's get started.

Overview of the technologies

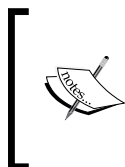
The `BatteryMonitor` application makes reference to two very important frameworks to allow for drawing of graphics to the iOS device's view, as well as composing and sending of e-mail messages, directly within the application.

In this chapter, we will be making use of the Core Graphics framework that will be responsible for handling the creation of our battery gauge to allow the contents to be filled based on the total amount of battery life remaining on the device. We will then use the `MessageUI` framework that will be responsible for composing and sending e-mails whenever the application has determined that the battery levels fall below the 20 percent threshold. This is all handled and done directly within our app.

We will make use of the `UIDevice` class that will be used to gather the device information for our iOS device. This class enables you to recover device-specific values, including the model of the iOS device that is being used, the device name, and the OS name and version. We will then use the `MFMailComposeViewController` class object to directly open up the e-mail dialog box within the application.

The information that you can retrieve from the `UIDevice` class is shown in the following table:

Type	Description
System name	This returns the name of the operating system that is currently in use. Since all current generation iOS devices run using the same OS, only one will be displayed; that is iOS 5.1.
System version	This lists the firmware version that is currently installed on the iOS device; that is, 4.3, 4.31, 5.01, and so on.
Unique identifier	<p>The unique identifier of the iOS device generates a hexadecimal number to guarantee that it is unique for each iOS device, and does this by applying an internal hash to several of its hardware specifiers, including the device's serial number.</p> <p>This unique identifier is used to register the iOS devices at the iOS portal for provisioning of distribution of software apps. Apple is currently phasing out and rejecting apps that access the Unique Device Identifier on an iOS device to solve issues with piracy, and has suggested that you should create a unique identifier that is specific to your app.</p>
Model	The iOS model returns a string that describes its platform; that is, iPhone, iPod Touch, and iPad.
Name	This represents the assigned name of the iOS device that has been assigned by the user within iTunes. This name is also used to create the localhost names for the device, particularly when networking is used.



For more information on the `UIDevice` class, you can refer to the Apple Developer Documentation that can be found and located at the following URL: https://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UIDevice_Class/Reference/UIDevice.html.

Building the BatteryMonitor application

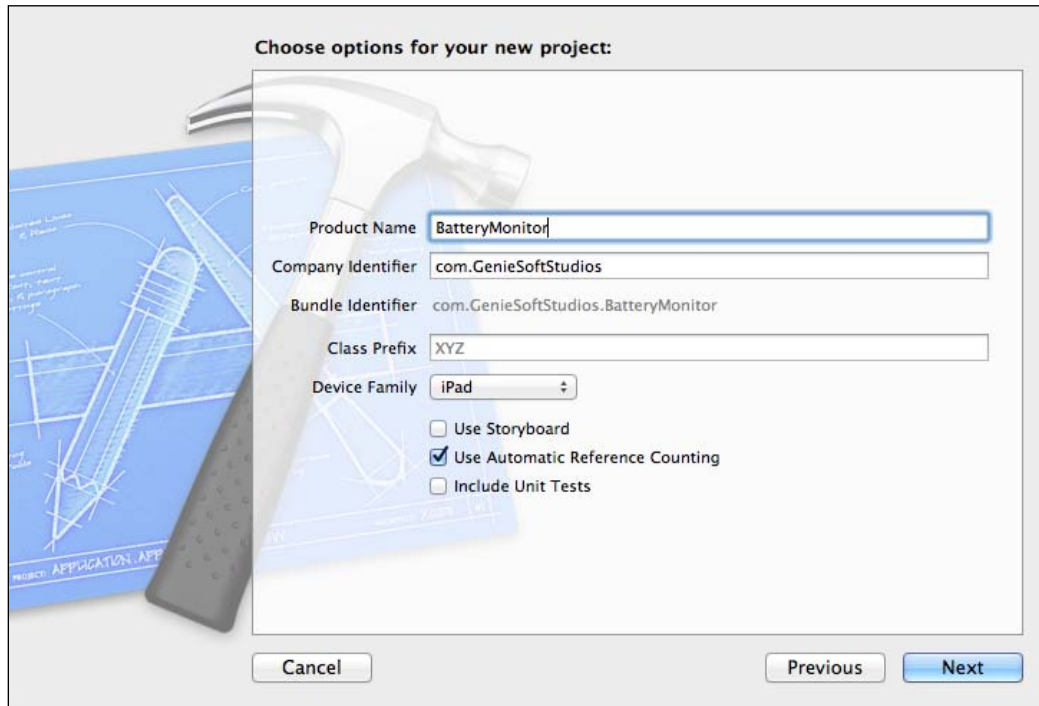
Monitoring battery levels is a common thing that we do in our everyday lives. The battery indicator on the iPhone/iPad lets us know when it is time for us to recharge our iOS device. In this section, we will look at how to create an application that can run on an iOS device to enable us to monitor battery levels on an iOS device, and then send an e-mail alert when the battery levels fall below the threshold.

Before we can proceed, we first need to create our `BatteryMonitor` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 3, VoiceRecorder App – Audio Recording and Playback*, under the section named *Building the VoiceRecorder app*.

It is very simple to create this in Xcode. Just follow the steps listed here.

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the **Single View Application** template from the list of available templates.
4. Select **iPad** from under the **Device Family** drop-down list.
5. Ensure that the **Use Storyboard** checkbox has not been selected.
6. Select the **Use Automatic Reference Counting** checkbox.
7. Ensure that the **Include Unit Tests** checkbox has not been selected.

- Click on the **Next** button to proceed with the next step in the wizard.



- Enter in BatteryMonitor as the name for your project.
- Then click on the **Next** button to proceed with the next step of the wizard.
- Specify the location where you would like to save your project.
- Then, click on the **Save** button to continue and display the Xcode workspace environment.

Now that we have created our BatteryMonitor project, we need to add the MessageUI framework to our project. This will enable us to send e-mail alerts when the battery levels fall below the threshold.

Adding the MessageUI framework to the project

As we mentioned previously, we need to add the MessageUI framework to our project to allow us to compose and send an e-mail directly within our iOS application, whenever we determine that our device is running below the allowable percentage.

To add the `MessageUI` framework, select **Project Navigator Group**, and follow the simple steps outlined here:

1. Click and select your project from **Project Navigator**.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Binary With Libraries** disclosure triangle.
5. Finally, use **+** to add the library you want.
6. Select `MessageUI.framework` from the list of available frameworks.

Now that we have added `MessageUI.framework` into our project, we need to start building our user interface that will be responsible for allowing us to monitor the battery levels of our iOS device, as well as handle sending out e-mails when the battery levels fall below the agreed threshold.

Creating the main application screen

The `BatteryMonitor` application doesn't do anything at this stage; all we have done is created the project and added the `MessageUI` framework to handle the sending of e-mails when our battery levels are falling below the threshold.

We now need to start building the user interface for our `BatteryMonitor` application. This screen will consist of a View controller, and some controls to handle setting the number of bars to be displayed, as well as whether the monitoring of the battery should be enabled or disabled.

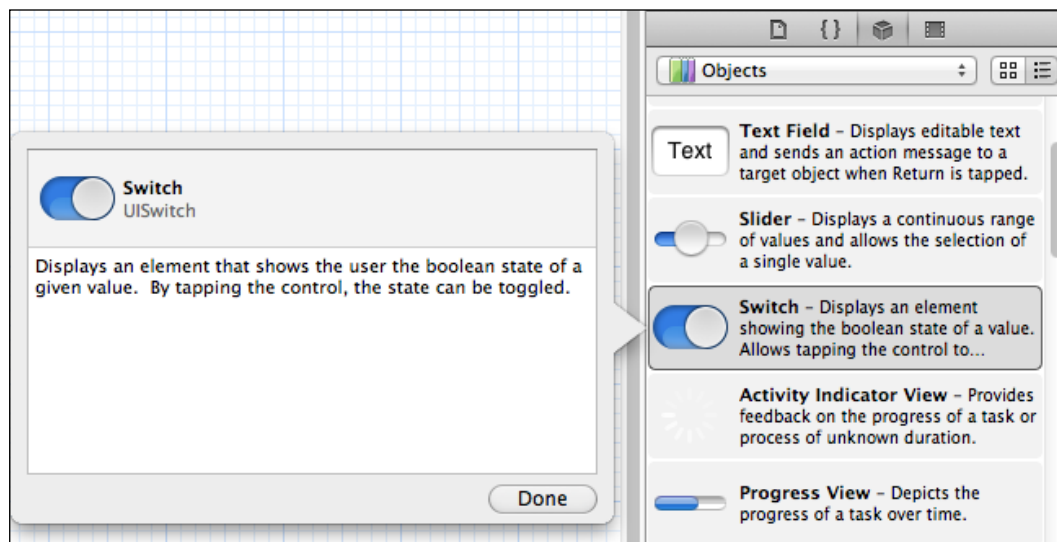
1. Select the `ViewController.xib` file from **Project Navigator**.
2. Set the value of **Background** of the View controller to read **Black Color**.
3. Next, from **Object Library**, select-and-drag a (`UILabel`) **Label** control, and add this to our view.
4. Modify the **Text** property of the control to **Battery Status:**.
5. Modify the **Font** property of the control to **System 42.0**.
6. Modify the **Alignment** property of the control to **Center**.
7. Next, from **Object Library**, select-and-drag another (`UILabel`) **Label** control, and add this to our view directly underneath the **Battery Status** label.
8. Modify the **Text** property of the control to **Battery Level:**.
9. Modify the **Font** property of the control to **System 74.0**.
10. Modify the **Alignment** property of the control to **Center**.

Now that we have added our label controls to our view controller, our next step is to start adding the rest of our controls that will make up our user interface. So let's proceed to the next section.

Adding the Enable Monitoring UISwitch control

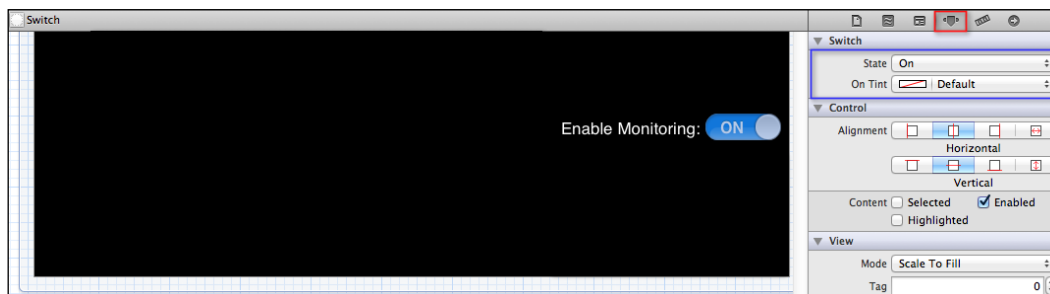
Our next step is to add a switch control to our view controller; this will be responsible for determining whether or not we are to monitor our battery levels and send out alert e-mails whenever our battery life is running low on our iOS device. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (UILabel) **Label** control, and add this to the bottom right-hand corner of our view controller.
2. Modify the **Text** property of the control to **Enable Monitoring**.
3. Modify the **Font** property of the control to **System 17.0**.
4. Modify the **Alignment** property of the control to **Left**.
5. Next, from **Object Library**, select-and-drag a (UISwitch) **Switch** control to the right of the **Enable Monitoring** label.



6. Next, from the **Attributes Inspector** section, change the value of **State** to **On**.

- Then, change the value of **On Tint** to **Default**.




Now that we have added our **Enable Monitoring** switch control to our **BatteryMonitor** View controller, our next step is to add the **Send E-mail Alert** switch that will be responsible for sending out e-mail alerts if it has determined that the battery levels have fallen below our threshold. So, let's proceed with the next section.

Adding the Send E-mail Alert UISwitch control

Now, we need to add another switch control to our view that will be responsible for sending e-mail alerts. This can be achieved by following these simple steps:

- From **Object Library**, select-and-drag another (UILabel) **Label** control, and add this underneath our **Enable Monitoring** label.
- Modify the **Text** property of the control to **Send E-mail Alert**.
- Modify the **Font** property of the control to **System 17.0**.
- Modify the **Alignment** property of the control to **Left**.
- Next, from **Object Library**, select-and-drag a (UISwitch) **Switch** control to the right of the **Send Email Alert** label.
- Next, from the **Attributes Inspector** section, change the value of **State** to **On**.
- Then, change the value of **On Tint** to **Default**.

 To duplicate a UILabel and/or UISwitch control and have them retain the same attributes, you can use the keyboard shortcut *Command + D*. You can then update the **Text** label for the newly added control.

Now that we have added our **Send E-mail Alert** button to our **BatteryMonitor** view controller, our next step is to add the **Fill Gauge Levels** switch that will be responsible for filling our battery gauge when it has been set to **ON**.

Adding the Fill Gauge Levels UISwitch control

Now, we need to add another switch control to our view that will be responsible for determining whether our gauge should be filled to show the amount of battery remaining. This can be achieved by following these simple steps:

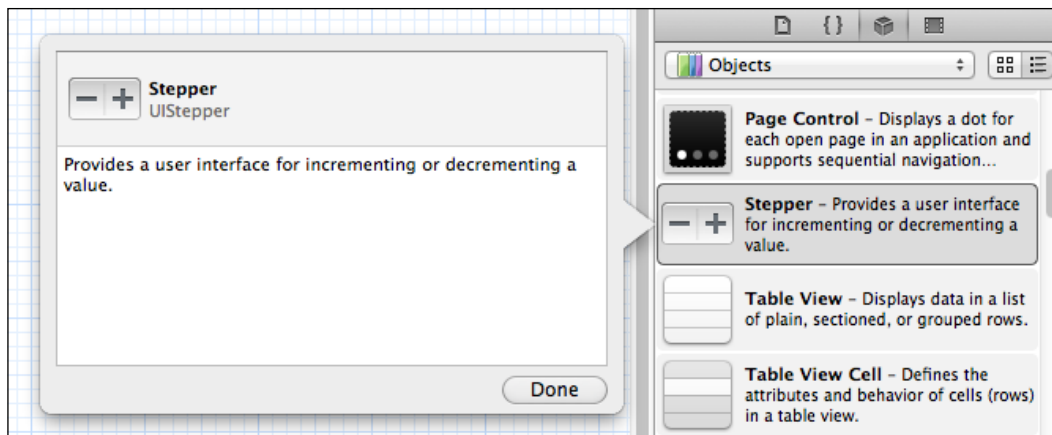
1. From Object Library, select-and-drag another (UILabel) **Label** control, and add this underneath our **Send E-mail Alert** label.
2. Modify the **Text** property of the control to **Fill Gauge Levels**.
3. Modify the **Font** property of the control to **System 17.0**.
4. Modify the **Alignment** property of the control to **Left**.
5. Next, from **Object Library**, select-and-drag a (UISwitch) **Switch** control to the right of the **Fill Gauge Levels** label.
6. Next, from the **Attributes Inspector** section, change the value of **State** to **On**.
7. Then, change the value of **On Tint** to **Default**.

Now that we have added our **Fill Gauge Levels** switch control to our BatteryMonitor view controller, our next step is to add the **Increment Bars** stepper that will be responsible for increasing the number of bar cells within our battery gauge.

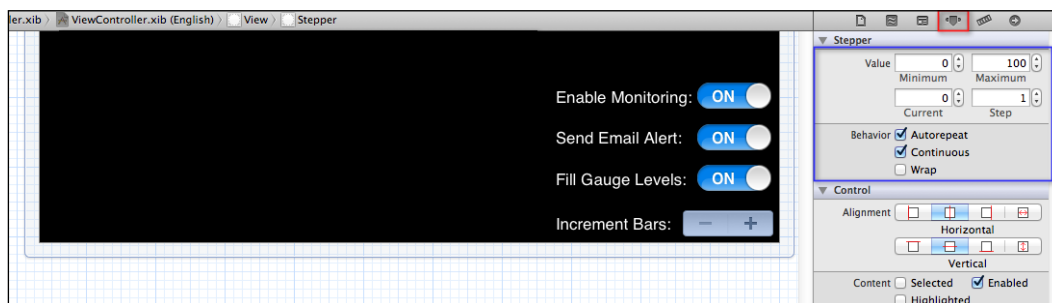
Adding the Increment Bars UIStepper control

Next, we need to add another control to our view that will allow the user to specify the number of battery bars that should be displayed for our battery gauge. This can be achieved by following these simple steps:

1. From Object Library, select-and-drag another (UILabel) **Label** control, and add this underneath our **Fill Gauge Levels** label.
2. Modify the **Text** property of the control to **Increment Bars**.
3. Modify the **Font** property of the control to **System 17.0**.
4. Modify the **Alignment** property of the control to **Left**.
5. Next, from **Object Library**, select-and-drag a (UIStepper) **Stepper** control to the right of the **Increment Bars** label.



6. Next, from the **Attributes Inspector** section, change the **Minimum** value to **0**.
7. Then, change the **Maximum** value to **100**.
8. Next, set the **Current** value to **0** and the **Step** increment to **1**.

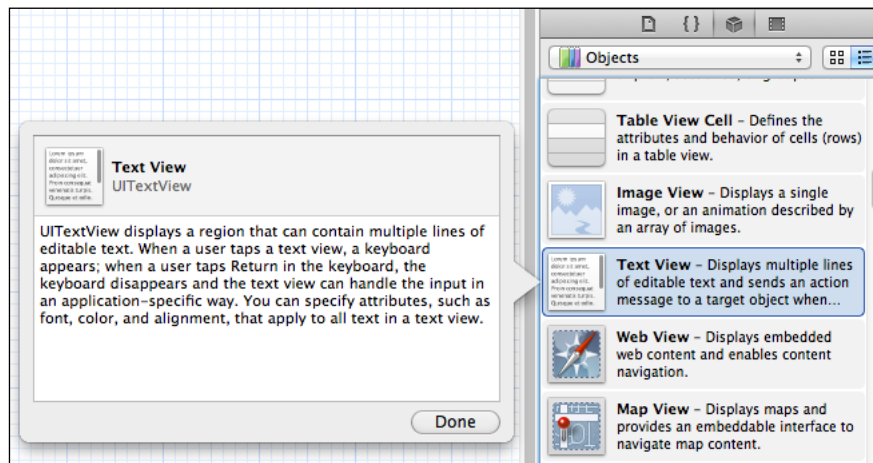


Now that we have added our **Increment Bars** stepper control to our **BatteryMonitor** view controller, our next step is to add an **TextView** control that will be used to display information about the device, such as its model and the version.

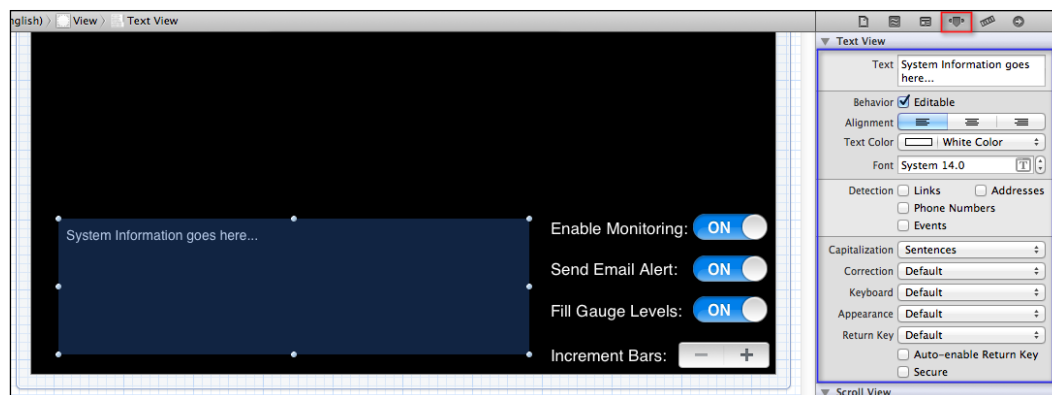
Adding the System Information (UITextView) control

Now, we need to add another control to our view that will be responsible for showing iOS device specific information. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (UITextView) **TextView** control, and add this to the left of the **Enable Monitoring** label.



2. Next, from the **Attributes Inspector** section, modify the **Text** property of the control to **System Information goes here....**
3. Change the **Font** property of the control to **System 14.0**.
4. Modify the **Alignment** property of the control to **Left**.
5. Change the **Capitalization** property to **Sentences**.

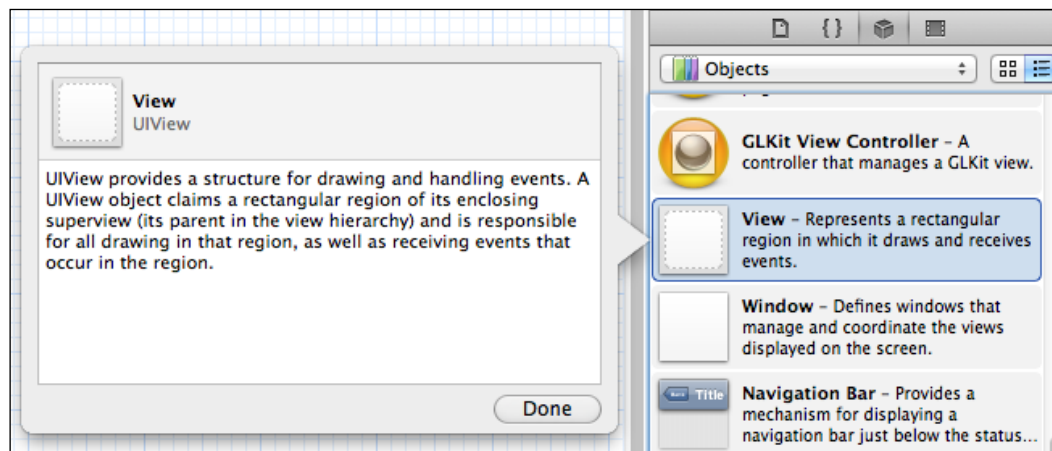


Now that we have added all of our form controls and have built our user interface, our next step is to create our very own custom `UIView` subclass that will be used to display a visual representation of how much battery we have remaining on our iOS device.

1. Select the `BatteryMonitor` folder, choose **File | New | New File...** or press *Command + N*.
2. Select the **Objective-C** class template from the list of available templates.
3. Click on the **Next** button to proceed with the next step within the wizard.
4. Enter in `BatteryGauge` as the name of the file to create.
5. Ensure that you have selected `UIView` as the type of subclass to be created from the **Subclass** dropdown list.
6. Ensure that you have selected the **Targeted for iPad** option.
7. Click on the **Next** button to proceed with the next step of the wizard.
8. Then, click on the **Create** button to save the file to the folder location specified.

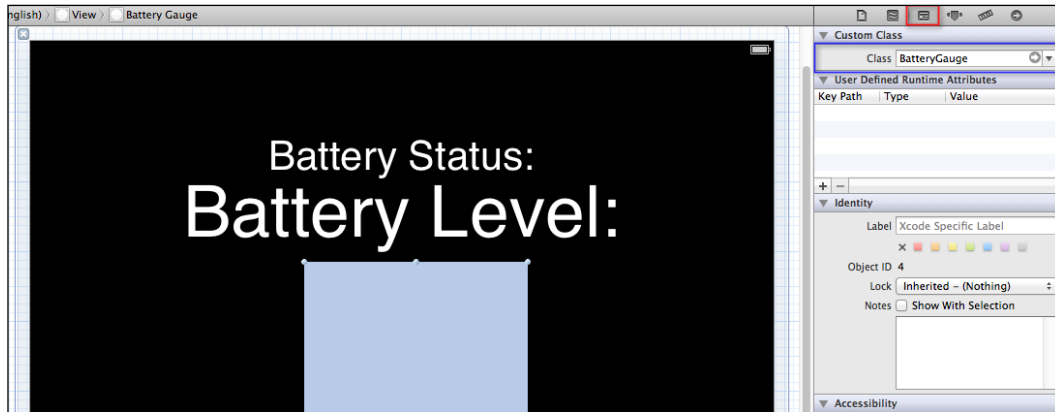
Now that we have created our `BatteryGauge` class, we need to drag a new `UIView` controller and update this to use our newly created `BatteryGauge` class, rather than the default `UIViewController` class.

1. Select the `ViewController.xib` file from the `BatteryMonitor` folder.
2. From **Object Library**, select-and-drag a (`UIView`) **View** control, and add this to the center of our **View** controller.



3. Click-and-select the (`UIView`) controller that we just added to our view.

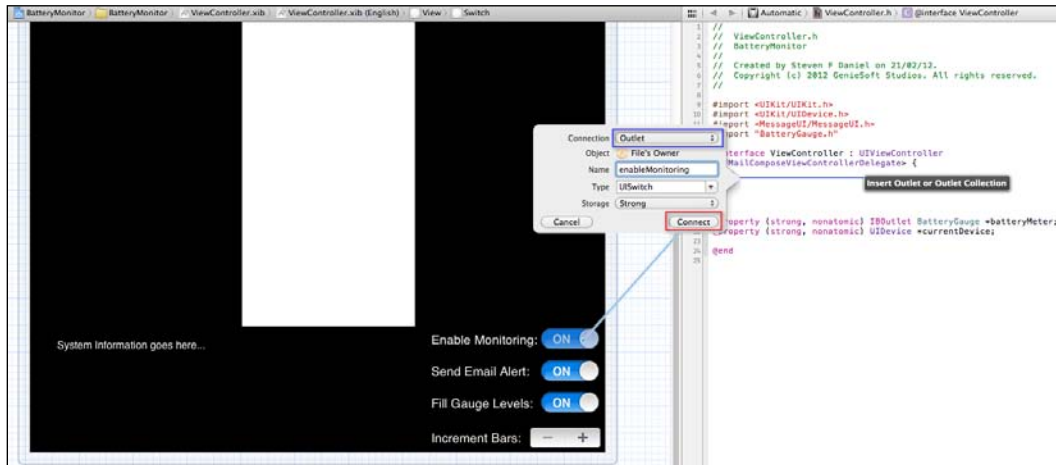
4. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read BatteryGauge.



Our next step is to create the `Outlet` events for each of our controls that have been added to our View controller. Creating these will allow us to access these controls within our code and make modifications to the control properties. To create an `Outlet`, follow these simple steps:

1. Open **Assistant Editor** by choosing **Navigate | Open in Assistant Editor**, or press the *Option + Command + ,* keys.
2. Ensure that the `ViewController.h` interface file is displayed to the left of **ViewController.xib**.
3. Select the **Enable Monitoring** (`UISwitch`) control, then hold down the *Control* key, and drag it into the `ViewController.h` interface file.
4. Choose **Outlet** from the **Connection** dropdown list for the connection to be created.
5. Enter in `enableMonitoring` for the name of the **Outlet** property to be created.

6. Choose **Strong** from the **Storage** dropdown list.

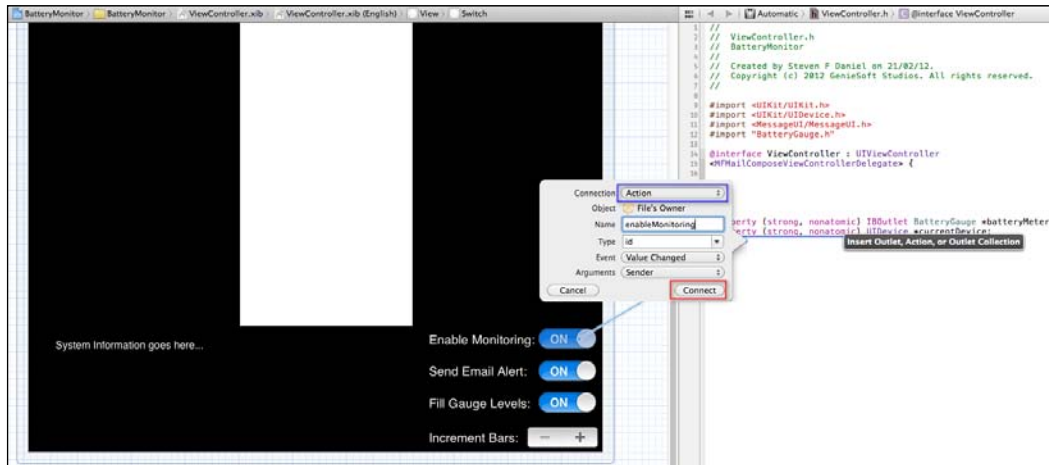


7. Repeat steps 3 to 6 to create IBOutlet's for the **Battery Status, Battery Level, Enable Monitoring, Send E-mail Alert, Fill Gauge Levels, Increment Bars, System Info, and Battery Gauge** controls, while providing the following naming for each, as follows: lblBatteryStatus, lblBatteryLevel, enableMonitoring, sendEmailAlert, fillGauge, totalNoBars, tvSystemInfo, and batteryMeter.

Now that we have created the Outlet events for our controls, we need to create the associated Action events for those Outlets. Creating these actions allows an event to be fired when the button has been pressed. To create an Action, follow these simple steps:

1. With the ViewController.h interface file still displayed to the left of the ViewController.xib View Controller, select the **Enable Monitoring** (UISwitch) control, then hold down the **Control** key, and drag it into the ViewController.h interface file.
2. Choose **Action** from the **Connection** dropdown for the connection to be created.

3. Enter in enableMonitoring for the name of the method to be created.



4. Repeat steps 2 to 3 to create IBActions for the **Enable Monitoring, Send E-mail Alert, Fill Gauge Levels, Increment Bars, and Battery Gauge** controls, while providing the following naming for each, as follows:
enableMonitoring, sendEmailAlert, fillGauge, totalNoBars,
and batteryMeter.

Now that we have successfully connected up each of our controls, and created the required outlets and associated action methods, we can start taking a look at building the functionality for our BatteryMonitor application, so that it has the ability to display the battery levels for our iOS device and e-mail when the threshold reaches 20 percent.

Building the Battery Monitor functionality

Well done! You have made it this far; we have successfully finished building the user interface for both the **Battery Monitor** and **Battery Gauge** screens. Our next step is to start implementing the methods that will be used by each of our controls.

Each of these controls will be responsible for handling the monitoring of our iOS device's battery, with the ability to fill and clear our gauge, increase the number of battery cells present, as well as manually send e-mails when the device's battery level reaches the threshold.

Implementing the View Controller class

We are now ready to start adding additional content to our `ViewController` class. We need to import some interface header files and declare some objects that we will be using throughout our application. We will also need to extend our class in order to provide us with the functionality to compose in-app e-mailing.

1. Open the `ViewController.h` interface file, located within the `BatteryMonitor` folder, and enter in the following highlighted code sections:

```
//ViewController.h
//BatteryMonitor
//Created by Steven F Daniel on 21/02/12.
//Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import <UIKit/UIDevice.h>
#import <MessageUI/MessageUI.h>
#import "BatteryGauge.h"

@interface ViewController : UIViewController
<MFMailComposeViewControllerDelegate> {

    IBOutlet UITextView    *tvSystemInfo;
    IBOutlet UILabel       *lblBatteryLevel;
    IBOutlet UILabel       *lblBatteryStatus;
    IBOutlet UISwitch      *sendEmailAlert;
    IBOutlet UISwitch      *enableMonitoring;
    IBOutlet UISwitch      *fillGauge;
    IBOutlet UIStepper      *totalNoBars;
    IBOutlet BatteryGauge  *batteryMeter;
}

@property (strong, nonatomic) IBOutlet UITextView
    *tvSystemInfo;
@property (strong, nonatomic) IBOutlet UILabel
    *lblBatteryLevel;
@property (strong, nonatomic) IBOutlet UILabel
    *lblBatteryStatus;
@property (strong, nonatomic) IBOutlet UISwitch
    *sendEmailAlert;
@property (strong, nonatomic) IBOutlet UISwitch
    *enableMonitoring;
@property (strong, nonatomic) IBOutlet UISwitch
    *fillGauge;
```

```
@property (strong, nonatomic) IBOutlet UIStepper
    *totalNoBars;

@property (strong, nonatomic) IBOutlet BatteryGauge
    *batteryMeter;
@property (strong, nonatomic) UIDevice *currentDevice;

@end
```

In the preceding code snippet, we import the interface file header information for our `UIDevice.h`, `MessageUI.h`, and `BatteryGauge.h` interface files, so that we can access their class methods. We extended our class, so that we can include the class protocol for `MFMailComposeViewControllerDelegate`, as well as its methods to enable us to compose and send e-mails directly within our application.

2. We then declared each outlet for each of the controls within our view, as well as declared a new outlet instance of our `BatteryGauge` control, so that we can display the total amount of battery life that is left on the iOS device.
3. Next, open the `ViewController.m` implementation file, located within the `BatteryMonitor` folder, and modify the `viewDidLoad` method as shown in the following code snippet:

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

    // Do any additional setup after loading the view,
    // typically from a nib.

    // Enable monitoring of the battery status
    [[UIDevice currentDevice]
     setBatteryMonitoringEnabled:YES];

    // Initialise our Stepper to use the default
    // number of bars.
    [totalNoBars setMinimumValue:1];
    [totalNoBars setMaximumValue:20];
    [totalNoBars setValue:batteryMeter.numBars];

    // Initialize the background color for our Bar
    [batteryMeter setNormalBarColor:[UIColor greenColor]];

    // Get the current status of the iOS device battery.
```

```

[self determineBatteryStatus];

// Request to be notified when battery charge
// or state changes
[[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(determineBatteryStatus)
 name:UIDeviceBatteryLevelDidChangeNotification
 object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(determineBatteryStatus)
 name:UIDeviceBatteryStateDidChangeNotification
 object:nil];

}

```

In the preceding code snippet, we enable the monitoring for our iOS device by setting the `batteryMonitoringEnabled` property to YES, so that our application can be notified of changes when the battery state changes.

4. We then initialize and set the minimum and maximum values for our `UIStepper` control, and initialize the total number of bars to the default values represented by our `BatteryGauge` class. In our next step, we initialize our `BatteryGauge` and fill the color to green, before making a call to the `determineBatteryStatus` function in order to determine the current status of the battery.
5. Finally, we initialize our `UIDeviceBatteryLevelDidChangeNotification` and `UIDeviceBatteryStateDidChangeNotification` methods to make a call to our `determineBatteryStatus` function over-and-over, whenever a change in the battery level or battery state has been detected. Change in the battery state is determined whenever the iOS device has been plugged into a power source, or it has been unplugged.

Implementing the `determineBatteryStatus:` method

Now, that we have set up our `Battery Monitor View` controller and have initialized everything correctly, we are ready to start implementing the method that will be responsible for determining the status of the battery when it has been called by the notification methods.

1. Open the `ViewController.m` implementation file, located within the `BatteryMonitor` folder, and enter in the following code for the `determineBatteryStatus` function:
- ```
// Handle displaying of the battery status.
```

```
- (void)determineBatteryStatus
{
 NSString *OutputString;
 NSArray *batteryStatus = [NSArray arrayWithObjects:
 @"Battery State cannot be determined.",
 @"Battery is in use. Discharging.",
 @"Battery is currently being charged.",
 @"Battery is fully charged.", nil];

 // Determine the current status of the iOS Device Battery.
 switch ([[UIDevice currentDevice] batteryState])
 {
 case UIDeviceBatteryStateUnknown:
 OutputString = [batteryStatus objectAtIndex:0];
 break;
 case UIDeviceBatteryStateUnplugged:
 OutputString = [batteryStatus objectAtIndex:1];
 break;
 case UIDeviceBatteryStateCharging:
 OutputString = [batteryStatus objectAtIndex:2];
 break;
 case UIDeviceBatteryStateFull:
 OutputString = [batteryStatus objectAtIndex:3];
 break;
 default:
 OutputString = [batteryStatus objectAtIndex:0];
 break;
 }

 // Check to determine the state of the battery.
 // If it cannot be determined.
 if ([[UIDevice currentDevice] batteryState] ==
 UIDeviceBatteryStateUnknown)
 {
 batteryMeter.value = -1;
 lblBatteryStatus.text = OutputString;
 lblBatteryLevel.text = [NSString
 stringWithFormat:@"Battery Level: %0.2f%%\n", 0];

 tvSystemInfo.text = @"";
 tvSystemInfo.editable = NO;
 }
 else
 {
 // Get the Battery State and Battery Level and
 // display to the screen.
 }
}
```

---

```

NSString *SystemInfo = [NSString
 stringWithFormat:@"Device Model: %@\nDevice Name:
 %@\nSystem Name: %@\nSystem Version
 %@\nMultitasking
 Supported: %@\n",
 [[UIDevice currentDevice] model],
 [[UIDevice currentDevice] name],
 [[UIDevice currentDevice] systemName],
 [[UIDevice currentDevice] systemVersion],
 currentDevice.multitaskingSupported ? @"YES":
 @"NO"];

lblBatteryStatus.text = OutputString;
lblBatteryLevel.text = [NSString stringWithFormat:
 @"Battery Level: %0.2f%%\n",
 [[UIDevice currentDevice] batteryLevel] * 100];

tvSystemInfo.text = SystemInfo;
tvSystemInfo.editable = NO;

// Show the battery level meter.
batteryMeter.value = [[UIDevice currentDevice]
 batteryLevel];
}

// Determine the level of battery life remaining, and
// Notify the user accordingly.
if ([[UIDevice currentDevice] batteryLevel] * 100) <=
 20)
{
 if (sendEmailAlert.on == YES)
 {
 [self sendEmailAlert];
 }
 else
 {
 UIAlertView *alertMessage = [[UIAlertView alloc]
 initWithTitle:@"Battery Status"
 message:@"Your Battery life is below 20%. Please
 recharge your iOS Device."
 delegate:self
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil];

 [alertMessage show];
 }
}
}
}

```

In the preceding code snippet, we start by setting up an `NSArray` object to store the string value representations of our battery state, and then use the `batteryState` property of the `UIDevice` class that represents the current iOS device to determine what the current status of the battery is. We then perform a check to ensure that we managed to determine the status of our battery. If for some reason we are unable to determine this, we initialize our battery gauge's value to `-1`, so that it doesn't display anything within the bar, and set our battery `Status` and `Level` fields to their defaults.

If we are able to determine the state of the battery, we obtain the device model, the name of the identifying device, the `systemName` property to identify the operating system that is currently running on the iOS device, and `systemVersion` to determine the current version of the operating system that is installed, as well as determine whether the device provides support for multitasking.

2. We then work out current battery level of the iOS device as a percentage, and then update our `batteryMeter` value property to reflect the current battery level on the iOS device. We then perform a check to see if our battery levels reading is less than or equal to 20 percent.
3. In our final steps, we check to see if we have enabled sending of alerts, and if this true, we make a call to the `sendEmailAlert` function to display the e-mail composition sheet, where the information relating to the battery state is pre-populated within the body of the message. Alternatively, if the `sendEmailAlert` option is turned off, we create a new instance of the `UIAlertView` class, and display a warning message to the user.

## Implementing the `enableMonitoring:` method

Now, that we have set up our `BatteryMonitor` View controller and have initialized everything correctly, we are ready to start implementing a method that will be responsible for recording the audio when the user presses the **Start Recording** button.

Open the `ViewController.m` implementation file, located within the `BatteryMonitor` folder, locate the `enableMonitoring` method, and enter in the following code snippet:

```
// Enable monitoring of the iOS battery.
- (IBAction)enableMonitoring:(id)sender
{
 if (enableMonitoring.on == YES) {
 // Turn on monitoring of the iOS battery.
 }
}
```

```
[[UIDevice currentDevice]
 setBatteryMonitoringEnabled:YES];
sendEmailAlert.on = YES;
sendEmailAlert.enabled = YES;
}
else
{
 // Turn off monitoring of the iOS battery.
 [[UIDevice currentDevice]
 setBatteryMonitoringEnabled:NO];

 sendEmailAlert.on = NO;
 sendEmailAlert.enabled = NO;

 // Display an alert message to let the user
 // know that monitoring is disabled.
 UIAlertView *alertMessage = [[UIAlertView alloc]
 initWithTitle: @"Battery Monitoring"
 message: @"Monitoring of the iOS
 Battery has been switched off.
 You will no longer receive alert
 notifications."
 delegate: nil
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil];

 // Show the alert message box.
 [alertMessage show];
}
}
```

In the preceding code snippet, we perform a check on the value of the `enableMonitoring` switch, to determine if we are currently monitoring our iOS device battery. If the value of our `enableMonitoring` switch is off, we disable battery monitoring for our iOS device and disable our `sendEmailAlert` switch to prevent e-mails from being sent. We also create an instance of the `UIAlertView` dialog box to notify the user that battery monitoring has been turned off and e-mail alerts will no longer be sent. Alternatively, if `enableMonitoring` is turned back on, we re-enable battery monitoring on the device and enable the `sendEmailAlert` buttons.



## Implementing the sendEmailAlert: method

Next, we need to implement a method that will be responsible for displaying the in-app e-mail composition sheet when the battery level of our iOS device falls below our 20 percent mark.

1. Open the `ViewController.m` implementation file, located within the `BatteryMonitor` folder, locate the `sendEmailAlert` method, and enter in the following code snippet:

```
- (IBAction)sendEmailAlert:(id)sender {

 // Perform a check to see if we are set up for
 // sending email alerts
 if (sendEmailAlert.on == YES)
 {
 MFMailComposeViewController *mailComposer =
 [[MFMailComposeViewController alloc] init];
 mailComposer.mailComposeDelegate = self;

 // Check to make sure that we are set up to send mail
 if ([MFMailComposeViewController canSendMail]) {
 [mailComposer setToRecipients:[NSArray
 arrayWithObjects:
 @"youremail@yourdomain.com",nil]];
 [mailComposer setSubject:@"Battery Status"];
 [mailComposer setMessageBody:@"Your iOS device is
 running on less than 20% of battery.\nPlease
 recharge your device." isHTML:NO];

 [mailComposer.navigationBar setTintColor:[UIColor
 redColor]];

 mailComposer.modalPresentationStyle =
 UIModalPresentationFormSheet;

 [self presentModalViewController:mailComposer
 animated:YES];
 }
 else
 {
 // Error sending the email message, so
 // notify the user.
 UIAlertView *alertMessage = [[UIAlertView alloc]
 initWithTitle:@"Failure"
 message:@"Your device hasn't been set up for
 email"
```

```

 delegate:nil
 cancelButtonTitle:@"OK"
 otherButtonTitles: nil];

 [alertMessage show];
 }
}
}

```

In the preceding code snippet, we start by determining if we are able to send e-mail messages by checking the status of the `sendEmailAlert` switch control. Next, we create a new object instance of the `MFMailComposeViewController` class, which controls the mail dialog view, thus allowing the user to compose and send an e-mail without leaving the application. We then change the color of the mail composition sheet using the `navigationBar:setTintColor:` method of the controller to red, and then set the subject heading and body of our e-mail message.

2. We then set the controller's `mailComposeDelegate` to self, so that our controller receives the `mailComposeController:didFinishWithResult:error:` message from the `MFMailComposeViewControllerDelegate` protocol when the user finishes with the e-mail dialog box.
3. Finally, we call the controller's `presentModalViewController:animated:` method to display the e-mail dialog box.

```

//=====
// Dismiss our Mail view controller when the user finishes
//=====
- (void) mailComposeController:(MFMailComposeViewController
*)controller didFinishWithResult:(MFMailComposeResult)result
error:(NSError *)error
{
 NSString *emailMessage = nil;

 // Notifies users about errors associated with
 // the interface
 switch (result)
 {
 case MFMailComposeResultCancelled:
 emailMessage = @"Email sending has been cancelled";
 break;
 case MFMailComposeResultSaved:
 emailMessage = @"Email draft saved successfully";
 break;
 case MFMailComposeResultSent:

```

```
 emailMessage = @"Email sent successfully.";
 break;
 case MFMailComposeResultFailed:
 emailMessage = @"Email sending failure.";
 break;
 default:
 emailMessage = @"Problem sending the email.";
 break;
 }
 //Display the alert dialog based on the message derived
 // from the preceding case statement.
 UIAlertView *alert = [[UIAlertView alloc]
 initWithTitle: @"Battery Monitor Email"
 message: emailMessage
 delegate: nil
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil];
 [alert show];

 // make the MFMailComposeViewController disappear
 [self dismissModalViewControllerAnimated:YES];
}
```

In the preceding code snippet, we declare the `mailComposeController:delegate`. The `mailComposeController:didFinishWithResult:error:` method is called when the user finishes with the e-mail dialog box, either by sending an e-mail or by cancelling out of this view. Next, we determine the type of the error that was received by the delegate, and then assign this to an `NSString` object variable `emailMessage`.

4. In our final step, we declare an instance of the `UIAlertView` dialog box to display the error message, before calling the `dismissModalViewControllerAnimated:` method of our view controller.

## Implementing the `fillGauge:` method

Next, we need to start implementing the method that will be responsible for filling our battery gauge control to represent the level of battery life remaining, when the user sets the **Fill Gauge Levels** option to **On**, and **Off** accordingly.

Open the `ViewController.m` implementation file, located within the `BatteryMonitor` folder, locate the `fillGauge` method and enter in the following code snippet:

```
// Handles coloring of the battery bar.
- (IBAction)fillGauge:(id)sender
{
```

```

 if (fillGauge.on == YES)
 {
 // Set the bar color to green and update the display.
 [batteryMeter setNormalBarColor:[UIColor greenColor]];
 [batteryMeter setNeedsDisplay];
 totalNoBars.enabled = YES;
 }
 else
 {
 // Set the bar color to gray and update the display.
 [batteryMeter setNormalBarColor:[UIColor blackColor]];
 [batteryMeter setNeedsDisplay];
 totalNoBars.enabled = NO;
 }
}

```

In the preceding code snippet, we perform a check on our `fillGauge` object control to see if we are currently filling our battery gauge. If we are, we set the `normalBarColor` property of our battery gauge to green, and then make a call to the `setNeedsDisplay` method to redraw our `UIView`, so that the changes are reflected.

Alternatively, if we have determined that we have turned off our `fillGauge` control, we set the color of our battery gauge to black, and again call the `setNeedsDisplay` method to update the display, and then disable our **Increment Bars** control.

## Implementing the `totalNoBars:` method

Next, we need to start implementing the method that will be responsible for incrementing the total number of bars to display within our **Battery Gauge** control.

Open the `ViewController.m` implementation file, located within the `BatteryMonitor` folder, locate the `totalNoBars` method, and enter in the following code snippet:

```

// Handle Incrementing and Decrementing our Battery Bars
- (IBAction)totalNoBars:(id)sender {

 NSInteger iTotalsBars = totalNoBars.value;

 // We need to ensure that our bars are within our range
 if (totalNoBars.value >= 7 && totalNoBars.value <= 20) {
 iTotalsBars++;
 }
 else

```

```
{
 // We need to check that we don't fall below our default
 iTotalsBars = (iTotalsBars < 7) ? iTotalsBars = 7 :
 iTotalsBars--;
}
// Update our Number of bars accordingly
[batteryMeter setNumBars:iTotalsBars];
[batteryMeter setNeedsDisplay];
}
```

In the preceding code snippet, we need to check to ensure that the number of bars within our battery meter doesn't exceed the total allowable number. This is achieved by checking the value of our `totalNoBars` object, whilst checking to see if it falls within our allowable range prior to incrementing its value.

When reducing the total number of bars within the battery gauge, we will need to check to ensure that we don't exceed the minimum value of the control. This is achieved by using the ternary operator, which is represented as a question (?) mark. Finally, we assign the total number of bars to our `setNumBars` method of our `BatteryMeter` class, and then redraw the display by calling the `setNeedsDisplay` method.

## Implementing the Battery Gauge class

In our final step, we need to implement the class that will be used to display a graphical visual representation of our battery levels when the device has determined that the battery state has changed.

1. Open the `BatteryGauge.h` interface file, located within the `BatteryMonitor` folder, and enter in the following code snippet:

```
// BatteryGauge.h
// BatteryMonitor
// Created by Steven F Daniel on 21/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>

// Interface file structure for our Battery Gauge
@interface BatteryGauge : UIView
{
 @private
 float m_flValue, // Current value being displayed
 m_flFillValue, // Current Bar fill value last seen
 m_flMaxLimit, // The bars maximum fill limit
}
```

```

 m_flMinLimit; // The bars minimum fill limit

 int m_iNumBars, // Number of bar segments
 m_iOnIdx, // The index of the bar to turn on
 m_iOffIdx, // The index of the bar to turn off
 m_iFillBarIdx; // The index of the bar to fill

 UIColor *m_clrOBorder, // Color of outer border
 *m_clrIBorder, // Color of inner border
 *m_clrBackgrd, // Background color of gauge
 *m_clrNormal; // Normal segment color
}
// Create the Getters and Setters for the variables.
@property (readwrite, nonatomic) float value;
@property (readwrite, nonatomic) float maxLimit;
@property (readwrite, nonatomic) float minLimit;
@property (readwrite, nonatomic) int numBars;
@property (readonly, nonatomic) float fillValue;

@property (readwrite, retain) UIColor *oBorderColor;
@property (readwrite, retain) UIColor *iBorderColor;
@property (readwrite, retain) UIColor *backgrndColor;
@property (readwrite, retain) UIColor *normalBarColor;

// Battery Gauge Class Methods
-(void) setDefaults;
-(void) drawBar:(CGContextRef)a_ctx withRect:(CGRect) a_rect
andColor:(UIColor *)a_clr barLit:(BOOL)a_IsBarlit;

@end

```

In the preceding code snippet, we declare a new instance of our `UIView` subclass, as our `BatteryGauge` will act as a view within our view controller on our main screen. Next, we declare a collection of private class variables that will be used within our implementation file, as well as define the object properties that can be set and accessed from our **Battery Monitor** view controller.

2. Open the `BatteryGauge.m` implementation file, located within the `BatteryMonitor` folder, and enter in the following code snippet:

```

// BatteryGauge.m
// BatteryMonitor
// Created by Steven F Daniel on 21/02/12.

```

```
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "BatteryGauge.h"

@implementation BatteryGauge

// Synthesized getters and setters properties
@synthesize maxLimit = m_flMaxLimit;
@synthesize minLimit = m_flMinLimit;
@synthesize numBars = m_iNumBars;
@synthesize fillValue = m_flFillValue;
@synthesize oBorderColor = m_clrOBorder;
@synthesize iBorderColor = m_clrIBorder;
@synthesize backgrndColor = m_clrBackgrnd;
@synthesize normalBarColor = m_clrNormal;
```

In the preceding code snippet, we use the `@synthesize` class directive to have it automatically generate the setters and getters for each of the properties that we have declared within the interface file.

3. In the following code snippet, we create an `initWithCoder:` method object, to initialize the battery gauge class and then check to ensure that it has been initialized correctly, prior to making a call to our `setDefaults` method to initialize **Battery Gauge**.

```
// Initializes the instance when brought from nib, etc.
-(id) initWithCoder:(NSCoder *)aDecoder
{
 self = [super initWithCoder:aDecoder];
 if (self) {
 // Assign default values
 [self setDefaults];
 }
 return self;
}
```

4. In the following code snippet, we create a `Method value:` object that we can use to pass in, and set and retrieve the value of our battery gauge.

```
// Method: value accessor
-(float) value
{
 return m_flValue;
}
```

5. In the following code snippet, we create a Method `setValue:` object that determines when the bars within the battery gauge are to be shown as active battery cells or inactive battery cells. A check is performed to determine which bar should be lit within the gauge. We then check our `fRedrawBar` variable to see if the display needs to be redrawn, by calling our `setNeedsDisplay` method to redraw our `UIView`.

```
// Method: value setter
-(void) setValue:(float)a_value
{
 BOOL fRedrawBar = false;

 // take a copy of our current bar value
 m_flValue = a_value;

 // Point at which bars need to light up
 int iOnIdx = (m_flValue >= m_flMinLimit) ? 0 :
 m_iNumBars;
 if (iOnIdx != m_iOnIdx)
 {
 m_iOnIdx = iOnIdx;
 fRedrawBar = true;
 }
 // Point at which bars no longer require to be lit
 int iOffIdx = ((m_flValue - m_flMinLimit) /
 (m_flMaxLimit - m_flMinLimit)) * m_iNumBars;

 if (iOffIdx != m_iOffIdx)
 {
 m_iOffIdx = iOffIdx;
 fRedrawBar = true;
 }
 // Yes, save the fill value of our bars index
 m_iFillBarIdx = MIN(m_iOffIdx, m_iNumBars - 1);
 m_flFillValue = a_value;

 // Determine if we need to redraw the display
 if (fRedrawBar == true) {
 [self setNeedsDisplay];
 }
}
```



6. In the following code snippet, we create a `setNumBars(int)a_iNumBars:` method object, to initialize and set the total number of bars to display to the `UIView` by making a call to the `setValue` method, passing the current battery level reading.

```
// Sets the number of bars for our battery gauge
- (void) setNumBars:(int)a_iNumBars
{
 // save a copy of the value and then update the bars.
 m_iNumBars = a_iNumBars;
 [self setValue:m_flValue];
}
```

7. In the following code snippet, we create a `setDefaults:` function that will be used to initialize the battery gauge when it is first displayed. This function sets the default number of bars to be displayed, as well as the frame and background colors of the gauge.

```
// Configures the default settings for our battery gauge
- (void) setDefaults
{
 // Initialize the Maximum/Minimum limits for our gauge
 m_flMaxLimit = 1.0f;
 m_flMinLimit = 0.0f;
 m_flValue = 0.0f;

 // Set the defaults for our gauge
 m_iNumBars = 20;
 m_iOffIdx = 0;
 m_iOnIdx = 0;

 // Set our gauge default colors
 m_clrBackgrnd = [UIColor blackColor];
 m_clrOBorder = [UIColor grayColor];
 m_clrIBorder = [UIColor blackColor];
 m_clrNormal = [UIColor greenColor];

 m_iFillBarIdx = -1;
 m_flFillValue = -INFINITY;
}
```

8. In the following code snippet, we use the `drawRect:` method to draw each of our battery cells, and then determine which ones need to be filled in and which ones should not be filled in. We first need to work out the boundaries of our battery gauge control, and then calculate the height and width of each of the bars that need to be placed between each of the bars, using the controller's bounds property.

```
// Function to draw the Battery Gauge
-(void) drawRect:(CGRect)rect
{
 CGContextRef ctx; // Graphics context
 CGRect rectBounds, // Bounds for the rectangle
 rectBar; // Rectangle for the bar
 size_t iBarSize; // Size of each bar to be lit

 // Determine the boundaries of our bar
 rectBounds = self.bounds;

 // Adjust the height of our bar
 iBarSize = rectBounds.size.height / m_iNumBars;
 rectBounds.size.height = iBarSize * m_iNumBars;

 // Compute the width and height sizes of our bar
 rectBar.size.width = rectBounds.size.width - 2;
 rectBar.size.height = iBarSize;

 // Obtain the current graphics context of our view.
 ctx = UIGraphicsGetCurrentContext();
 CGContextClearRect(ctx, self.bounds);

 // Fill in the background for each of our bars.
 CGContextSetFillColorWithColor(
 ctx, m_clrBackgrnd.CGColor);
 CGContextFillRect(ctx, rectBounds);

 // Initialize each of our bars,
 //including their line with
 CGContextSetStrokeColorWithColor(ctx, m_clrIBorder
 .CGColor);
 CGContextSetLineWidth(ctx, 1.0);

 // Cycle through the total number of bars and draw each
 // one and lighting up those that need to be lit.
 for (int iX = 0; iX < m_iNumBars; ++iX)
 {
```

```
// Determine the position of this bar.
rectBar.origin.x = rectBounds.origin.x + 1;
rectBar.origin.y = CGRectGetMaxY(rectBounds) -
 (iX + 1) * iBarSize;

// Draw top and bottom borders for each of the bars
CGContextAddRect(ctx, rectBar);
CGContextStrokePath(ctx);

// Determine the fill color for each of our bars
UIColor *clrFill = m_clrNormal;

// Determine if the bar should be filled
BOOL fIsBarLit = ((iX >= m_iOnIdx && iX < m_iOffIdx)
 || iX == m_iFillBarIdx);

// Fill the interior for each of our bars
CGContextSaveGState(ctx);
CGRect rectFill = CGRectInset(rectBar, 1.0, 1.0);
CGPathRef clipPath = CGPathCreateWithRect(rectFill,
 NULL);
CGContextAddPath(ctx, clipPath);
CGContextClip(ctx);

// Call our function to draw and fill each of our
//bars, checking to see if the bar is should be
//filled in.
if (fIsBarLit)
{
 // Draw the bar as a solid fill color
 CGContextSetFillColorWithColor(ctx,
 clrFill.CGColor);
 CGContextFillRect(ctx, rectFill);
}
else
{
 // Draw the bar as background color.
 CGColorRef fillClr =
 CGColorCreateCopyWithAlpha(clrFill.CGColor,
 0.2f);
 CGContextSetFillColorWithColor(ctx,
 m_clrBackgrnd.CGColor);

 CGContextFillRect(ctx, rectFill);
 CGContextSetFillColorWithColor(ctx, fillClr);
 CGContextFillRect(ctx, rectFill);
 CGColorRelease(fillClr);
}
```

```

 }

 CGContextRestoreGState(ctx);
 CGPathRelease(clipPath);
}

// Finally, draw a nice border around our gauge control
CGContextSetStrokeColorWithColor(ctx,m_clrOBorder
 .CGColor);
CGContextSetLineWidth(ctx, 2.0);
CGContextAddRect(ctx, CGRectInset(rectBounds, 1, 1));
CGContextStrokePath(ctx);
}

```

9. Our next step is to obtain the current graphics context for our view using the `UIGraphicsGetCurrentContext` function, and then set the background color for each of our bars with the specified background color. Next, we perform a loop to create each bar determined by the `m_iNumBars` variable. We then determine the `x` and `y` positions for the placement of each of the bars, use `CGContextAddRect` to draw the top and bottom borders of each bar, and use the `CGContextStrokePath` function to draw our bar to the graphics window.
10. We then move on to determine if the bar should be lit and at what position within the gauge this should happen, and then call the `CGContextSaveGState` method to push a copy of the current graphics state onto the graphics stack for the current context to fill the bar. We, then we set the style of the bar, and add this to the current graphics context.
11. Finally, we draw the bar to our view, determining whether it should be filled in or not, by checking the `fIsBarLit` variable.

If the bar is to be lit, we use the `CGContextSetFillColorWithColor` method to fill the bar, and add this to our current graphics context. If the bar does not need to be filled in, we use the `CGContextCreateCopyWithAlpha` method to create a copy of an existing Quartz color, then substitute a new alpha value and use this to fill the cell within the gauge. Since we have used the `CGContextCreateCopyWithAlpha` method, we will need to release this once we have finished with it.

```

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation {

 // Return YES for supported orientations
 return(interfaceOrientation==
 UIInterfaceOrientationPortrait);

}

```

In the preceding code snippet, we force the device to always display in a portrait mode when the device has been rotated. We do this by checking the `interfaceOrientation` variable of the iOS device, so that it will only allow support for the portrait mode, by setting this to the value of the `UIInterfaceOrientationPortrait` type.

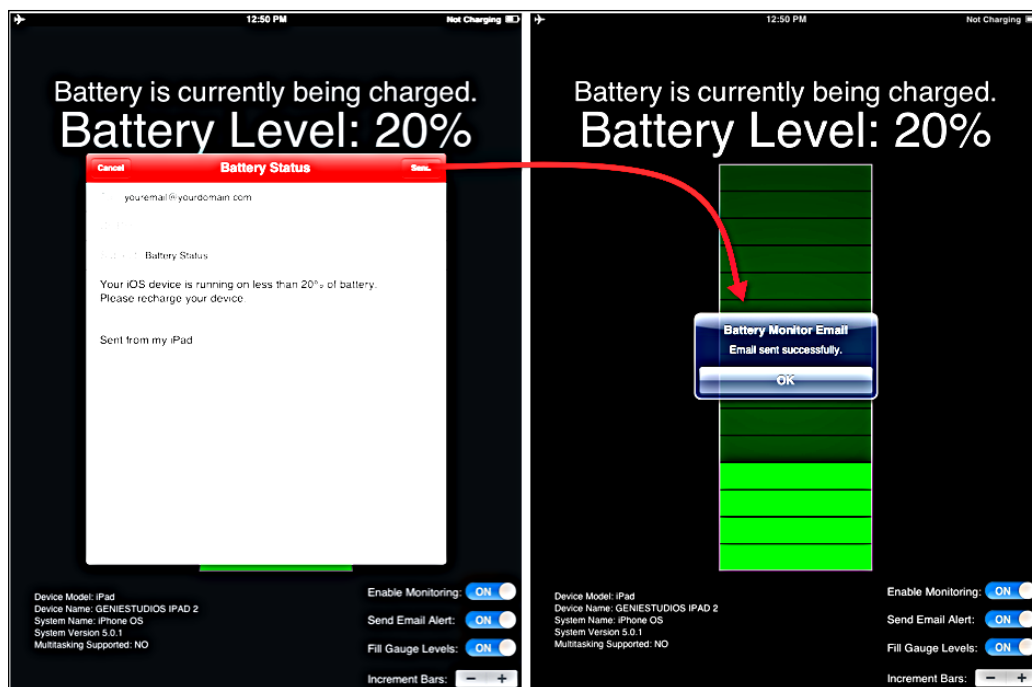


For more information relating to the `UIViewController`, `UIView`, `CGColor`, and `CGContext` classes, you can refer to the Apple Developer Connection Documentation located at the following URLs:

- [https://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UIViewController\\_Class/Reference/Reference.html](https://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UIViewController_Class/Reference/Reference.html)
- [https://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIView\\_Class/UIView/UIView.html#//apple\\_ref/doc/uid/TP40006816](https://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIView_Class/UIView/UIView.html#//apple_ref/doc/uid/TP40006816)
- <https://developer.apple.com/library/mac/#documentation/graphicsimaging/reference/CGColor/Reference/reference.html>
- <https://developer.apple.com/library/mac/#documentation/GraphicsImaging/Reference/CGContext/Reference/reference.html>

## Finishing up

Congratulations, we have finally implemented the methods for our `BatteryMonitor` application. Next, we are ready to build and run our application by choosing **Product | Run** from the **Product** menu, or alternatively pressing *Command + R*. The following screenshot shows the `BatteryMonitor` application running on an iOS Device, displaying the e-mail composition sheet when the **Compose** button has been pressed on the toolbar. As you can see, it contains the pre-populated **Subject** and **Body** message.



From this screenshot, you can amend the subject header and include additional content within the body of the message. Once you have finished composing your e-mail, click on the **Send** button to have your e-mail sent. You will then be presented with a dialog letting you know that your email has been sent successfully.

## Summary

In this chapter, we learned how to use the `UIDevice` class to derive device-related information in order to monitor battery levels for our iOS device, using the `batteryMonitoringEnabled` as well as the `batteryState` methods.

We then looked at how to create a custom `UIViewController` subclass, to display a graphical representation of our battery levels, using the `batteryState` method that comes as a part of the `UIDevice` class. To end the chapter, we looked at how we can use the `MFMailComposeViewController` class to allow the user to send e-mails directly within the app, whenever the battery levels fall below the 20 percent threshold.

In the next chapter, we will look at how we can use the `MapKit` and `Core Location` frameworks to create a `RouteTracking` application that we can use to monitor our location and direction visually, and display the route on the map.



# 6

## RouteTracker Application

The `RouteTracker` application allows you to track the location and direction that you are heading in, visually drawing the route on the map. The application works out the current location of the iOS device, and represents the user's location on the map contained by a blue dot, which shifts as the user's location changes within the map. The route that the user is taking is then visually drawn on the map using a blue line with transparency, so that the underlying road names can be still seen.

In this chapter, we will be taking a closer look at how we can use each of the `Core Location` and `MapKit` frameworks to determine the location that is being travelled by the user, and draw this visually onto our map, using an overlay.

We will look at how to create an instance of `UIViewController` that will be used to create our custom `TrackingOverlay` class. This class will be used to visually draw the route taken by the user on our `Map` view.

In this chapter we will:

- Get an overview of the technologies that we will be using
- Learn how to add the `Core Location` and `MapKit` frameworks
- Walk through the steps to build the `RouteTracker` application
- Implement and use overlays to draw the route taken on the map
- Implement a method to handle the tracking of the user's current location using the `CLLocation` method object
- Implement a method to choose between the different `Map` views
- Use the `NSDate` object to calculate the speed at which the user moves along the route on the map

We have an exciting project ahead of us; so let's get started.



## Overview of the technologies

The `RouteTracker` application makes reference to two very important frameworks, to determine the current location of our device, as well as allow for drawing of graphics to the iOS device's view.

In this chapter, we will be making use of the `MapKit` framework that is based on both the Google Earth and Google Maps data, as well as the APIs that provide developers with a simple mechanism of integrating detailed and interactive mapping capabilities into their applications. The core element of the `MapKit` framework is the `MKMapView` class. This class is a subclass of `UIView` that provides a canvas onto which map and/or satellite information is presented to the user.

The types of information that can be presented within the map are the **Satellite** or **Hybrid** views, whereby the map is displayed on top of the satellite image. The user can manually zoom in and out of the map, by simply pinching or stretching, and shift the location by using the panning gestures. The current location of the device may also be displayed and tracked on the map view. The `MapKit` framework includes support for adding annotations to a map. This takes the form of a pin or custom image, title, and subview that may be used to mark specific locations on a map.

Implementation of the `MKMapViewDelegate` protocol allows an application to receive notifications of events relating to the map view, such as a change in either the location of the user or region of the map displayed, or the failure of the device to identify the user's current location. We will also be making use of the Core Location framework to allow our application to determine and track the current route location of our device, have this information visually represented within our map, then calculate the total distance travelled, and have this displayed to the user once tracking has been turned off.

The information that you can specify on the `Map` view includes the following:

| Map type                        | Description                                                                                                                           |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <code>MKMapTypeStandard</code>  | This is the default map type to be displayed, if none is specified. This will show a normal map containing the street and road names. |
| <code>MKMapTypeSatellite</code> | This type of map will display the satellite view information.                                                                         |
| <code>MKMapTypeHybrid</code>    | This type of map will show a combination of a satellite view with road and street information overlaid onto the map.                  |



For more information on the `MapKit` class, you can refer to the Apple Developer Documentation located at the following URL: [https://developer.apple.com/library/ios/#documentation/MapKit/Reference/MKMapView\\_Class/MKMapView/MKMapView.html](https://developer.apple.com/library/ios/#documentation/MapKit/Reference/MKMapView_Class/MKMapView/MKMapView.html).

## Building the RouteTracker application

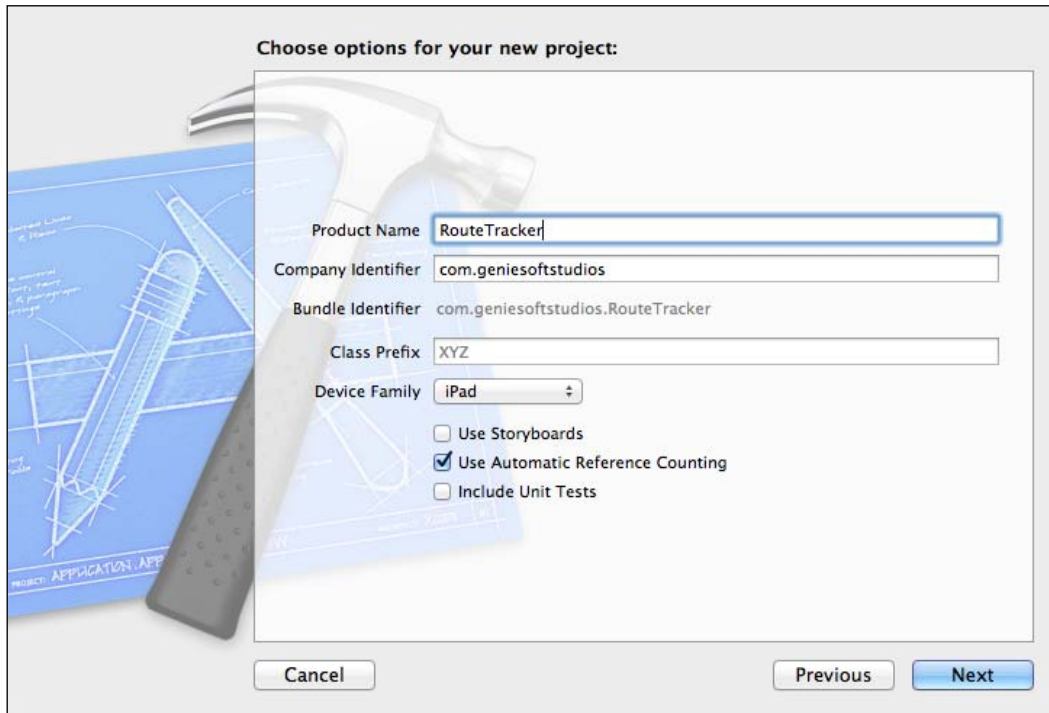
In this section, we will take a look at how to create an application that we can use to run on an iOS device using the `MapKit` and `Core Location` frameworks to allow our application to determine and track the current user's location, and have this information visually represented within our map.

Before proceeding, we first need to create our `RouteTracker` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 3, VoiceRecorder App – Audio Recording and Playback*, under the section named *Building the VoiceRecorder App*.

It is very simple to create our application in Xcode. Just follow the steps listed here.

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the **Single View Application** template from the list of available templates.
4. Click on the **Next** button to proceed with the next step in the wizard.
5. Next, enter in `RouteTracker` as the name for your project.
6. Select **iPad** from under the **Device Family** drop-down list.
7. Ensure that the **Use Storyboard** checkbox has not been selected.
8. Ensure that the **Use Automatic Reference Counting** checkbox is selected.
9. Ensure that the **Include Unit Tests** checkbox has not been selected.

- Click on the **Next** button to proceed with the next step in the wizard.



- Specify the location where you would like to save your project.
- Then, click on the **Save** button to continue and display the Xcode workspace environment.


Now that we have created our `RouteTracker` project, we need to add the Core Location and MapKit frameworks to our project. This will allow us to track the current device location travelled by the user, and have this information visually draw the route taken onto our map.

## Adding the Core Location and MapKit frameworks

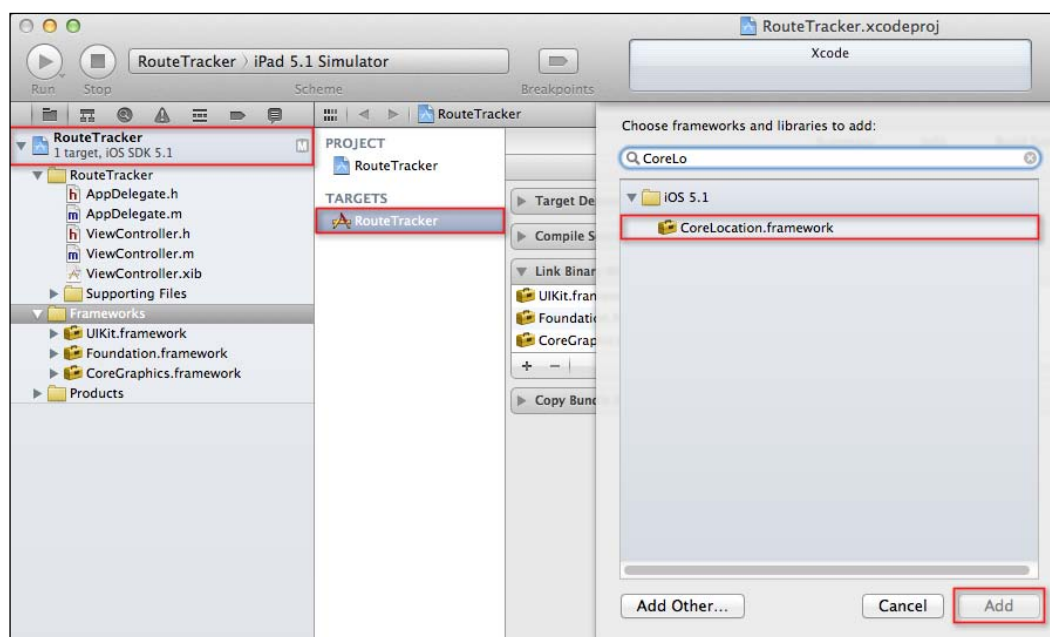
As we mentioned previously, we need to add the Core Location and MapKit frameworks to our project to enable us to track the user location travelled within our map, so that we can use this to draw the route that has been taken to our map.

To add the Core Location framework, select **Project Navigator Group**, and follow these simple steps:

1. Click and select your project from **Project Navigator**.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Binary With Libraries** disclosure triangle.
5. Finally, use **+** to add the library you want.
6. Select `CoreLocation.framework` from the list of available frameworks.

 If you can't find the framework you are looking for, there is also the added ability to search for this directly, right from within the list of available frameworks.

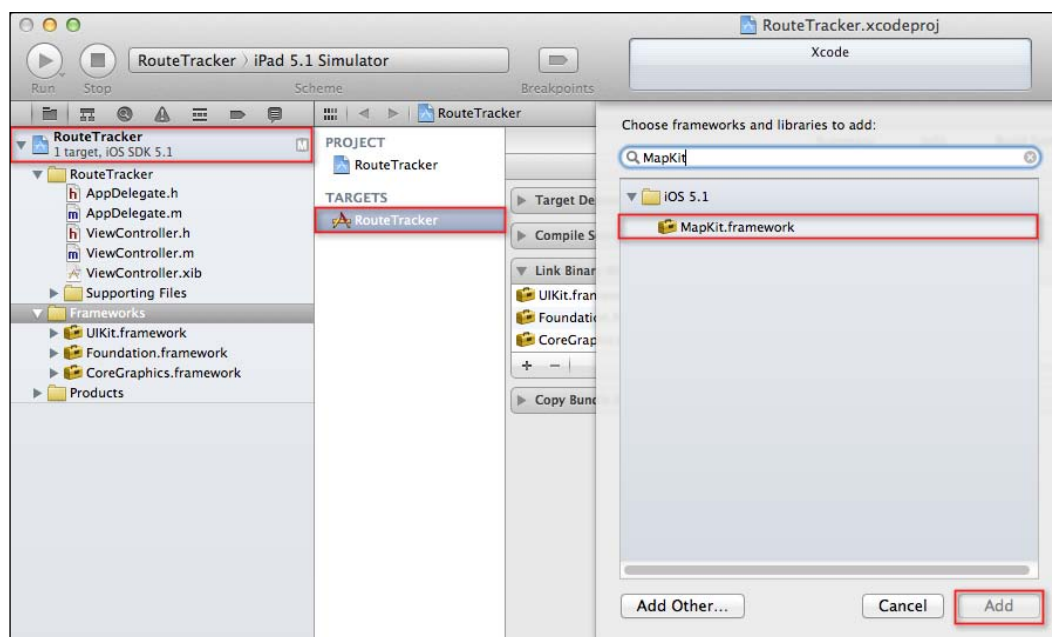
If you are still confused how to go about adding the Core Location framework, you can follow the next screenshot, which highlights the areas that you need to select (surrounded by a rectangle):



Next, we need to add `MapKit.framework` to our project that will allow us to determine our current device location, and the direction that we are currently moving within the map. To add the `MapKit` framework, select **Project Navigator Group**, and follow these simple steps:

1. Click and select your project from **Project Navigator**.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link Binary With Libraries** disclosure triangle.
5. Finally, use **+** to add the library you want.
6. Select `MapKit.framework` from the list of available frameworks.

If you are still confused how to go about adding the `MapKit` framework, you can follow the next screenshot, which highlights the areas you need to select (surrounded by a rectangle):



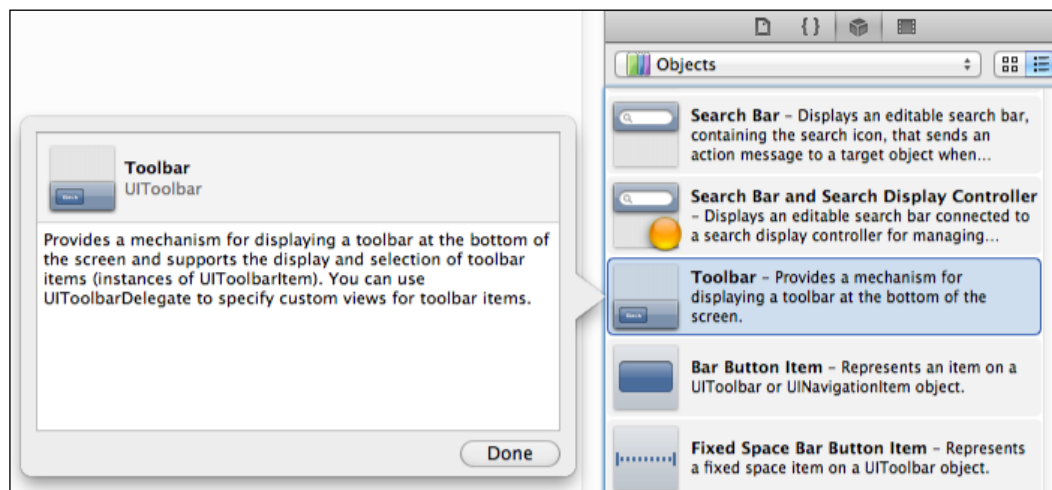
Now that we have added `MapKit.framework` into our project, we need to start building our user interface that will be responsible for allowing us to track our user location and draw this onto our map.

## Creating the main application screen

We have successfully created our project and added the Core Location and MapKit frameworks to allow for tracking of the current device location and have this reflected on our map. Our next step is to build the user interfaces for our RouteTracker application.

This screen will consist of View controller and a toolbar, as well as some controls to handle tracking of our user location and changing between the various map views.

1. Select the `ViewController.xib` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UIToolbar) **Toolbar** controller, and add this to our view.



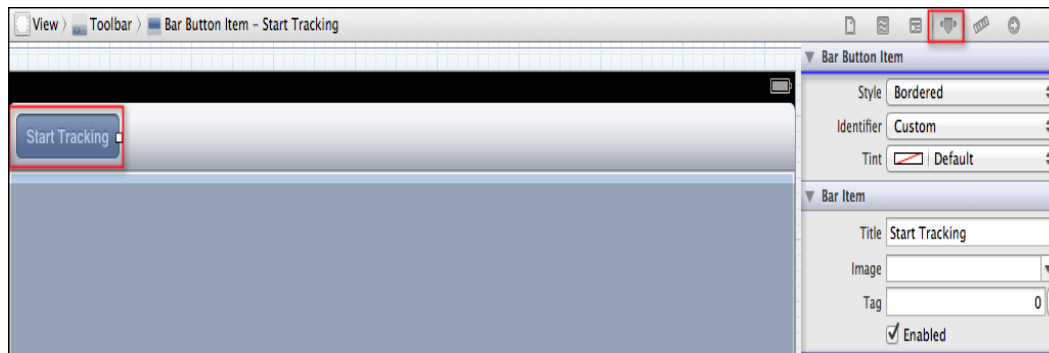
Now we have added our `UIToolbar` toolbar control to our view controller. Our next step is to start adding the **Start Tracking**, **Refresh Map**, and **Change Map Type** buttons. So let's proceed with the next section.

## Adding the Start Tracking button

Our next step is to modify the button within our previously added toolbar; this button will be responsible for handling the tracking of the current user location, and have our map reflect this within the view. This can be achieved by following these simple steps:

1. Select the `ViewController.xib` file from **Project Navigator**.
2. Next, select the **Item** button located within our toolbar that we previously added.

3. From the **Attributes Inspector** section, change the value of **Identifier** to **Custom**.
4. Change the value of **Style** to **Bordered**.
5. Then, change the value of **Title** to **Start Tracking**.



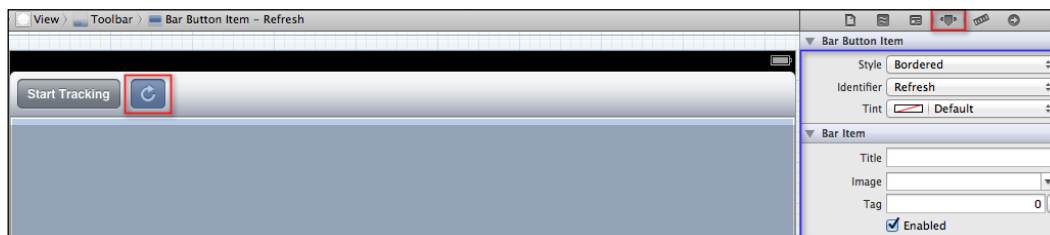
Now that we have added our **Start Tracking** button to our RouteTracker View controller, our next step is to add the **Refresh Map** button that will be responsible for removing our route information from the map, as well as for clearing out the array holding each of these location points.

## Adding the Refresh Map button

Our next step is to add a button to our previously added toolbar; this button will be responsible for removing the visually drawn route information from our map view control. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control inside the toolbar, next to the **Start Tracking** button.
2. From the **Attributes Inspector** section, change the value of **Identifier** to **Custom**.
3. Change the value of **Style** to **Bordered**.

- Then, change the value of **Identifier** to **Refresh**.

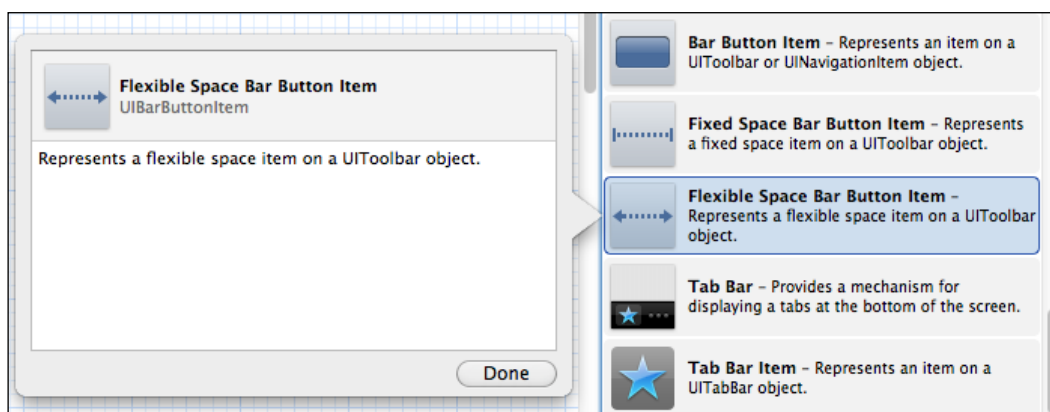


Now that we have added our **Refresh Map** button to our `RouteTracker View` controller, our next step is to add the **Change Map Type** button that will be responsible for changing our map between the various map types that are available.

## Adding the Change Map Type button

Our next step is to add a button to our toolbar; this will be responsible for allowing the user to change between the different map viewing types that are available. The user has the flexibility to choose to have their map displayed as a Satellite, Hybrid or the default Map view. This can be achieved by following these simple steps:

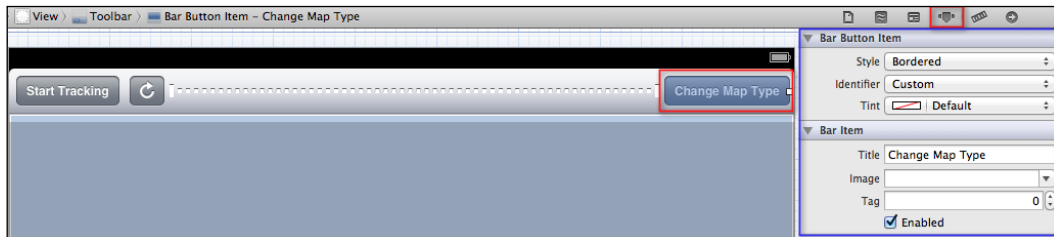
- From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Flexible Space Bar Button Item** control after the **Change Map Type** button within our `UIToolbar`.



- Next, from **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button** control after the **Flexible Space Bar Button Item** control, located within our `UIToolbar`.



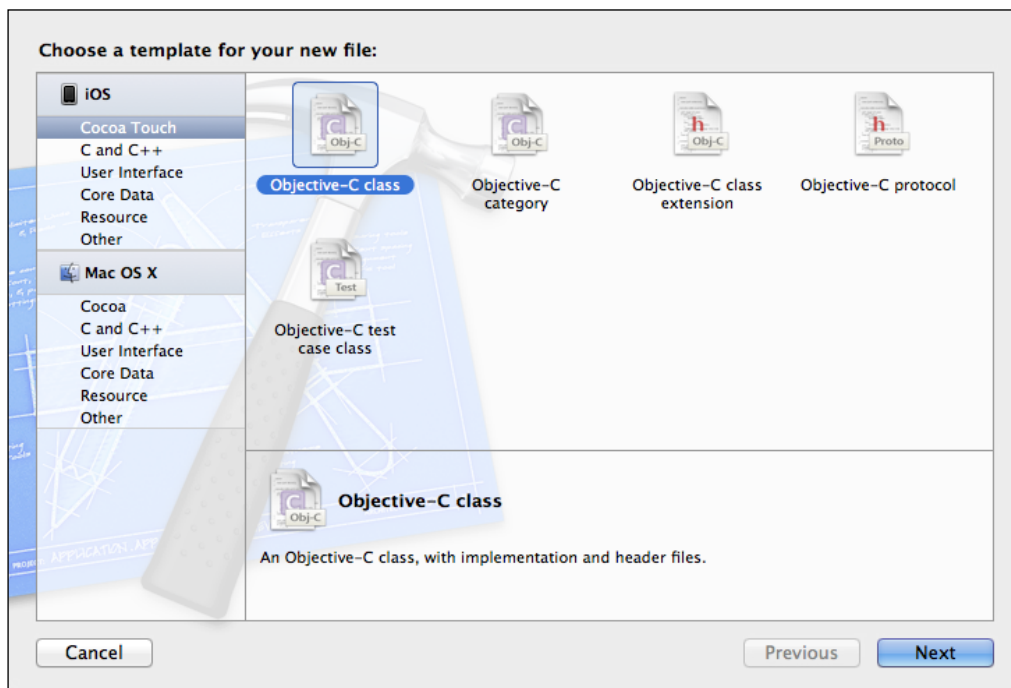
3. From the **Attributes Inspector** section, change the value of **Title** to **Change Map Type**.
4. Change the value of **Style** to **Bordered**.



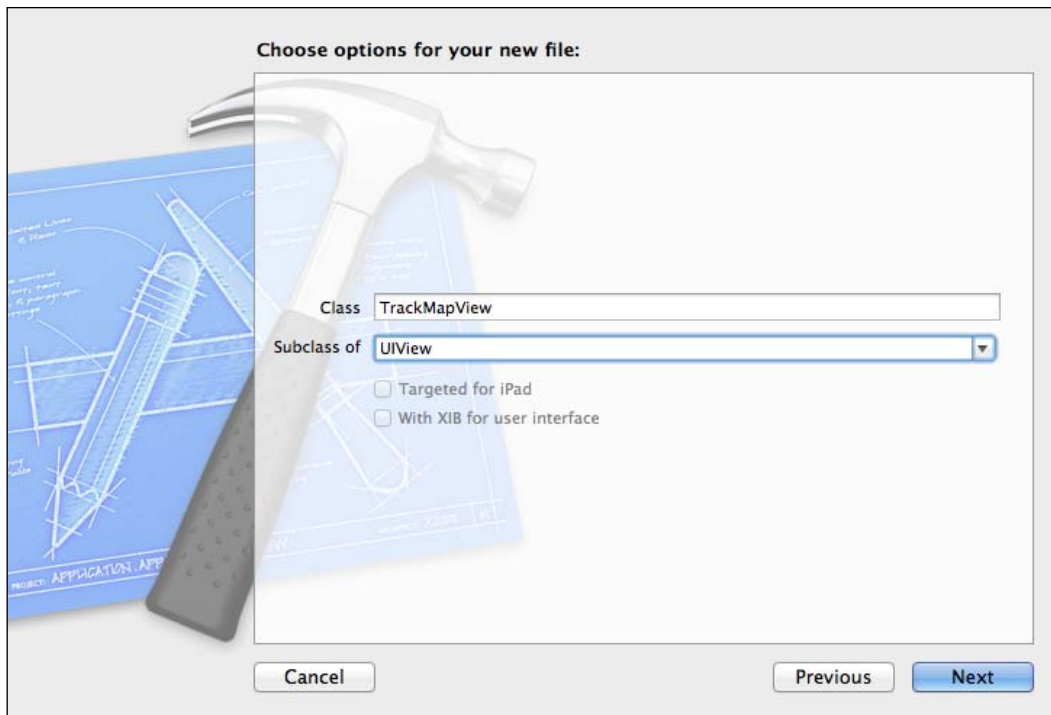
Now that we have added our buttons and have built our user interface, our next step is to create our very own custom `UIView` subclass.

This class will be used to display a visual representation of the route taken by our current location within the map.

1. Select the `RouteTracker` folder, choose **File | New | New File...** or press **Command + N**.
2. Select the **Objective-C** class template from the list of available templates.



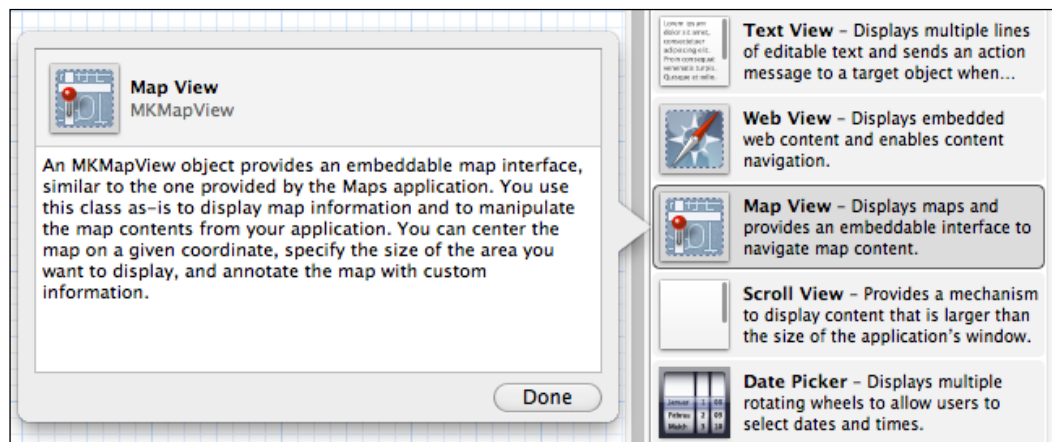
3. Click on the **Next** button to proceed with the next step within the wizard.
4. Enter in `TrackMapView` as the name of the class name to be created.
5. Ensure that you have selected `UIView` as the type of subclass to create from the **Subclass of** drop-down list.



6. Click on the **Next** button to proceed with the next step of the wizard.
7. Then, click on the **Create** button to save the file to the folder location specified.

We have successfully finished creating our `TrackMapView` class; our next step is to add the `MKMapView` controller to our user interface. In this example, we will be creating a new instance of the `MapView` control and overlay our `TrackMapView` class onto it, so that we can draw the route taken by the user and add the necessary waypoints.

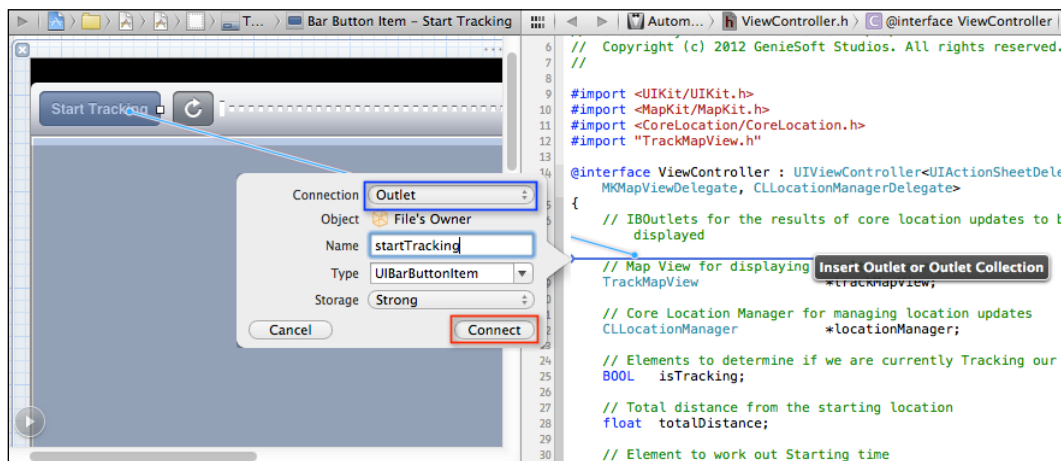
1. Select the **ViewController.xib** file from the `RouteTracker` folder.
2. From **Object Library**, select-and-drag a (`MKMapView`) **Map View** control to the center of our View controller, and adjust the size of **Map View** to fill the entire area of the screen.



Our next step is to create the `Outlet` events for the **Start Tracking**, **Map View**, and **Change Map Type** buttons. Creating these will allow us to access these controls within our code, and make modifications to the control properties. To create an `Outlet`, follow these simple steps:

1. Open **Assistant Editor** by choosing **Navigate | Open in Assistant Editor**, or press *Option + Command + ,*.
2. Ensure that the `ViewController.h` interface file is displayed on **Assistant Editor**.
3. Select the **Start Tracking** (`UIBarButtonItem`) control, then hold down the *Control* key, and drag it into the `ViewController.h` interface file.
4. Choose **Outlet** from the **Connection** dropdown for the connection to be created.
5. Enter in `startTracking` for the name of the **Outlet** property to create.

6. Choose **Strong** from the **Storage** dropdown.

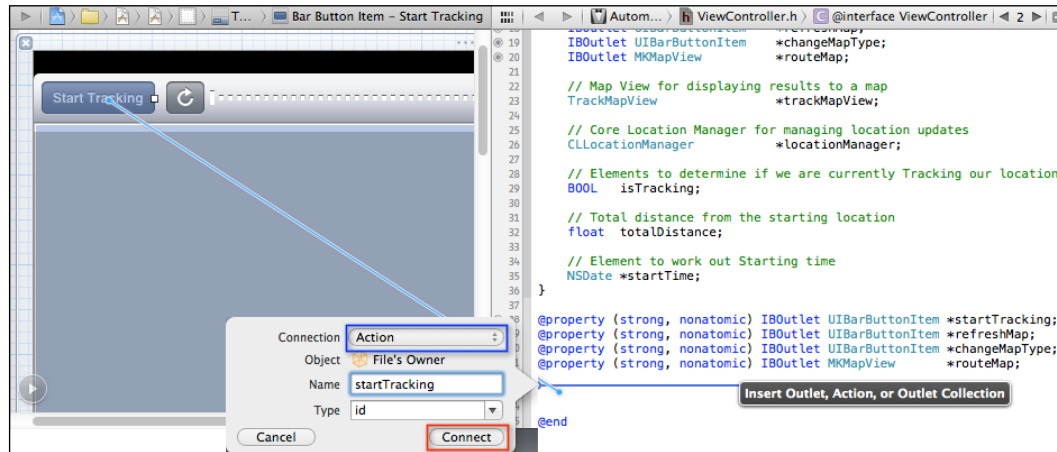


7. Repeat steps 3 to 6 to create the IBOutlets for the **Refresh Map**, **Change Map Type**, **Map View**, and **Track Map View** controls, while providing the following namings for each, as follows: refreshMap, changeMapType, routeMap, and TrackMapView.

Now that we have created the instance variable Outlets for our controls, we need to create the associated Actions for those Outlets events. Creating these actions allows an event to be fired when the button has been pressed. To create an Action, follow these simple steps:

1. With the ViewController.h interface file still displayed in **Assistant Editor**, select the **Start Tracking** (UIBarButtonItem) control, then hold down the **Control** key, and drag it into the ViewController.h interface file.
2. Choose **Action** from the **Connection** dropdown list for the connection to be created.

3. Enter in `startTracking` for the **Name** of the method to be created.



4. Repeat steps 2 to 4 to create the IBActions for the **Refresh Map** and **Change Map Type** controls, while providing the following naming, as follows: `refreshMap` and `changeMapType`.

We have successfully connected up each of our controls, and created the required outlets and associated action methods. Now we can start to take a look at building the functionality for our *RouteTracker* application, so that it has the ability to track and draw our current route, and the ability to change between the different map views.

## Building the RouteTracker functionality

Well done! You have made it this far; we have successfully finished building the user interfaces for both the *RouteTracker* and *TrackMapView* screens. We now need to implement the methods that will be used by our **Start Tracking**, **Refresh Map**, and **Change Map Type** buttons. These will be responsible for determining our current GPS location within the map, and draw the route taken to the map, as well as enable us to change between each of the different types of map views available.

## Implementing the View Controller class

We are now ready to start adding any additional content to our `ViewController` class. We need to import some interface header files and declare some objects that we will be using throughout our application. We will need to extend our class, so that we can use the `ActionSheet`, `MapKit`, and `Core Location` functionality.

1. Open the `ViewController.h` interface file, located within the `RouteTracker` folder, and enter in the following highlighted code sections:

```
// ViewController.h
// RouteTracker
//
// Created by Steven F. Daniel on 12/03/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>
#import "TrackMapView.h"

@interface ViewController : UIViewController<UIActionSheetDelegate,
MKMapViewDelegate, CLLocationManagerDelegate>
{
 // IBOutlets for the results of core location
 // updates to be displayed.
 IBOutlet UIBarButtonItem :startTracking;
 IBOutlet UIBarButtonItem :refreshMap;
 IBOutlet UIBarButtonItem :changeMapType;
 IBOutlet MKMapView :routeMap;

 // Map View for displaying results to a map
 TrackMapView: trackMapView;

 // Core Location Manager for managing location updates
 CLLocationManager: locationManager;

 // Elements to determine if we are currently
 // tracking our location.
 BOOL isTracking;

 // Total distance from the starting location
 float totalDistance;

 // Element to work out Starting time
```

```
 NSDate :startTime;
 }
 @property (strong, nonatomic) IBOutlet UIBarButtonItem
 :startTracking;
 @property (strong, nonatomic) IBOutlet UIBarButtonItem
 :refreshMap;
 @property (strong, nonatomic) IBOutlet UIBarButtonItem
 :changeMapType;
 @property (strong, nonatomic) IBOutlet MKMapView
 :routeMap;

 - (IBAction)startTracking:(id)sender;
 - (IBAction)changeMapType:(id)sender;
 - (IBAction)refreshMap:(id)sender;

@end
```

In the preceding code snippet, we import the interface file header information for our `MapKit.h`, `CoreLocation.h`, and `TrackMapView.h` interface files, so that we can access their class methods. We then need to extend our class, so that we can include each of the following class protocols: `UISheetDelegate`, `MKMapViewDelegate`, and `CLLocationManagerDelegate`, and so that we can access each of their respective methods.

2. Finally, we declared a new outlet to our `trackMapView`, which is used to call the events to draw the route taken, based on the GPS location of our user, and then declare an `NSDate` object that is used to calculate the distance travelled.
3. Next, open the `ViewController.m` implementation file, located within the `RouteTracker` folder, and modify the `viewDidLoad` method as shown in the following code snippet:

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

 // Initialise our isTracking first time round
 isTracking = NO;

 // Do any additional setup after loading the view,
 // typically from a nib.
 trackMapView = [[TrackMapView alloc]
 initWithFrame:routeMap.frame];
```

```

[routeMap setUserTrackingMode:MKUserTrackingModeFollow
 animated:YES];

[routeMap addSubview:trackMapView];
[routeMap setShowsUserLocation:YES];

// Initialize the location manager
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self;
routeMap.delegate = trackMapView;

// Set locationManager to provide the most
// accurate readings possible
locationManager.desiredAccuracy =
 kCLLocationAccuracyBest;
}

```

In the preceding code snippet, we initialize our super class's inherited members and then set `isTracking` to `NO` the first time round. We then initialize our `trackMapView` custom class to take on the same size as our `mapView` control, then add this as a subview of the `mapView` control, and set the `delegate` property of `mapView` to our `trackMapView` class, so that the `mapView` can pass on the notifications to the `trackMapView` class whenever the map moves.

4. In our next step, we initialize the `locationManager` class, and set its `delegate` property to our `trackMapView` class object, then set the `desiredAccuracy` property of the object of `locationManager` to `kCLLocationAccuracyBest`, which specifies that the location and heading information provided by `locationManager` should be as accurate as the iOS device's hardware can provide. This option is also the most energy-demanding option, is quite power- and CPU-intensive, and should only be used when dealing with turn-by-turn navigation.



For more information on the `CoreLocation` class, refer to the Apple Developer Documentation located at the following URL: [https://developer.apple.com/library/ios/#documentation/CoreLocation/Reference/CoreLocation\\_Framework/\\_index.html#//apple\\_ref/doc/uid/TP40007123](https://developer.apple.com/library/ios/#documentation/CoreLocation/Reference/CoreLocation_Framework/_index.html#//apple_ref/doc/uid/TP40007123).



## Implementing the startTracking: method

Now that we have set up RouteTrackerviewController and have initialized everything correctly, we are ready to start implementing the method that will be responsible for tracking our current GPS location when the user presses the **Start Tracking** button.

1. Open the ViewController.m implementation file, located within the RouteTracker folder, locate the startTracking method, and enter in the following code:

```
// called when the user touches the "Start Tracking" button
- (IBAction)startTracking:(id)sender {

 // if the app is currently tracking
 if (isTracking) {
 // Update the button's label and stop tracking
 // and clear out our location points array.
 isTracking = NO;
 startTracking.title = @"Start Tracking";
 [locationManager stopUpdatingLocation];
 [locationManager stopUpdatingHeading];
 }
 else{
 // Start Tracking the user location.
 isTracking = YES;
 startTracking.title = @"Stop Tracking";
 totalDistance = 0.0;

 startTime = [[NSDate date] init];
 [locationManager startUpdatingLocation];
 [locationManager startUpdatingHeading];
 }
 // If we are not tracking display the distance
 //travelled
 if (isTracking != YES) {
 // get the time elapsed since the tracking started
 floatstopTime= -[startTime timeIntervalSinceNow];

 // format the ending message with various
 //calculations
 NSString :message = [NSString stringWithFormat:
 @"Distance: %.02f km\nSpeed: %.02f km/h",
 totalDistance / 1000, totalDistance : 3.6 /
 stopTime];
 }
}
```

```

 // create an alert that shows the message
 UIAlertView :alert = [[UIAlertViewalloc]
 initWithTitle:@"RouteTracker Statistics"
 message:message delegate:self
 cancelButtonTitle:@"OK" otherButtonTitles:nil];

 // Display our alert to the user
 [alert show];
 }
}

```

In the preceding code snippet, we use the `isTracking` object to determine if we are currently tracking. If we have determined that we are currently tracking, we set the `isTracking` variable to `NO`, update the title of the button to **Start Tracking**, and call `stopUpdatingLocation` and `stopUpdatingHeading` to prevent the `locationManager` object from monitoring the iOS device's position.

2. We then make a call to the `timeIntervalSinceNow` method to work out the number of seconds that have elapsed since we started tracking our location, and assign this to our float variable, `stopTime`. In the final steps, we create an `NSString` object to hold both the distance and speed travelled, using the standard metric conversions to calculate distance in kilometers and speed in kilometers per hour, and display this information within a `UIAlertView` dialog box.
3. Alternatively, if the application determined that we were not tracking, we set the `isTracking` variable to `YES`, and update the title of the button to **Stop Tracking**. In our next step, we reset the value of our distance variable to `0.0`, then create a `startTime` variable to monitor how much time is taken for our route, and call `startUpdatingLocation` and `startUpdatingHeading` to begin monitoring the iOS device's position.

## Implementing the refreshMap: method

Next, we need to start implementing the method that will be responsible for providing the user the ability to remove the visual representation of the user's route from our `TrackMapView` class, when the user presses the **Refresh** button.

Open the `ViewController.m` implementation file, located within the `RouteTracker` folder, locate the `refreshMap` method, and enter in the following code snippet:

```

// Resets the map values and clears all location points.
- (IBAction)refreshMap:(id)sender {
 [trackMapView resetWayPoints];
}

```

In the preceding code snippet, we call our `resetWayPoints` method of our `trackMapView` class to remove the visual route taken by our user and remove all location points from our `NSArray` object.

## Implementing the `changeMapType:` method

Next, we need to start implementing the method that will be responsible for providing the user to change between each of the different map views that the MapKit framework provides when the user presses the **Change Map Type** button.

1. Open the `ViewController.m` implementation file, located within the `RouteTracker` folder, locate the `changeMapType` method, and enter in the following code snippet:

```
// Called when the user presses the Change Map Type button
- (IBAction)changeMapType: (id) sender
{
 // Define an instance of our action sheet
 UIActionSheet :actionSheet;

 // Initialize our action sheet with the
 // different mapping types.
 actionSheet = [[UIActionSheet alloc] initWithTitle:
 @"Select a Map View from the list below"
 delegate:self cancelButtonTitle:@"Cancel"
 destructiveButtonTitle:@"Close"
 otherButtonTitles:@"Map View",
 @"Satellite View",
 @"Hybrid View", nil];

 // Set our Action Sheet style and then display
 // it to the user.
 actionSheet.actionSheetStyle =
 UIBarStyleBlackTranslucent;
 [actionSheet showInView:self.view];
}

// Delegate that handles the chosen action sheet options
- (void)actionSheet: (UIActionSheet :)
 actionSheet clickedButtonAtIndex: (NSInteger)buttonIndex
{
 // Determine the chosen item
 switch (buttonIndex) {
 case 1: mapView.mapType = MKMapTypeStandard; break;
 case 2: mapView.mapType = MKMapTypeSatellite; break;
 case 3: mapView.mapType = MKMapTypeHybrid; break;
 }
}
```

```

 default: break; // Catch the Close button and exit.
 }
}

```

In the preceding code snippet, we declare and instantiate an `actionSheet` object that is based on the `UIActionSheet` class, and then initialize our `actionSheet` to display the different map types to choose from.

2. Next, we proceed to set the style for `actionSheet` using the `actionSheetStyle` property of the `UIActionSheet` class, and then display the `actionSheet` into the current view using the `showInView:self.view` method. In our next part, we declare a delegate method to determine the button that was pressed from `actionSheet`, use the `clickedButtonIndex` method of the `actionSheet` property, and check the value of the `buttonIndex` variable to determine the index of the button that was pressed. When using the `buttonIndex` variable, keep in mind that the starting value is always 0.

## Implementing the locationManager: method

Next, we need to start implementing the method that will be responsible for updating the current location each time the `CLLocationManagerDelegate` protocol is called, whenever the `CLLocationManager` updates the current location of the iOS device.

Open the `ViewController.m` implementation file, located within the `RouteTracker` folder, and enter in the following code snippet:

```

// called whenever the location manager updates the
// current location
- (void)locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation
:)newLocation fromLocation:(CLLocation *)oldLocation
{
 // add the new location to the map
 [trackMapView addWayPoint:newLocation];

 // if there was a previous location then add the distance
 // from the old location to the total distance.
 if (oldLocation != nil) {
 totalDistance += [newLocation
 distanceFromLocation:oldLocation];
 }
}

```

In the preceding code snippet, the `didUpdateToLocation` method gets called whenever the `CLLocationManager` class changes the current location of the iOS device. Once this occurs, we pass the new location to our `trackMapView:addWayPoint` method, so that the location can be added to our current list of location points. Next, we perform a check to ensure that we have a previous location point, and then call the `distanceFromLocation` method to work out the distance travelled between `newLocation` and the `oldLocation`, prior to adding the calculated value to our `totalDistance` variable.

## Implementing the `locationManager:didFailWithError:` method

Next, we need to implement the method that will handle whenever an error has occurred with obtaining the current location, or if the user has denied access to the use of location services.

Open the `ViewController.m` implementation file, located within the `RouteTracker` folder, and enter in the following code snippet:

```
// Handle when an error occurs
- (void) locationManager: (CLLocationManager *) manager
 didFailWithError: (NSError *) error
{
 // Stop the location service if the user disallows access
 if ([error code] == kCLErrorDenied) {
 [locationManager stopUpdatingLocation];
 }
}
```

In the preceding code snippet, the `locationManager:didFailWithError` class gets called whenever the use of location services is unavailable or unable to retrieve a location straight away. We use the `error code` property to determine the type of error that occurred, and then call the `stopUpdatingLocation` method of the `locationManager` object.

The following table shows each of the valid error codes and their descriptions, as returned by the `locationManager:didFailWithError` method:

| Core location error code                          | Location manager error description                                                                      |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| <code>kCLErrorLocationUnknown</code>              | This error tells you that the location manager was unable to obtain a location value right now.         |
| <code>kCLErrorDenied</code>                       | This error lets you know that the user denied the access to the location service.                       |
| <code>kCLErrorNetwork</code>                      | The error tells you that the network was unavailable or a network error occurred.                       |
| <code>kCLErrorHeadingFailure</code>               | This error tells you that the heading location travelled could not be determined.                       |
| <code>kCLErrorRegionMonitoringDenied</code>       | This error tells you that the user denied the access to the region monitoring service.                  |
| <code>kCLErrorRegionMonitoringFailure</code>      | This error tells you that a registered region could not be monitored.                                   |
| <code>kCLErrorRegionMonitoringSetupDelayed</code> | This error tells you that Core Location could not initialize the region-monitoring feature immediately. |



For more information on the `didFailWithError:` error code of the Core Location class, refer to the Apple Developer Documentation located at the following URL: [http://developer.apple.com/library/ios/#documentation/CoreLocation/Reference/CoreLocation\\_Framework/\\_index.html#//apple\\_ref/doc/uid/TP40007123](http://developer.apple.com/library/ios/#documentation/CoreLocation/Reference/CoreLocation_Framework/_index.html#//apple_ref/doc/uid/TP40007123).

## Implementing the `shouldAutorotateToInterfaceOrientation:` method

Next, we need to implement a method that will be responsible for preventing our device from having our application being displayed within the various views.

Open the `ViewController.m` implementation file, located within the `RouteTracker` folder, locate the `shouldAutorotateToInterfaceOrientation:` method, and enter in the following code snippet:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(
 UIInterfaceOrientation)interfaceOrientation
{
 return (interfaceOrientation ==
 UIInterfaceOrientationPortrait);
}
```

In the preceding code snippet, we force the device to always display in the portrait mode when the device has been rotated. We do this by checking the `interfaceOrientation` variable of the iOS device, so that it will only support the portrait mode by setting this to the value of the `UIInterfaceOrientationPortrait` type.

## Implementing the `TrackMapView` class

In our final step, we need to implement the class that will be used to display a visual graphical representation of the location points along the route taken by the user using the iOS device's built-in GPS capabilities, with the help of Google Map web Services that come as part of the `MapKit` framework.

1. Open the `TrackMapView.h` interface file, located within the `RouteTracker` folder, and enter in the following code snippet:

```
// TrackMapView.h
// RouteTracker
//
// Created by Steven Daniel on 18/03/12.
// Copyright (c) 2012 GENIESOFT STUDIOS. All rights reserved.

#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>

@interface TrackMapView :UIView<MKMapViewDelegate>
{

```

```

 // An array of way points containing each location
 NSMutableArray :wayPoints;
 }

 // Declare our function to add each new location to our array
 -(void)addWayPoint:(CLLocation :) wayPoint;
 -(void)resetWayPoints;

@end

```

In the preceding code snippet, we declare a new instance of our `UIView` subclass as `TrackMapView`, which will act as a view within our Map view on our main screen. We declare an `NSMutableArray` object's `wayPoints` variable that will hold each of the location points travelled within the map. We declare an `addWayPoint:` method to basically add each new location travelled within the map to the `wayPointsNSMutableArray` object, and draw the route taken on the map.

2. Finally, we declare another `resetWayPoints` method to remove all previously added location items from the `wayPoints` array, thus removing the route taken from the map.
3. Open the `TrackMapView.m` implementation file, located within the `RouteTracker` folder, and enter in the following code snippet:

```

// Initialize the Track Map View class
- (id)initWithFrame:(CGRect)frame
{
 self = [super initWithFrame:frame];
 if (self) {
 // Initialization background and our location points
 self.backgroundColor = [UIColor clearColor];
 wayPoints = [[NSMutableArray alloc] init];
 }
 return self;
}

```

In the preceding code snippet, we modify the `initWithFrame:` method to initialize the `TrackMapView` class that checks to ensure that it has been initialized correctly, prior to initializing our `wayPoints` array using the `NSMutableArray init:` method.



4. In the following code snippet, we perform a call to the `NSMutableArray removeAllObjects` method to remove all array elements from the `wayPoints` array, resulting in the user's visual route being removed from the `TrackMapView` class. Finally, we call the `setNeedsDisplay` method to refresh the view, after the changes have been applied.

```
// Remove all points within our array
- (void)resetWayPoints
{
 [wayPoints removeAllObjects];
 [self setNeedsDisplay];
}
```

5. In the following code snippet, we use the `drawRect:` method to draw the route line of the path travelled by the user. We then check to make sure that we have enough points within our array, and check to ensure that our view is not hidden, as we want to draw our path when the map is visible. Next, we need to obtain the current graphics context using the `UIGraphicsGetCurrentContext` function, and then set the width of the line that needs to be drawn to our view, representing the path.

```
// Called automatically when the view needs to be
// displayed to draw our route taken.
- (void)drawRect:(CGRect)rect
{
 // Get our current graphics context
 CGContextRef context = UIGraphicsGetCurrentContext();
 CGContextSetLineWidth(context, 6.0);
 CGContextPoint point;

 // Exit from our method if our view is hidden or
 // there is only one point to draw
 if (self.hidden || wayPoints.count == 1) return;

 // loop through each of our location points in our
 //Array
 for (intwayPoint=0; wayPoint<wayPoints.count;
 wayPoint++)
 {
 // Set the lines's color to red with transparency
 CGContextSetRGBStrokeColor(context, 0, 0, 1.0, 0.6f);

 // Get the next location from our points array
```

---

```

CLLocation :nextLocation = [wayPoints
 objectAtIndex:wayPoint];
CGPointlastPoint = point;

// get the view point for the given map coordinate
point = [(MKMapView :)self.superview
 convertCoordinate:nextLocation.coordinate
 toPointToView:self];

// Don't process our first starting position
if (wayPoint != 0)
{
 // move to the last point and draw a line
 CGContextMoveToPoint(context, lastPoint.x,
 lastPoint.y);
 CGContextAddLineToPoint(context, point.x, point.y);
}
// Draw the line to the view
CGContextStrokePath(context);
}
}

```

6. Next, we declare a variable point of type `GPoint` that will be used to store the next point in the line, then perform a loop to cycle through each of the location points, and use the `CGContextSetRGBStrokeColor` method to set the line color.
7. Finally, we call the `convertCoordinate:toPointToView` method to receive the point to the next location, then add a line from the last point draw to the current points coordinates point, use `CGContextMoveToPoint` and `CGContextAddLineToPoint` to work out and draw the line, and use the `CGContextStrokePath` function to draw our line to the graphics window.

```

// add a new point to the list
- (void)addWayPoint:(CLLocation :)wayPoint
{
 // store last element of point
 CLLocation :lastPoint = [wayPoints lastObject];

 // if new point is at a different location than
 //lastPoint
 if (wayPoint.coordinate.latitude !=
 lastPoint.coordinate.latitude ||
 wayPoint.coordinate.longitude !=
 lastPoint.coordinate.longitude)
 {

```

```
 // add the point to our map and redraw the view.
 [wayPoints addObject:wayPoint];
 [self setNeedsDisplay];
 }
}
```

In the preceding code snippet, we store the last point drawn on our map, and check to ensure that our current coordinates don't describe the same coordinates as the last point drawn on our map. If either are found to be different, we add the new `CLLocation` point coordinates to our `NSMutableArray` object, and then force the display to refresh the view by calling the `setNeedsDisplay` method of `UIView`.

8. In the following code snippet, we set the `hidden` property of our `trackMapView` view to `YES`, whenever the area that is currently being displayed on our `MKMapView` is about to shift. This is to prevent the line drawn from appearing misplaced on the map, whenever the transition takes place.

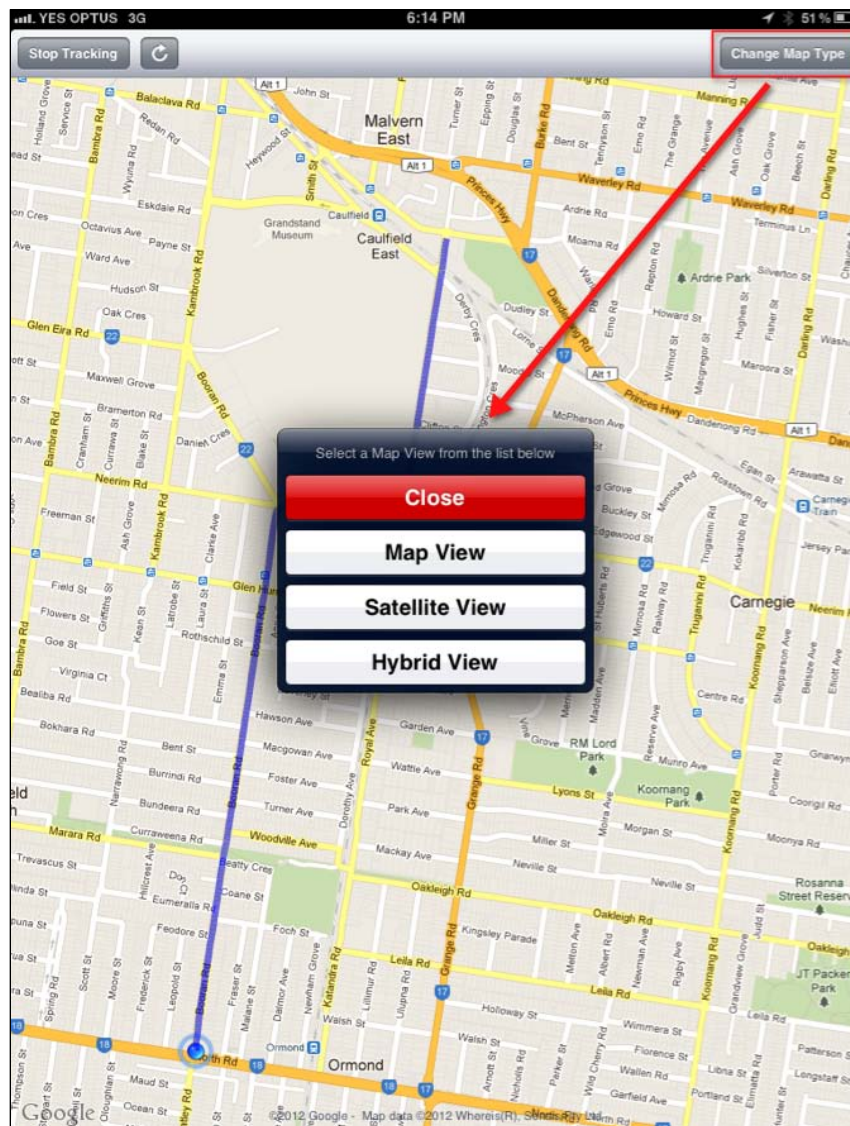
```
#pragma mark mapView delegate functions
-(void)mapView:(MKMapView :)
 mapView regionWillChangeAnimated:(BOOL)animated
{
 // Hide the view when the region is going to change.
 self.hidden = YES;
}
```

9. In the following code snippet, we set the `hidden` property of our `trackMapView` view to `NO` and then refresh our view. This method gets called whenever the `MKMapView` finishes transitioning to its new location.

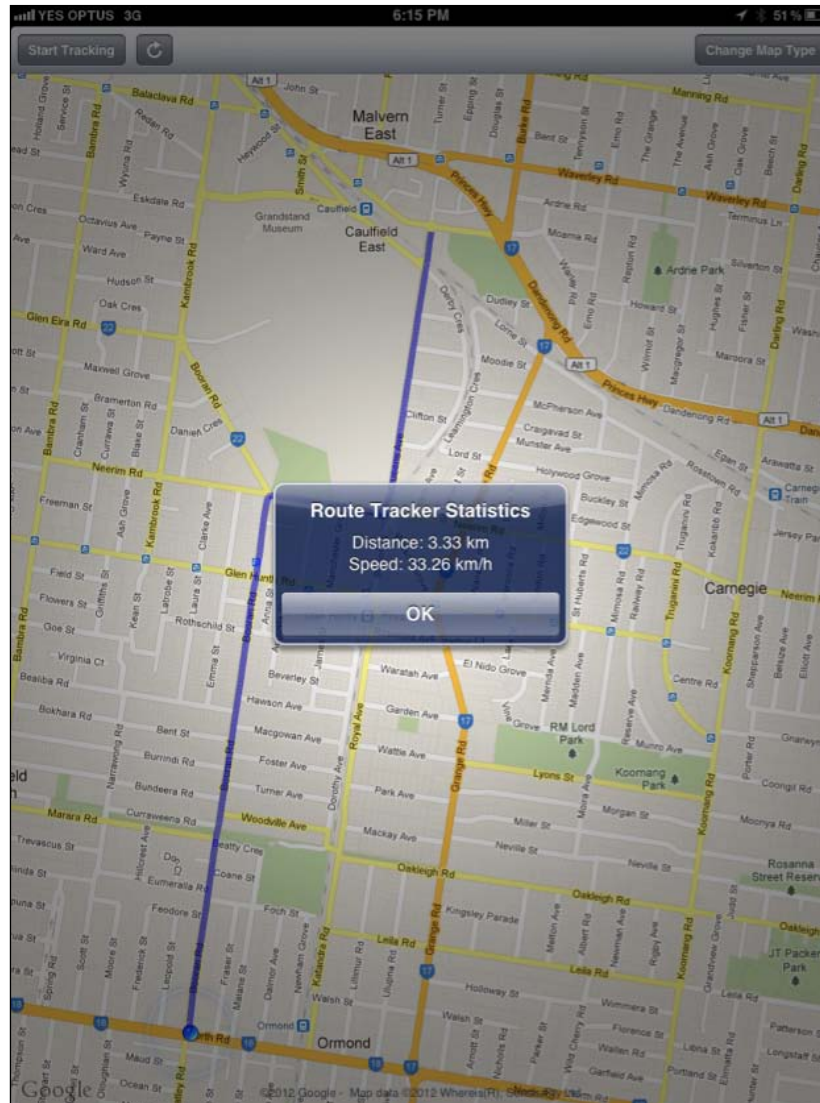
```
// refresh the route display.
-(void)mapView:(MKMapView :)
 mapView regionDidChangeAnimated:(BOOL)animated
{
 self.hidden = NO;
 [self setNeedsDisplay];
}
```

## Finishing up

Congratulations, we have finally implemented the methods for our RouteTracker application. Next, we are ready to build and run our application by choosing **Product | Run** from the **Product** menu, or alternatively by pressing **Command + R**. The following screenshot shows the RouteTracker application running on their OS device, and the information currently being tracked from inside a moving car.



The previous screenshot shows you how you can choose from the various types of map views when **Change Map Type** is pressed. In the following screenshot, we display the total distance travelled and the speed at which we travelled. This happens when the user has pressed the **Stop Tracking** button.



## Summary

In this chapter, we learned how to use and work with both the Core Location and MapKit frameworks to determine the current user location, which relies on the use of Google Maps Web services to obtain the map data. We looked at how to create a new `UIView` class to enable us to draw our route on top of `MKMapView` through the use of overlaying this information as a sub-view, and how we can use the `CLLocationManager` class to allow us to access the current user's coordinates, so that this information can be used to draw and track this information on our map.

We also looked at how we can use action sheets as well as how we can change between the various Map views, Map, Satellite and Hybrid. To end the chapter, we looked at how we can use the `NSDate` class to calculate the distance, speed, and time taken to complete the route.

In the next chapter, we will look at how to create a `VeterinaryClinic` application that will store pet information, including photo images into an SQLite database using Core Data.





# 7

## VeterinaryClinic Application

The `VeterinaryClinic` application allows you to keep a visual track of each pet that visits the veterinary surgery. The application records information specific to each pet and even allows you to upload a photo, either using the camera or from the photo album library.

In this chapter, we will take a look at how we can create a simple application to store information on pets and veterinary information. In our example, we will be making use of the powerful Core Data framework that will allow us to create and edit information relating to cats and dogs using a form, and then having this information stored within an SQLite database.

We will look at how to incorporate the camera and the image picker control to capture images from the iOS device, and have this information stored within an SQLite database relating to each pet.

In this chapter we will:

- Build the `VeterinaryClinic` application using Storyboards
- Build the Core Data Model and create the table schema
- Learn how to navigate between screens using Storyboards
- Implement the method to populate a `UITableView` view from a database
- Implement a method to save a record to the database
- Implement a method to use the photo library or the iOS camera
- Learn how to set the different keyboard styles on the control fields
- Implement the method to delete the table view items
- Implement an ability to perform searches within a `UITableView` view

We have an exciting project ahead of us, so let's get started.



## Overview of the technologies

The `VeterinaryClinic` application makes reference to the Core Data framework. The Core Data framework is described as an abstraction layer that sits on top of an SQLite database and enables developers to easily implement data-centric applications, by modeling your data storage around entities (which are known as classes) that contain the relationships between them. If you are familiar with the Entity-Framework that comes part of the Microsoft .NET framework, then this one is of a similar nature.

We will also be taking a look at how to store image data using the **Binary Data** data-type, which is a part of Core Data that enables us to upload images. We will take a look at how to convert an image photo to `NSData` format, before it can be stored within the SQLite database, using the `dataWithImagePNGRepresentation` method.

For further coverage of this area, please refer to *Chapter 4, Enhanced AddressBook App – Core Data*, where we discuss more about this.

## Building the VeterinaryClinic application

The ability to store information relating to our pets is one of the most common things that veterinary clinics use on a daily basis. These can relate to adding a photo of your pet, as well as their attributes, such as the name, age, date of birth, breed, and type of animal.

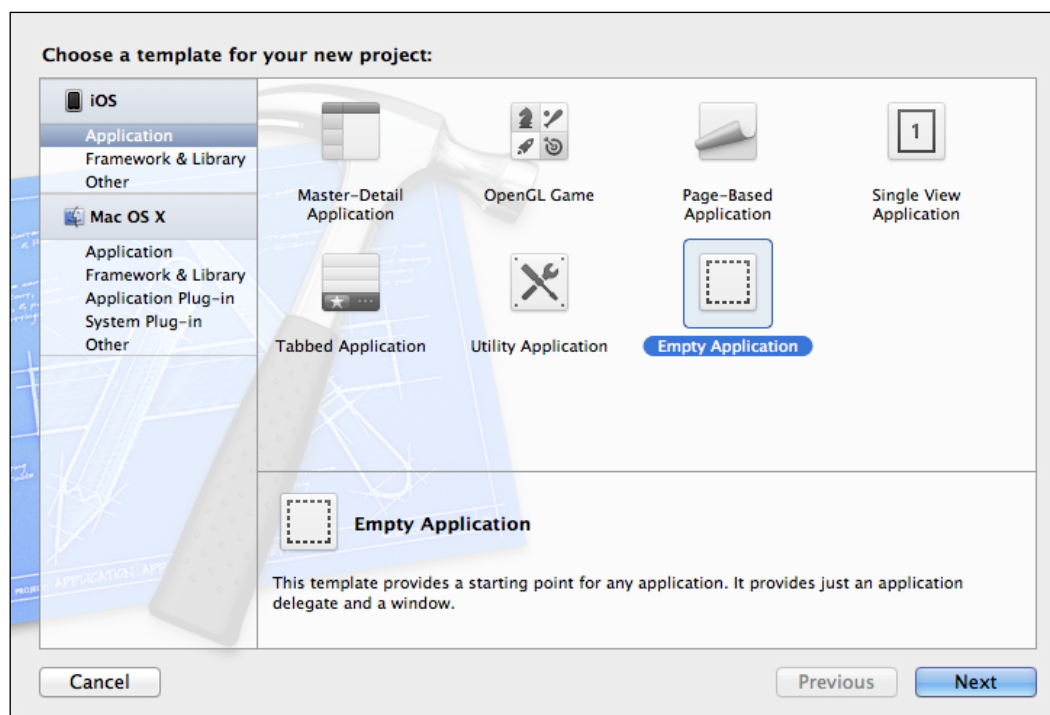
In this section, we will take a look at how to create an application, which will do just that so it can run on an iOS device, enabling us to create and edit new pet information, assign their photo and their basic attributes, as well as their address information and owner contact details.

We will also be storing this information within a SQLite database, and have this information populated within a `UITableView` control with the added functionality of being able to delete items that have been previously added to the list.

Before we can proceed, we first need to create our `VeterinaryClinic` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 2, Task Priorities – Building a TaskPriorities iOS App*, under the section named *Building the TaskPriorities app*.

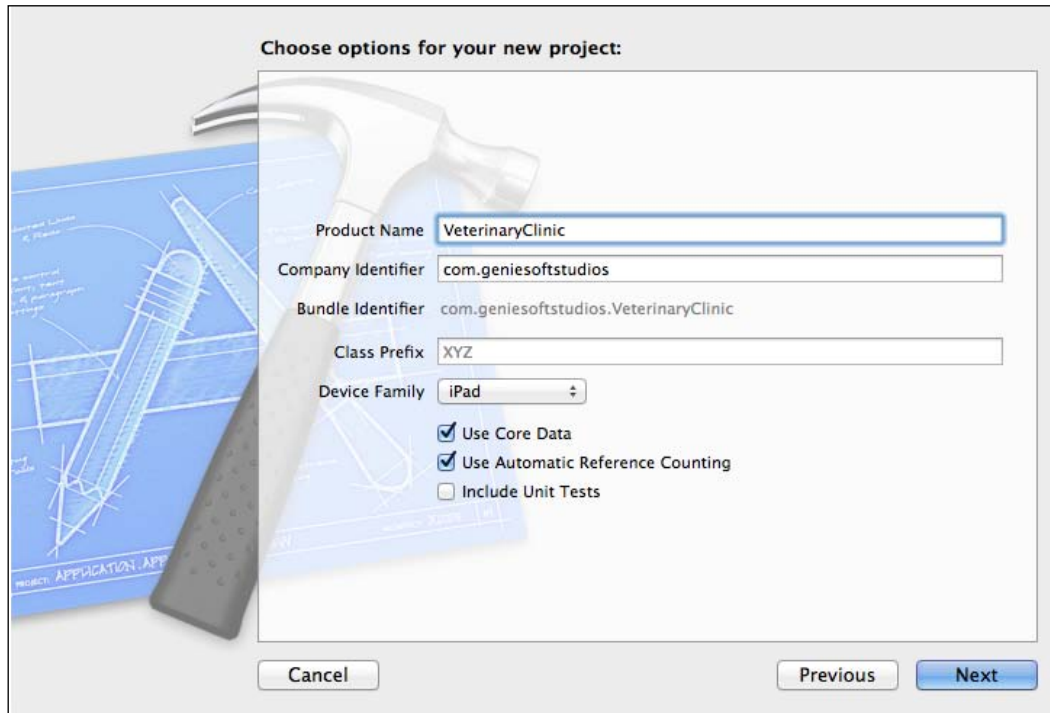
It is very simple to create our VeterinaryClinic application in Xcode. Just follow the steps listed here.

1. Launch Xcode from the /Xcode4/Applications folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the **Empty Application** template from the list of available templates.
4. Click on the **Next** button to proceed with the next step in the wizard.



5. Next, enter in VeterinaryClinic as the name for your project.
6. Select **iPad** from under the **Device Family** drop-down list.
7. Ensure that the **Use Core Data** checkbox has been selected.
8. Ensure that the **Use Automatic Reference Counting** checkbox has been selected.
9. Ensure that the **Include Unit Tests** checkbox has not been selected.

10. Click on the **Next** button to proceed with the next step in the wizard.



11. Specify the location where you would like to save your project.
12. Then, click on the **Create** button to continue and display the Xcode workspace environment.

Now that we have created our `VeterinaryClinic` project, we need to start building our user interface that will be responsible for allowing us to create and add new pet information directly into our list.

## Building the Core Data model

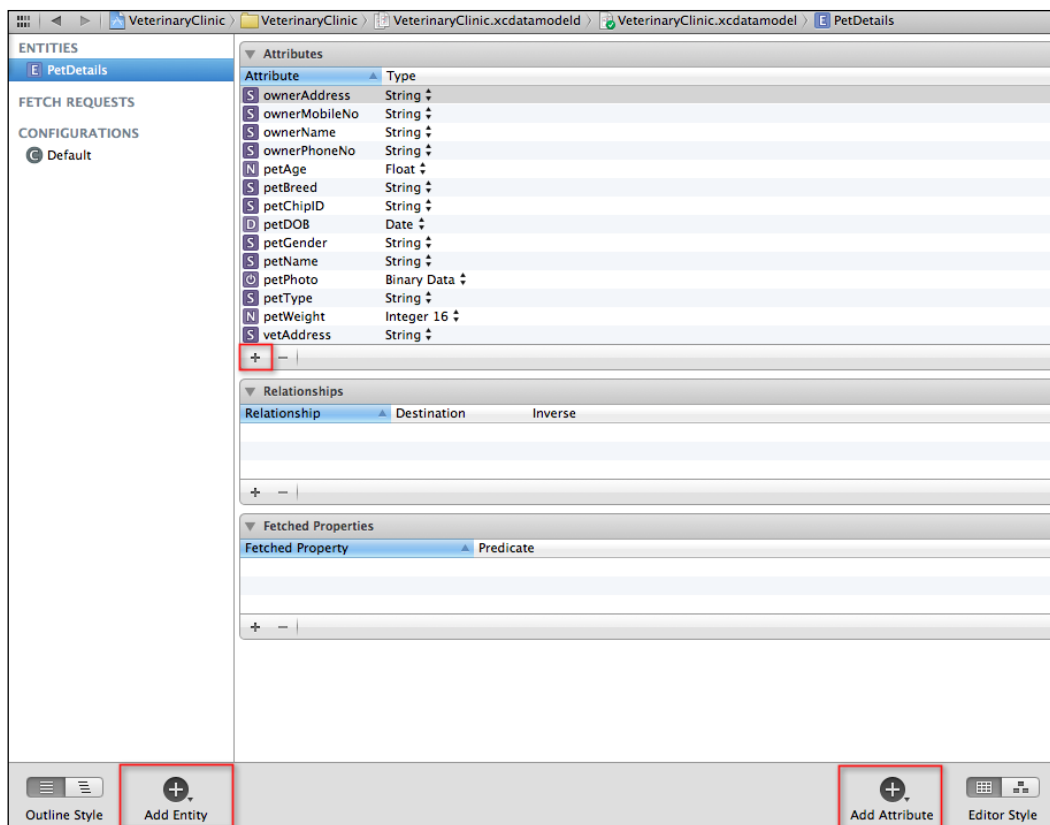
The Core Data database model is stored within the `VeterinaryClinic.xcdatamodeld`, located within the `VeterinaryClinic` group within the **Project Navigator** window. This file will be used to define the database schema for our SQLite database, as we will be defining the entities (table) and the attributes (fields) that make up our `VeterinaryClinic` Data Base.

Since we have selected the **Use Core Data** option, Xcode has automatically set up some important variables and has created the file for us within our project. One thing you will also notice is that we don't need to include `CoreData.framework`, as this has been automatically added for us. Our next step is to create an entity and add the necessary attributes that will enable our application to write to these fields, hence storing this information within the database, so that it can be queried later.

To create a new entity, follow these steps.

1. Select the `VeterinaryClinic.xcdatamodeld` file from **Project Navigator**.
2. Click on the **+ Add Entity** button, located at the bottom left-hand corner of the entity panel, and name this entity `PetDetails`.
3. Next, click on the **+ Add Attribute** button, (located at the bottom right-hand corner of the entity panel or alternatively from the **Attributes** pane, and enter in `ownerAddress` for **Attribute**.
4. Change the attribute type to **String** from the **Type** selection box.
5. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `ownerMobileNo` for **Attribute**.
6. Change the attribute type to `String` from the **Type** selection box.
7. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `ownerName` for **Attribute**.
8. Change the attribute type to **String** from the **Type** selection box.
9. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `ownerPhoneNo` for **Attribute**.
10. Change the attribute type to **String** from the **Type** selection box.
11. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petAge` for **Attribute**.
12. Change the attribute type to **Float** from the **Type** selection box.
13. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petBreed` for **Attribute**.
14. Change the attribute type to **String** from the **Type** selection box.
15. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petChipID` for **Attribute**.

16. Change the attribute type to **String** from the **Type** selection box.
17. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petDOB` for **Attribute**.
18. Change the attribute type to **Date** from the **Type** selection box.
19. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petGender` for **Attribute**.
20. Change the attribute type to **String** from the **Type** selection box.
21. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petName` for **Attribute**.
22. Change the attribute type to **String** from the **Type** selection box.
23. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petPhoto` for **Attribute**.
24. Change the attribute type to **Binary Data** from the **Type** selection box.
25. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petType` for **Attribute**.
26. Change the attribute type to **String** from the **Type** selection box.
27. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `petWeight` for **Attribute**.
28. Change the attribute type to **Integer 16** from the **Type** selection box.
29. Click on the **+ Add Attribute** button, located at the bottom right-hand corner of the entity panel, and enter in `vetAddress` for **Attribute**.
30. Change the attribute type to **String** from the **Type** selection box.
31. Save your project using **File | Save** as we are done defining our database table schema.



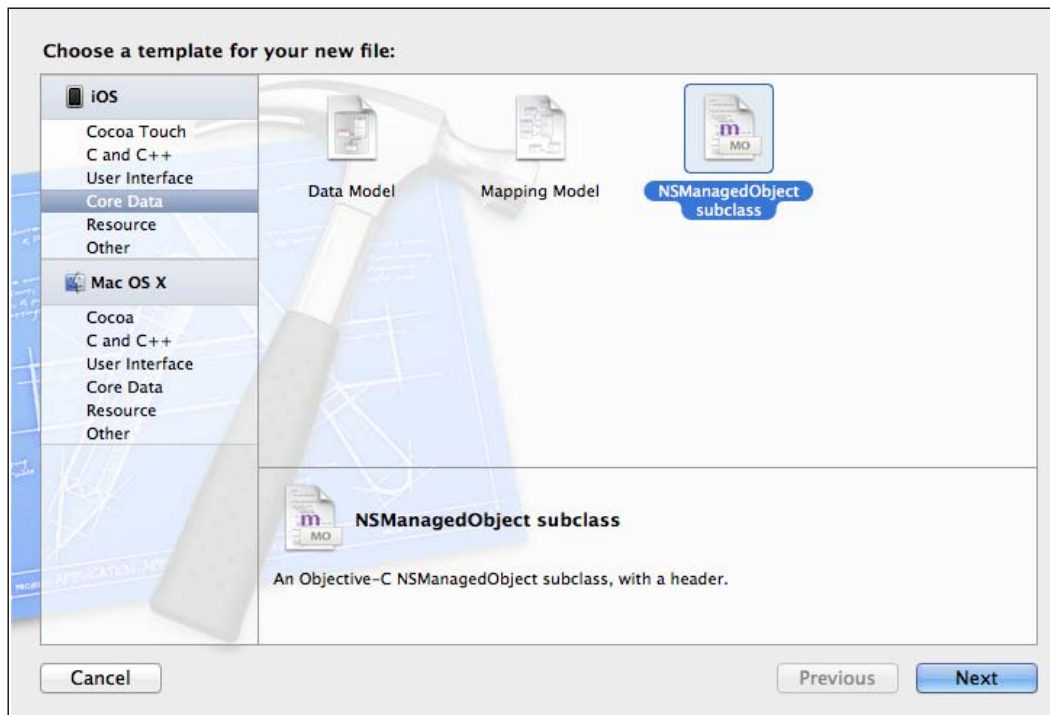
For more information on the Core Data Framework, refer to the Core Data Programming Guide located within the Apple Developer Documentation at [https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html%23//apple\\_ref/doc/uid/TP30001200-SW1](https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html%23//apple_ref/doc/uid/TP30001200-SW1).

So far we have created our `VeterinaryClinic` database model. Our next step is to take a look at how we can integrate and use the database within our application. In the next section, we will look at how to create the core data model files that will allow us to access the table definitions.

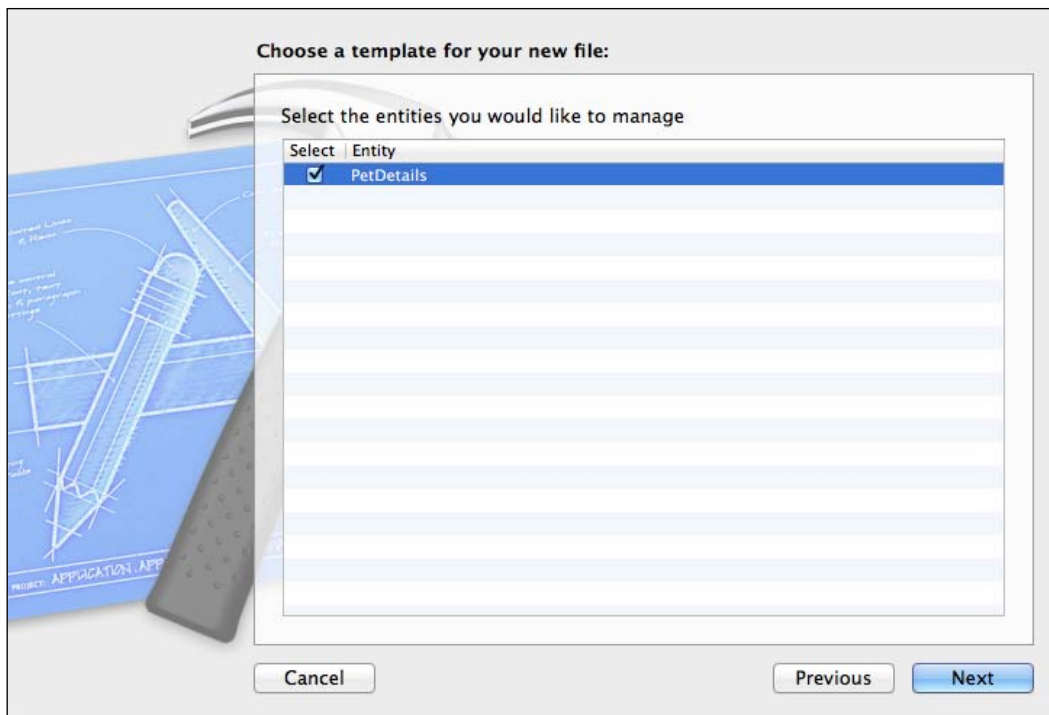
## Creating our Core Data model files

Before our application can start to use our VeterinaryClinic database, we need to create the entity class definitions that will define the variables that the database store contains, so that we can access these through code.

1. From the VeterinaryClinic folder, select the VeterinaryClinic.xcdatamodeld file from **Project Navigator**.
2. Choose **File | New | File...** or press *Command + N*.
3. Select **Core Data** located under the **iOS** header section.
4. Next, select the **NSManagedObject subclass** template from the list of available templates.



5. Click on the **Next** button to proceed with the next step within the wizard.
6. Next, we need to define what entities we want to create the `NSManagedObject` classes for.



7. Select the `PetDetails` entity from the **Select the entities you would like to manage** list.
8. Ensure that the **Use scalar properties for primitive data types** option is not selected.
9. Click on the **Next** button to proceed with the next step in the wizard.
10. Click on the **Create** button to generate the `NSManagedObject` class files.

You will notice that the wizard has created two new files for us, `PetDetails.m` and `PetDetails.h`. These files define the `NSManagedObject` class for the `PetDetails` entity that we created in the Core Data store.



They define the table schema fields for our `PetDetails` class, so that we can access their attributes at runtime. Let's take a quick look at the `PetDetails` class files, to see what the wizard generated for us.

1. Open the `PetDetails.h` interface file, located within the `PetDetails` folder.

```
// PetDetails.h
// VeterinaryClinic
// Created by Steven Daniel on 4/04/12.
// Copyright (c) 2012 GENIESOFT STUDIOS. All rights reserved.

#import<Foundation/Foundation.h>
#import<CoreData/CoreData.h>

@interface PetDetails : NSObject

@property (strong, nonatomic) NSNumber *petAge;
@property (strong, nonatomic) NSString *petBreed;
@property (strong, nonatomic) NSString *petChipID;
@property (strong, nonatomic) NSDate *petDOB;
@property (strong, nonatomic) NSString *petGender;
@property (strong, nonatomic) NSString *petName;
@property (strong, nonatomic) NSData *petPhoto;
@property (strong, nonatomic) NSString *petType;
@property (strong, nonatomic) NSNumber *petWeight;
@property (strong, nonatomic) NSString *vetAddress;
@property (strong, nonatomic) NSString *ownerName;
@property (strong, nonatomic) NSString *ownerAddress;
@property (strong, nonatomic) NSString *ownerPhoneNo;
@property (strong, nonatomic) NSString *ownerMobileNo;

@end
```

From the preceding code snippet, we can see that the wizard has generated a `PetDetails.h` interface file for us, which contains each of our entity attribute fields, with each being declared and assigned their respective object type.



Any changes to the datatypes for the fields relating to the `PetDetails` entity will need to be set using the Core Data schema editor. Any changes made directly to those types above will receive an error when the application launches, as the model is different to what was initially created. The only way to solve this is to delete the application from the iOS device/Simulator and re-compile your application.

2. Next, open the `PetDetails.m` implementation file, located within the `PetDetails` folder.

```
// PetDetails.m
// VeterinaryClinic
// Created by Steven Daniel on 4/04/12.
// Copyright (c) 2012 GENIESOFT STUDIOS. All rights reserved.

#import "PetDetails.h"

@implementation PetDetails

@dynamic petAge;
@dynamic petBreed;
@dynamic petChipID;
@dynamic petDOB;
@dynamic petGender;
@dynamic petName;
@dynamic petPhoto;
@dynamic petType;
@dynamic petWeight;
@dynamic vetAddress;
@dynamic ownerName;
@dynamic ownerAddress;
@dynamic ownerPhoneNo;
@dynamic ownerMobileNo;

@end
```

From the preceding code snippet, we can see that the wizard has generated a `PetDetails.m` implementation file for us, which contains each of our entity attribute fields, with each being declared as `dynamic`. This defines the entity attribute properties, so that they can be used when the data is being written or retrieved from Core Data.



The `@dynamic` directive tells the compiler that you will be creating the method implementations directly through code, as well as suppressing the warnings that the compiler would otherwise generate, if it can't find suitable implementations. You should only use this if you know that the methods will be available at runtime.

## Adding the Storyboard screen

Now that we have created our `VeterinaryClinic` project and created the `CoreData` database, our next step is to include the Storyboard template as part of our `VeterinaryClinic` project. Unfortunately, the Storyboard template is not added as part of the **Empty Application** project template. This template provides you with a starting point for any application, and comes with an application delegate and a window.

To refresh your memory on how to go about adding the Storyboard screen to your project, you can refer to the section that we covered in *Chapter 4, Enhanced AddressBook App – Core Data*, under the section named *Adding the Storyboard screen*.

Follow these simple steps to see how to add the Storyboard template into your application.

1. From the **Project Navigator** window, select the `VeterinaryClinic` folder.
2. Choose **File | New | File...** or press *Command + N*.
3. Select **User Interface**, located under the **iOS** header section.
4. Select **Storyboard** from the list of available templates.
5. Click on the **Next** button to proceed with the next step in the wizard.
6. Ensure that you have selected **iPad** from under the **Device Family** drop-down list.
7. Click on the **Next** button to proceed with the next step of the wizard.
8. Enter in `MainStoryboard` as the name of the file to be created.
9. Click on the **Create** button to save the file to the folder specified.

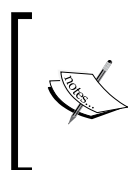
Now that we have created and added our Storyboard to our `VeterinaryClinic` application, our next task is to modify our project so that it is configured to use the Storyboard that we just created.

1. Click and select your project from the **Project Navigator** window.
2. Then, select your project target from under the **TARGETS** group.
3. Select the **Summary** tab.
4. Ensure that you select **MainStoryboard** from the **Main Storyboard** drop-down list.

- Next, open the `AppDelegate.m` implementation file from within **Project Navigator**, and modify the application `didFinishLaunchingWithOptions` method, as shown in the following code snippet:

```
- (BOOL)application:(UIApplication *)application
 didFinishLaunchingWithOptions:(NSDictionary
 *)launchOptions
{
 return YES;
}
```

When using Storyboards, we don't need to create a new `UIWindow` view object as this will create another white window and place this on top of the Storyboard. Now that we have added our Storyboard to our `VeterinaryClinic` application, our next step is to start building our main application.



For more information on using Storyboards, you can refer to the Apple Developer Documentation located at the following URL: [http://developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/000-About\\_Xcode/about.html](http://developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/000-About_Xcode/about.html).

## Creating the main application screen

Our next step is to build the user interface for our `VeterinaryClinic` application. These screens will consist of a Tab Bar controller, a Navigational controller, and View controllers. The Navigational controller enables us to create relationships between each of the other screens within the Storyboard and set up the required connections, known as segues, which represent a transition from one screen to another.

- Select the `MainStoryboard.storyboard` file from **Project Navigator**.
- From **Object Library**, select-and-drag a (`UITabBarController`) **Tab Bar Controller** control, and add this to our Storyboard canvas.
- Next, delete the two `UIViewController`s controllers that Xcode generated in the Storyboard when you dragged-and-dropped the `UITabBarController` control.

To see how to go about adding the `UITabBarController` and TabBar controller, you can refer to *Chapter 4, Enhanced AddressBook App*, under the section named *Creating the main application screen*.

## Adding the table control to hold pet information

Our next step is to add a `UITableViewController` control that will be used to hold and list our pet entries. We will need to include a Navigation controller that will be used to navigate back-and-forth between `UITableViewController` and itself. To see how to go about adding `UITableViewController`, you can refer to *Chapter 4, Enhanced AddressBook App*, under the section named *Adding the table control to hold item data*.

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UITableViewController`) **Table View Controller** control, and add this to our view.

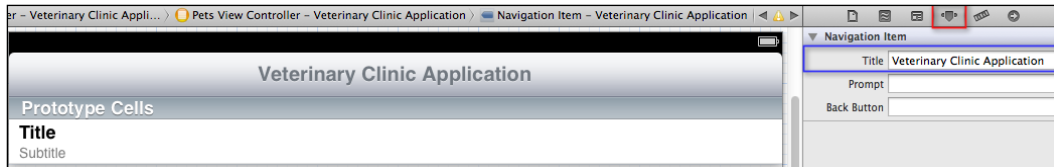
Next, we need to create a Navigation controller between the Tab Bar controller and the `UITableViewController` control that we just added. There are two ways that this can be achieved; you can either drag `UINavigationController` directly onto the view, or you can let Xcode do this for you automatically.

1. Select `UITableViewController` that we just added, and then choose **Editor | Embed In | Navigation Controller** from the **Editor** menu.
2. Select **Tab Bar Controller**, then hold down *Ctrl* and drag from **Tab Bar Controller** to **Navigation Controller**, and release the mouse.
3. Next, choose **root view controller** from the **Relationship Segue** pop up.
4. Next, we want to show the toolbar within our **Navigation Controller**. Select **Navigation Controller**, and from the **Attributes Inspector** dialog box, select the **Shows Toolbar** and **Shows Navigation Bar** options.
5. Then, change the **Bottom Bar** option under **Simulated Metrics** to read **Inferred**.

So far, we have linked up our Tab Bar controller and Navigation controllers, and have configured the properties required for the Navigation controller; our next step is to set up the properties on our Table View controller. Follow these simple steps below:

1. Select the Table View controller that we just added.
2. Next, click on the toolbar located at the top of the View controller.

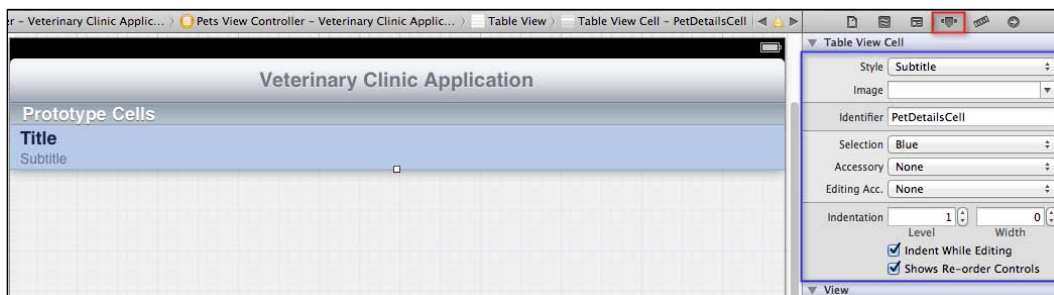
- Then, from **Attributes Inspector**, change **Title** to read **Veterinary Clinic Application**, as shown in the following screenshot:



If you prefer, you can also double-click the navigation bar and change its title. As we have seen in a previous chapter, you will notice that since we added our Table View controller, Xcode gave us a warning.

This happens whenever you add a Table View Controller to an existing or new storyboard; this is due to it wanting to use prototype cells as the default type. In the following screenshot, we will take a look at how to properly configure this.

- Click inside the **Table View** cell under the **Prototype Cells** header.
- From the **Attributes Inspector** section, change **Style** to **Subtitle**. This will change the cells' appearance to contain two labels.
- Select the **Identifier** item and enter in `PetDetailsCell` as its unique identifier. You will notice that once this has been entered in, Xcode will stop complaining about the warning message we received earlier on.
- Set the **Accessory** attribute to show **None**.

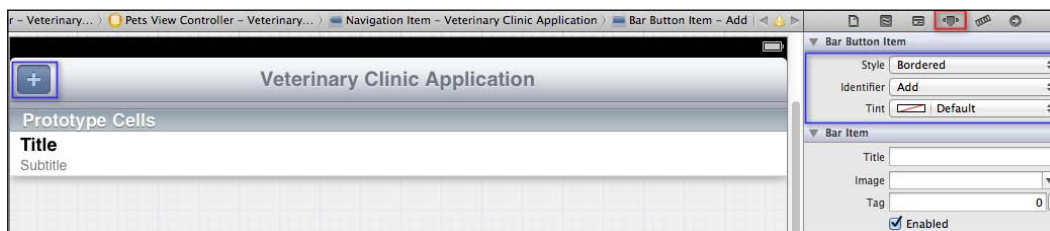


Now that we have successfully configured our table cell, we need to start adding a button to our Table View header that will be responsible for allowing new pet details to be added to our Core Data database.

## Adding the Add button

Our next step is to add a button to our `UITableViewController` control that will be responsible for displaying an additional screen where we can create pet record details. This can be achieved by following these simple steps:

1. Select the **MainStoryboard.storyboard** file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top left-hand corner of the navigation bar on our **Table View Controller** screen that we added previously.
3. From the **Attributes Inspector** section, change **Identifier** to **Add**.
4. Then, change the **Style** to **Bordered**.

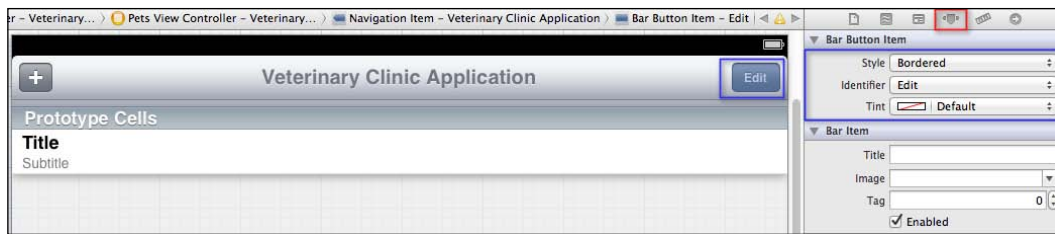


Now that we have added our **Add** button to the **Pets View Controller**, our next step is to add the **Edit** button that will be responsible for editing an existing item within the list when the button is clicked. So let's proceed with the next section.

## Adding the Edit button

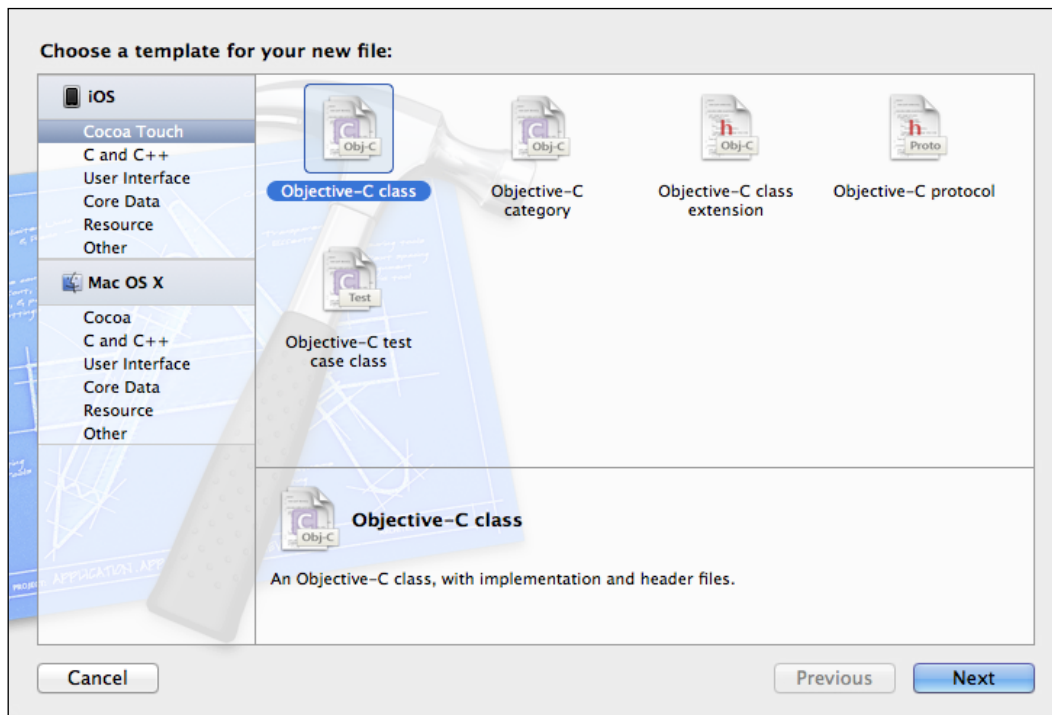
Now that we have added our button to add a new pet record, our next step is to add another button to `UITableViewController`; this will be responsible for allowing the user to make changes to an existing pet item within the table view. This can be achieved by following these simple steps:

1. Select the **MainStoryboard.storyboard** file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top right-hand corner of the navigation bar on the **Veterinary Clinic (UITableViewController)** section of the **Table View Controller** screen that we added previously.
3. From the **Attributes Inspector** section, change **Identifier** to **Edit**.
4. Then, change the **Style** to **Bordered**.



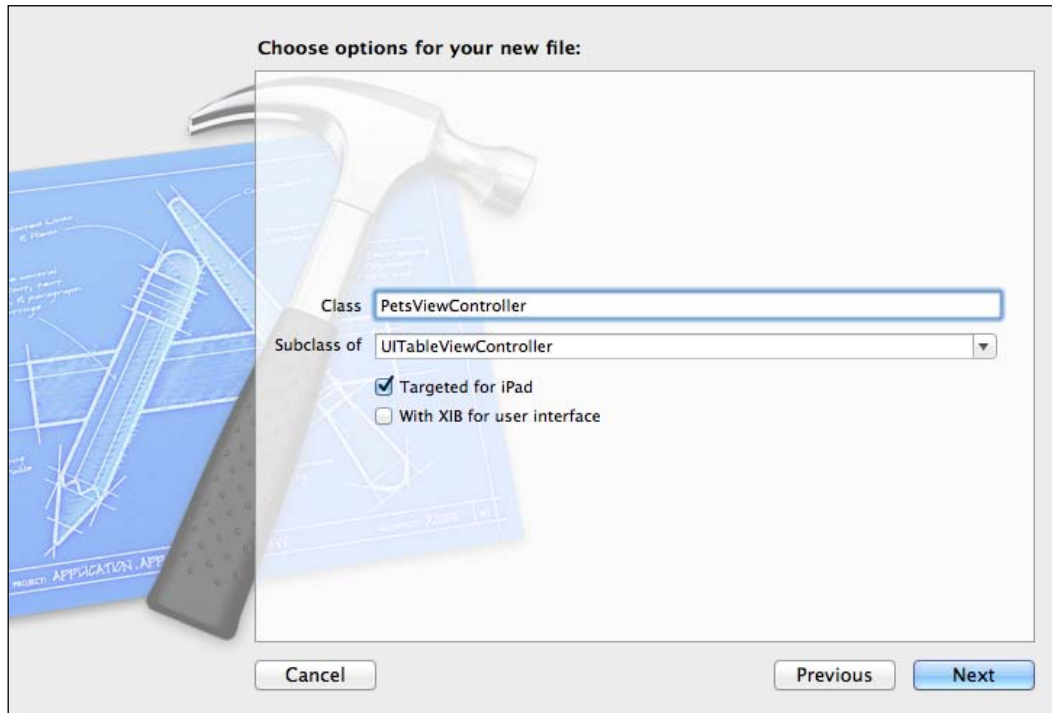
Now that we have added our **Add** and **Edit** buttons, properly configured our **Table View Controller**, and built our user interface, our next step is to create our very own custom `UIViewController` sub-class that will act as the data source for our table, so that it will know how many rows to display when it retrieves the pet details information from our database.

1. Select the `VeterinaryClinic` folder, choose **File | New | New File...** or press *Command + N*.
2. Select **Cocoa Touch** located under the **iOS** header section.
3. Next, select the **Objective-C** class template from the list of available templates.





4. Click on the **Next** button to proceed with the next step within the wizard.
5. Ensure that you have selected `UITableViewController` as the type of subclass to be created from the **Subclass of** dropdown.
6. Enter in `PetsViewController` as the name of the file to be created.
7. Ensure that you have selected the **Targeted** for the **iPad** option.



8. Click on the **Next** button to proceed with the next step of the wizard.
9. Then, click on the **Create** button to save the file to the folder location specified.

Now that we have added our `ViewController` class to our `VeterinaryClinic` application, our next step is to update the class of `UITableViewController` to use this class, instead of the default `UITableViewController` class.

1. Select the **MainStoryboard.storyboard** file from **Project Navigator**.
2. Select the Table View controller that we added to our `VeterinaryClinic` application.
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `PetsViewController`.

Our next step is to add a reference to the `NSManagedObjectContext` and `NSFetchedResultsController` objects as well as create an `NSMutableArray` array property within our `PetsViewController` interface file.

1. Open the `PetsViewController.h` interface file, located within the `VeterinaryClinic` folder. Enter in the following code snippet:

```
// PetsViewController.h
// VeterinaryClinic
// Created by Steven F. Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import "PetDetails.h"

@interface PetsViewController : UITableViewController
{
 NSManagedObjectContext *managedObjectContext;
 NSFetchedResultsController *fetchedResultsController;

 // Used for our selected table view item.
 NSMutableArray *petListArray;
 NSArray *fetchedObjects;
}

@property (strong, nonatomic) NSManagedObjectContext
*managedObjectContext;
@property (strong, nonatomic) NSFetchedResultsController
*fetchedResultsController;
@property (strong, nonatomic) NSMutableArray
*petListArray;

- (void)getPetDetails;

@end
```

As you can see, all we have done is created a reference to the `NSManagedObjectContext` and `NSFetchedResultsController` objects that will help us with managing the fetching, updating, and creating of records within the data store.

These objects also have the ability to handle validations and undo/redo managements of records. Next, we declared an `NSMutableArray` object, that will be used to hold each of our pet detail objects that we will create, and this will also be used to act as a data source to our **Veterinary Clinic Application** table view. It is a good practice to save the file at this point in the process, to avoid loss of changes.



For more information about the NSMutableArray object, you can refer to the Apple Developer documentation at the provided URL: [http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray\\_Class/Reference/Reference.html](http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/Reference/Reference.html).

2. Next, open the `AppDelegate.m` implementation file, located within the `VeterinaryClinic` folder, and add the following highlighted code:

```
// AppDelegate.m
// VeterinaryClinic
// Created by Steven F. Daniel on 26/03/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.
//
#import "AppDelegate.h"
#import "PetsViewController.h"
```

```
@implementation AppDelegate
@synthesize window = _window;
```

In the preceding code snippet, we need to import the `PetsViewController.h` interface header file, as we will be referencing these when we set up our data source for `PetsViewController` within our Storyboard.

3. Next, we need to change the `didFinishLaunchingWithOptions:` method, located within the `AppDelegate.m` implementation file.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions
{
 UITabBarController *tabBarController =
 (UITabBarController *)self.window.rootViewController;

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

 PetsViewController *petsViewController = [[
 navigationController.viewControllers] objectAtIndex:0];

 petsViewController.managedObjectContext =
 self.managedObjectContext;

 return YES;
}
```

In the preceding code snippet, we need to initialize the data source for `PetsViewController` using the `managedObjectContext` method. This will ensure that our controller has access to all of the required properties and methods required to add and retrieve the information from our data store.

Before this can happen, we must first cycle through each scene within our Storyboard in order to get a reference to `PetsViewController`. This is so that we can initialize its data source, so that it points to our database. Next, we need to populate our pet information to our table view.

4. Open the `PetsViewController.m` implementation file, and enter in the following highlighted code snippets:

```
// PetsViewController.m
// VeterinaryClinic
//
// Created by Steven F. Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "PetsViewController.h"

@implementation PetsViewController

@synthesize fetchedResultsController;
@synthesize managedObjectContext;
@synthesize petListArray;
```

5. Next, we need to modify the `viewDidLoad` method located within the `PetsViewController.m` implementation file, and enter in the following highlighted code snippet:

```
#pragma mark - View lifecycle
- (void)viewDidLoad
{
 [super viewDidLoad];

 // Initialize and reload our pet information.
 self.navigationController.navigationBar.tintColor =
 [UIColor purpleColor];
 [self getPetDetails];
}
```

In the preceding code snippet, we need to initialize and set the color of our Navigational controller bar to purple and then call the `getPetDetails` method to populate the database object items to our table view.

6. Next, open the `PetsViewController.m` implementation file, and enter in the following code snippet for the `getPetDetails` method:

```
#pragma mark Populate our UITableView Controller with all records
in our database.
- (void) getPetDetails
{
 // Define our table/entity name to use
 NSEntityDescription *entity = [NSEntityDescription
 entityForName:@"PetDetails"
 inManagedObjectContext:managedObjectContext];

 // Set up the fetch request
 NSFetchedRequest *fetchRequest = [[NSFetchedRequest alloc]
 init];
 [fetchRequest setEntity:entity];

 // Define how we are to sort the records
 NSSortDescriptor *sortDescriptor = [[NSSortDescriptor
 alloc] initWithKey:@"petName" ascending:NO];
 NSArray *sortDescriptors = [NSArray
 arrayWithObject:sortDescriptor];
 [fetchRequest setSortDescriptors:sortDescriptors];

 // Define the FetchResults controller
 fetchedResultsController = [[NSFetchedResultsController
 alloc] initWithFetchRequest:fetchRequest
 managedObjectContext:managedObjectContext
 sectionNameKeyPath:nil cacheName:@"Root"];

 // Fetch the records
 NSError *error;
 NSMutableArray *mutableFetchResults = [[
 managedObjectContext executeFetchRequest:fetchRequest
 error:&error] mutableCopy];
 if (!mutableFetchResults)
 {
 // Something seriously went wrong, so notify the
 // user.
 NSLog(@"There was an error retrieving the Veterinary
 Clinic records.");
 }
 // Save our fetched record to the array
 [self setPetListArray:mutableFetchResults];
 [self.tableView reloadData];
}
```

In the preceding code snippet, we define the table entity that we want to use as our main data source and then create an instance to our `fetchRequest` object that will be used to hold the returned items. Next, we then specify that we would like to have the results sorted on `petName` in descending order, so that the latest entry appears at the top of the list.

7. We then initialize our `fetchResultsController` object, in order for it to start retrieving the data from our database, then execute the record set and check for any errors that occurred using the `mutableFetchResults` method.
8. Finally, we save the result set to our `petListArray` property, and then call the `reloadData` method on our table View control to redisplay the records.
9. Next, we need to modify the `viewDidAppear` method that is located within the `PetsViewController.m` implementation file to refresh our Table view whenever the view is displayed. Create the `viewDidAppear` method, and enter in the following highlighted code snippets:

```
- (void)viewDidAppear:(BOOL)animated
{
 [super viewDidAppear:animated];
 [self getPetDetails];
}
```

10. Next, we need to change the table view data source methods that are located within the `PetsViewController.m` implementation file, and enter in the following highlighted code snippets:

```
#pragma mark - Table view data source
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
 // Return the number of sections.
 return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
 // Return the number of rows in the section.
 if ([fetchObjects count] == 0)
 {
 return [self.petListArray count];
 }
 else
 {
 // Handled by the search feature
 }
}
```

```
 return [fetchedObjects count];
 }
}
```

From the preceding code snippet, we can see that we set the number of table sections, and then have the `numberOfRowsInSection` method work out how many rows that will exist in each section. This is achieved by using the `count` property of our `petListArray` array object.

```
- (UITableViewCell *)tableView:(UITableView *)
tableViewcellForRowAtIndexPath:(NSIndexPath *)indexPath
{
 static NSString *CellIdentifier = @"PetDetailsCell";
 PetDetails *pet;

 // Check to ensure that we have items in our list.
 if ([fetchedObjects count] == 0)
 {
 pet = [petListArray objectAtIndex:indexPath.row];
 }
 else
 {
 // This section gets called when we are
 // using the search functionality.
 pet = [fetchedResultsController
 objectAtIndex:indexPath];
 }

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

 // Configure the cell...
 cell.textLabel.text = [NSString
 stringWithFormat:@"%@ %@, %@", pet.petName,
 pet.petType, pet.petGender];

 cell.detailTextLabel.text = pet.petBreed;
 UIImage *petPhoto = [UIImage
 imageDataWithData:pet.petPhoto];
 cell.imageView.image = petPhoto;

 return cell;
}
```

Finally, as you can see in the preceding code snippet, we supply the reuse identifier of the `TableViewCell` cell that we set up previously, then assign each of the properties from our pets list array, and write it to the cell labels.



When you reference the reuse identifier as a parameter to the method called `dequeueReusableCellWithIdentifier`, it will automatically make a new copy of the prototype and return the object back to you. For more information on this method, you can refer to the `UITableView` class reference at the following URL  
[http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableView\\_Class/Reference/Reference.html](http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableView_Class/Reference/Reference.html).

11. Now that we have set up the data source correctly for our Table view, we can run our application by choosing **Product | Run** from the **Product** menu, or alternatively press *Command + R* to see the application running on an iOS device, as shown in the following screenshot:



Now that we have successfully configured our data source for our list of pet information, we will see how we can navigate between the screens within the Storyboard. We will learn about segues, and the different types of views they can take on. We will look into static table view cells, as well as how to go about providing the ability for additional pet information to be added to the current list of pets within our veterinary clinic.



## Navigating between screens using Storyboards

In this section, we will be adding more view controllers to our Storyboard to allow the flexibility of adding new pet detail record information to our existing table view.

In order for us to transition between screens within our Storyboard, we need to create a connection, known as a segue. Segues are defined as having the ability to only go one way; they cannot go back to the previous screen, unless a delegate class has been set up.

For our new screen, we will be creating a "modal" segue. A **modal segue** is a screen that becomes the active screen that prevents the user from interacting with the underlying screen until they close the modal screen first.

To begin creating the **Add New Pet** screen, follow these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a new (UIViewController) **View Controller** control, and add this to our Storyboard to the right of the **Veterinary Clinic Application** screen.
3. Next, select UIViewController that we just added, and then choose **Editor | Embed In | Navigation Controller** from the **Editor** menu.
4. Next, select the + button that we added previously, and hold down the *Control* key while dragging it to the new **Navigation Controller**, and release the mouse button.
5. Finally, select **Modal** from the pop-up list of choices.
6. Next, we want to show the toolbar within our Navigation controller. Select the Navigation controller, and from the **Attributes Inspector** dialog box, select the **Shows Toolbar** and **Shows Navigation Bar** options.
7. Then, change the **Bottom Bar** option under **Simulated Metrics** to read **Toolbar**.

When you select **Modal** from the list of Storyboard Segues, a new arrow gets placed between the **Veterinary Details** screen and the Navigational controller. So, when you press the + button, a new screen will be displayed on the screen.

Next, we need to specify an identifier for our Storyboard Segue. This will be responsible for handling the cancelling and saving methods when the **Add New Pet Details** form is closed.

1. Select the segue relationship that is located between the **Veterinary Clinic Application** screen and UINavigationController for the **Add New Pet Details** screen.

2. Click on the **Attributes Inspector** button.
3. Change the **Identifier** property to `AddPetDetails`.
4. Change the **Style** property to **Modal**.
5. Change the **Presentation** property to **Default**.
6. Change the **Transition** property to **Cross Dissolve**.

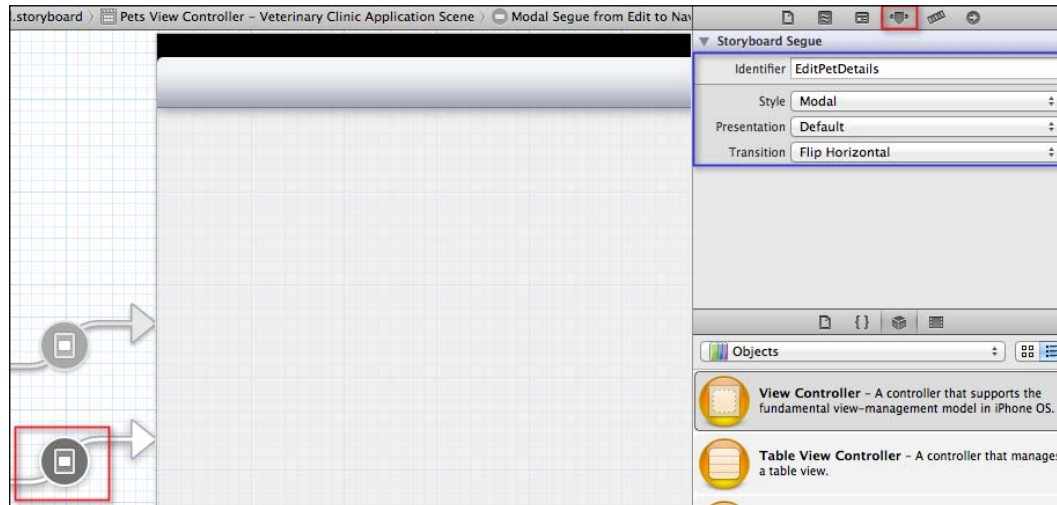


Next, we need to apply the same logic as we did for the **Add New Pet Details** form that will be responsible for calling the same form to handle the editing of an existing record when the **Edit** button has been pressed.

To begin creating the **Edit Pet Details** screen, follow these simple steps:

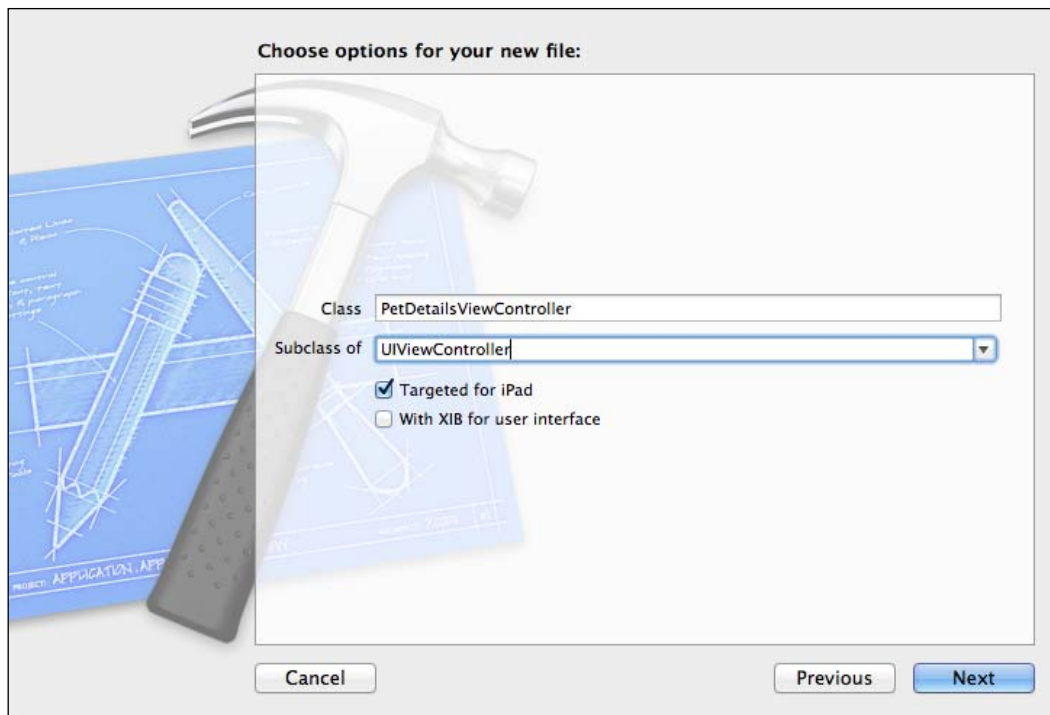
1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Next, select the **Edit** button that we added to `UITableViewController`, and hold down the **Control** key while dragging it to the same Navigation controller as our **Add** button, and release the mouse button.
3. Finally, select **Modal** from the pop-up list of choices.
4. Next, select the segue relationship that we just created for our **Edit Pet Details** screen.
5. Click on the **Attributes Inspector** button.
6. Change the **Identifier** property to `EditPetDetails`.
7. Change the **Style** property to **Modal**.
8. Change the **Presentation** property to **Default**.

9. Change the **Transition** property to **Flip Horizontal**.



Unfortunately, you won't be able to go back to the previous screen until we create a `UIViewController` subclass, the same as what we did for `PetsViewController`.

1. From the `VeterinaryClinic` folder, choose **File | New | File...** or press *Command + N*.
2. Select **Cocoa Touch** located under the **iOS** header section.
3. Select the **Objective-C** class template from the list of templates.
4. Click on the **Next** button to proceed with the next step within the wizard.
5. Enter in `PetDetailsViewController` as the name of the file to create.
6. Ensure that you select `UIViewController` as the type of subclass to be created from the **Subclass of** drop-down list.
7. Ensure that you have selected the **Targeted for iPad** option.

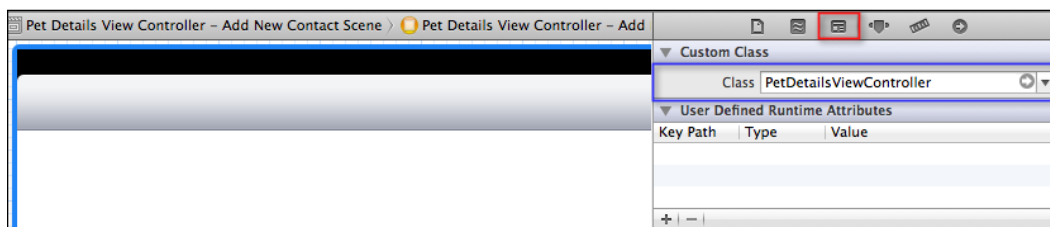


8. Click on the **Next** button to proceed with the next step of the wizard.
9. Then, click on the **Create** button to save the file to the folder location specified.

Once you have done this, we need to update the class method of our **Add Pet Details** screen of `UIViewController` to use our new view controller subclass. Follow these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Click and select our **Add Pet Details** (`UIViewController`) within our Storyboard.

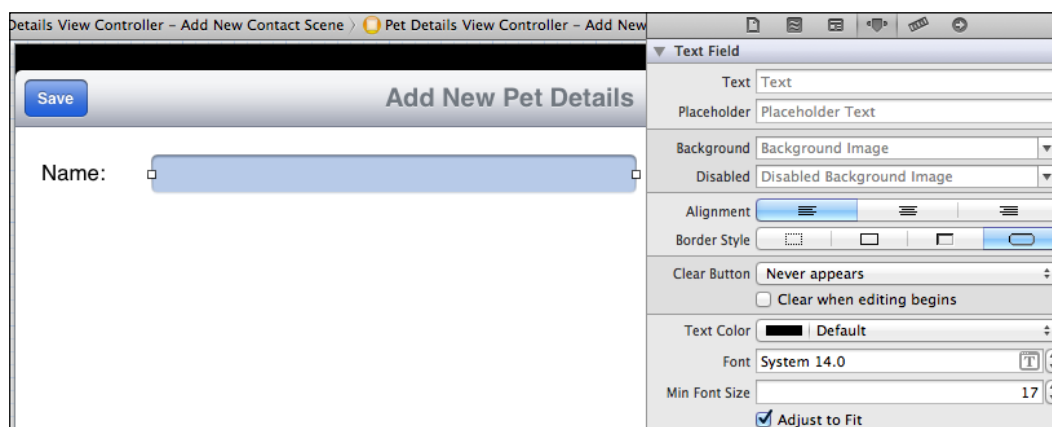
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `PetDetailsViewController`.



4. Next, from the **Attributes Inspector** section, change the **Title** property to read **Add New Pet Details**.
5. From **Object Library**, select-and-drag a **Bar Button Item** (`UIBarButtonItem`) control to the top left-hand corner of the navigation bar on the **Add New Pet** (`UIViewController`) View controller screen that we added previously.
6. From the **Attributes Inspector** section, change the **Identifier** property to **Save**.
7. Then, change the **Style** property to **Bordered**.
8. Next, from **Object Library**, select-and-drag a **Bar Button Item** (`UIBarButtonItem`) control to the top right-hand corner of the navigation bar.
9. From the **Attributes Inspector** section, change the **Identifier** property to **Cancel**.
10. Change the **Style** property to **Bordered**.
11. Next, from **Object Library**, select-and-drag a **Bar Button Item** (`UIBarButtonItem`) control to the bottom left-hand corner of the toolbar.
12. From the **Attributes Inspector** section, change the **Title** property to **Photo Library**.
13. Change the **Style** property to **Bordered**.
14. Next, from **Object Library**, select-and-drag a **Bar Button Item** (`UIBarButtonItem`) control to the bottom left-hand corner of the toolbar.
15. From the **Attributes Inspector** section, change the **Identifier** property to **Camera**.
16. Change the **Style** property to **Bordered**.

Our next step is to start building the screen that will allow us to enter information relating to each pet, so that it can be saved to the **Veterinary Clinic** list.

1. Select the **Add New Pet Details** View controller from within our Storyboard.
2. Next, drag a **Label Field** (UILabel) control onto the canvas.
3. Select **Attributes Inspector** for **Text Field**.
4. Set the **Text** field property to read **Name:**.
5. Set the **Alignment** field property to **Left Justify**.
6. Next, drag a **Text Field** (UITextField) control next to the **Name** field that we added in the previous step.
7. Select **Attributes Inspector** for **Text Field**.
8. Set the **Alignment** field property to **Left Justify**.
9. Set the **Border Style** property to **Rounded**.
10. Set the **Font** to **System 14.0**.
11. Adjust the size of the control by dragging the right edge of the control.




12. Ensure that you have unchecked the **Adjust to Fit** checkbox.
13. Repeat steps 2 to 11 to add the following control: **Photo:** (UIImageView), **Age:** (UITextField), **DOB:** (UITextField), **Type:** (UITextField), **Breed:** (UITextField), **Gender:** (UITextField), **Weight:** (UITextField), **Chip ID:** (UITextField), **Vet Address Details:** (UITextView), **Name:** (UITextField), **Address:** (UITextView), **Phone (H):** (UITextField), **Phone (M):** (UITextField).

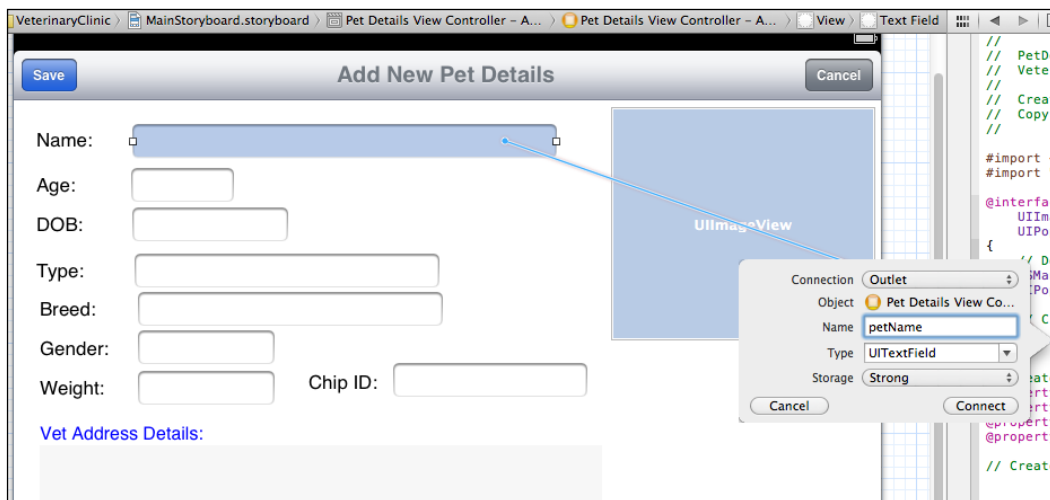
If you have followed the steps correctly, the completed **Add New Pet Details** screen should look similar to the following screenshot; feel free to adjust yours accordingly if it doesn't:

The next step is to create the outlets for each of the fields that we previously added to our **Add New Pet Details** form.

1. Open **Assistant Editor** by choosing **Navigate | Open In Assistant Editor** or press *Option + Command + ,*.
2. Ensure that the `PetDetailsViewController.h` interface file gets displayed.
3. Select **Name** (`UITextField`), hold down the *Control* key, and drag it into the `PetDetailsViewController.h` interface file.

 In order to create the IBOutlet, outlets, these will need to be created inside the curly braces { } under the @interface directive as these are not created by default.

4. Enter in petName for the name of the property to be created.
5. Choose **Strong** from the **Storage** drop-down list.



6. Repeat steps 3 to 5 and create the properties for the **Photo**, **Age**, **DOB**, **Type**, **Breed**, **Gender**, **Weight**, **Chip ID**, **Vet Address Details**, **Owner Details Name**, **Owner Details Address**, **Phone (H)**, and **Phone (M)** fields.

Now that we have created the outlets and properties for each of our form fields, we need to start modifying our `PetDetailsViewController.h` interface file. Here, we will add a reference to the `NSManagedObjectContext` and `NSFetchedResultsController` objects that will provide us with all of the Core Data fetch-related functions we need to perform when populating the Table view with data. These functions encapsulate the common functions that are associated with the table, and the Core Data data model.

1. Open the `PetDetailsViewController.h` interface file, located within the `VeterinaryClinic` folder, and enter in the following highlighted code snippets:
- ```
// PetDetailsViewController.h
// VeterinaryClinic
// Created by Steven F. Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.
```



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

@interface PetDetailsViewController : UIViewController<UIImagePickerControllerDelegate, UINavigationControllerDelegate,
UIPopoverControllerDelegate>
{
    // Declare our Core Data Objects
    NSManagedObjectContext *managedObjectContext;
    UIPopoverController *popoverController;

    // Create the required form Outlets
    IBOutlet UITextField *petName;
    IBOutlet UITextField *petAge;
    IBOutlet UITextField *petDateofBirth;
    IBOutlet UITextField *petType;
    IBOutlet UITextField *petBreed;
    IBOutlet UITextField *petGender;
    IBOutlet UITextField *petWeight;
    IBOutlet UITextField *petChipID;
    IBOutlet UIImageView *petPhoto;
    IBOutlet UITextView *vetAddress;
    IBOutlet UITextField *ownerName;
    IBOutlet UITextView *ownerAddress;
    IBOutlet UITextField *ownerHomePhone;
    IBOutlet UITextField *ownerMobilePhone;
}

// Create the required class Setters and Getters

@property (strong, nonatomic) NSManagedObjectContext
    *managedObjectContext;
@property (strong, nonatomic) UIPopoverController
    *popoverController;
@property (strong, nonatomic) UIImagePickerController
    *imagePicker;
@property (strong, nonatomic) PetDetails *currentPet;

// Create the required form properties for the Outlets
@property (strong, nonatomic)
    IBOutletUITextField*petName;
@property (strong, nonatomic) IBOutlet UITextField
    *petAge;
@property (strong, nonatomic) IBOutlet UITextField
```

```

        *petDateofBirth;
@property (strong, nonatomic) IBOutlet UITextField
        *petType;
@property (strong, nonatomic) IBOutlet UITextField
        *petBreed;
@property (strong, nonatomic) IBOutlet UITextField
        *petGender;
@property (strong, nonatomic) IBOutlet UITextField
        *petWeight;
@property (strong, nonatomic) IBOutlet UITextField
        *petChipID;
@property (strong, nonatomic) IBOutlet UIImageView
        *petPhoto;
@property (strong, nonatomic) IBOutlet UITextView
        *vetAddress;
@property (strong, nonatomic) IBOutlet UITextField
        *ownerName;
@property (strong, nonatomic) IBOutlet UITextView
        *ownerAddress;
@property (strong, nonatomic) IBOutlet UITextField
        *ownerHomePhone;
@property (strong, nonatomic) IBOutlet UITextField
        *ownerMobilePhone;

// Declare our class Instance methods
- (IBAction)btnSavePet:(id)sender;

@end

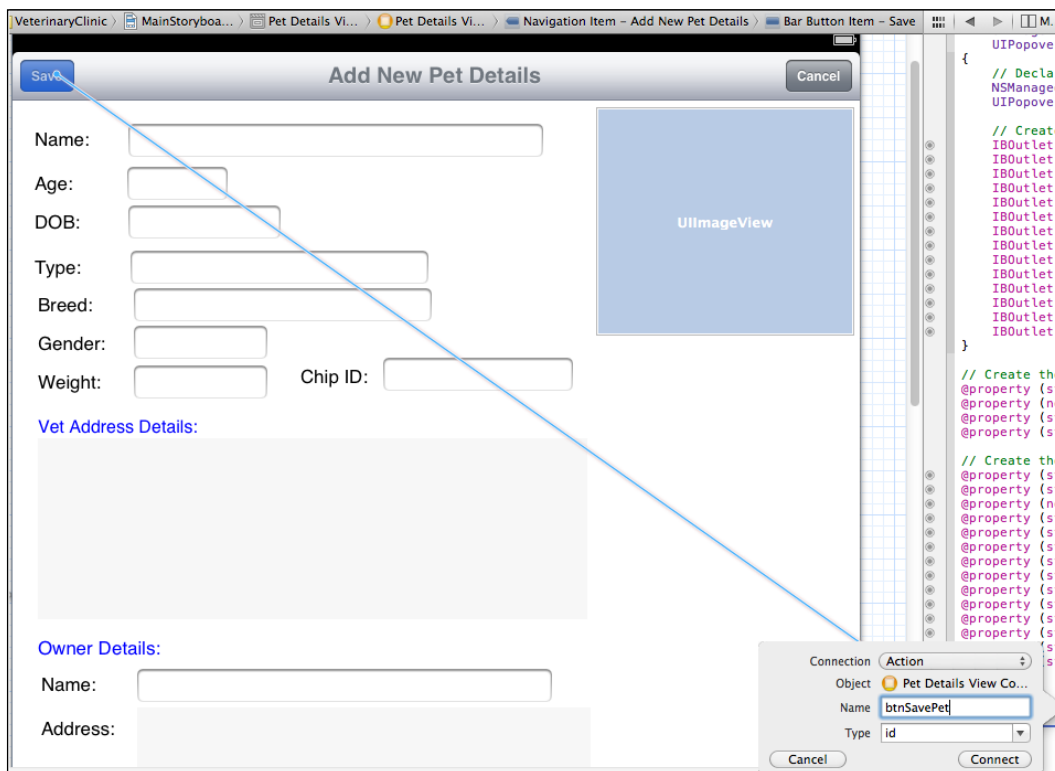
```

In the preceding code snippet, we have created a reference to our `NSManagedObjectContext` that will help us with managing the fetching, updating, and creating of records within the data store. These objects also come with the added advantage and ability to handle validations, and undo/redo functionality of records without having to write any additional code.

We will also need to extend our class so that we can include each of the following class protocols: `UIImagePickerControllerDelegate`, `UINavigationControllerDelegate`, and `UIPopoverControllerDelegate`. This is so that we can access each of their respective methods.

Now that we have created the instance variable Outlets for our controls, we now need to create the associated Actions for those Outlets events. Creating these actions allows an event to be fired when a button has been pressed. To create an Action, follow these simple steps:

1. With the PetDetailsViewController.h interface file still displayed to the right of the **Add New Pet Details** screen, select the **Save** (UIBarButtonItem) control, then hold down the **Control** key, and drag it into the ContactDetailsViewController.h interface file.
2. Choose **Action** from the **Connection** drop-down list for the connection to be created.
3. Enter in btnSavePet for the name of the method to be created.



4. Repeat steps 2 to 4 and hook up the **Cancel**, **Photo Album**, and **Camera** button, creating the following Action event(s): btnCancel, btnAddPhoto, and btnCameraPhoto, respectively.

In the next section, we will take a look at building the functionality for our veterinary clinic, so that it has the ability to add new and edit existing pet information to the **Veterinary Clinic** list.

Functionality

Well done! You have finally made it this far; we have successfully finished building the user interface for both the **Veterinary Clinic Application** and **Add New Pet** screens. Our next step is to start implementing the methods that will be used for our **Save**, **Cancel**, **Photo Library**, and **Camera** buttons. These will be responsible for returning us back to the **Veterinary Clinic Application** screen as well as having the ability to choose photos from the iOS device photo library to use as our pet's photo image.

1. Now, open the `PetDetailsViewController.m` implementation file, located within the `Veterinary` folder, and modify the `viewDidLoad` method, as shown in the following code snippet:

```
// PetDetailsViewController.m
// VeterinaryClinic
// Created by Steven F. Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import "PetDetailsViewController.h"

@implementation PetDetailsViewController

@synthesize managedObjectContext;
@synthesize popoverController;
@synthesize imagePicker;
@synthesize currentPet;

@synthesize petName;
@synthesize petAge;
@synthesize petDateofBirth;
@synthesize petType;
@synthesize petBreed;
@synthesize petGender;
@synthesize petWeight;
@synthesize petChipID;
@synthesize petPhoto;
@synthesize vetAddress;
@synthesize ownerName;
@synthesize ownerAddress;
```

```
@synthesize ownerHomePhone;
@synthesize ownerMobilePhone;

#pragma mark - View lifecycle
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Initialize and reload contacts.
    self.navigationController.navigationBar.tintColor =
        [UIColor blueColor];

    // If we are editing an existing picture, then put the
    // details from Core Data into the text fields for
    // displaying.
    if (self.currentPet)
    {
        NSDateFormatter *dateFormat =
            [[NSDateFormatter alloc] init];
        [dateFormat setDateFormat:@"yyyy-MM-dd"];
        NSString *theDOB = [dateFormat
            stringFromDate:[self.currentPet petDOB]];

        // Grab the Pet Detail Information
        [petName setText:[self.currentPet petName]];
        [petAge setText:[NSString
            stringWithFormat:@"%d", [self.currentPet petAge]]];
        [petDateOfBirth setText:theDOB];
        [petType setText:[self.currentPet petType]];
        [petBreed setText:[self.currentPet petBreed]];
        [petGender setText:[self.currentPet petGender]];
        [petWeight setText:[NSString
            stringWithFormat:@"%d", [
                self.currentPet petWeight]]];
        [petChipID setText:[self.currentPet petChipID]];
        [vetAddress setText:[self.currentPet vetAddress]];

        // Check to see if we have a photo previously added,
        // if yes, display this.
        if ([self.currentPet petPhoto])
        {
            [petPhoto setImage:[UIImage
                imageDataWithData:[self.currentPet petPhoto]]];
        }
    }
}
```

```

        // Grab the Owner Contact Details
        [ownerName setText:[self.current PetownerName]];
        [ownerAddress setText:[self.currentPet
            ownerAddress]];
        [ownerHomePhone setText:[self
            .currentPet ownerPhoneNo]];
        [ownerMobilePhone setText:[self.currentPet
            ownerMobileNo]];
    }
    // Set the UIKeyboardStyles for each of our field
    //controls
    petName.keyboardType= UIKeyboardTypeAlphabet;
    petAge.keyboardType= UIKeyboardTypeNumberPad;
    petDateofBirth.keyboardType =
        UIKeyboardTypeNumbersAndPunctuation;
    petType.keyboardType= UIKeyboardTypeAlphabet;
    petBreed.keyboardType= UIKeyboardTypeAlphabet;
    petGender.keyboardType = UIKeyboardTypeAlphabet;
    petWeight.keyboardType = UIKeyboardTypeDecimalPad;
    petChipID.keyboardType =
        UIKeyboardTypeNumbersAndPunctuation;
    vetAddress.keyboardType =
        UIKeyboardTypeAlphabet;
    ownerName.keyboardType = UIKeyboardTypeAlphabet;
    ownerAddress.keyboardType =
        UIKeyboardTypeNumbersAndPunctuation;
    ownerHomePhone.keyboardType = UIKeyboardTypeNumberPad;
    ownerMobilePhone.keyboardType =
        UIKeyboardTypeNumberPad;
}

```

In the preceding code snippet, we initialize our super class's inherited members, and then set the color of our navigation bar control to blue, to differentiate that we are editing an existing record.

2. Next, we check the value of the `currentPet` variable to determine if we are editing an existing record, and if so, we use a managed object called `CurrentPet` to grab each of the schema fields and the attribute values of the managed object. Finally, we use the `keyboardType` property of each entry field to set the relevant keyboard styles for each.



For more information on each of the different keyboard styles available, refer to the `UITextInputTraits` Protocol Reference Guide within the Apple Developer Documentation at the following URL: http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UITextInputTraits_Protocol/Reference/UITextInputTraits.html.

Implementing the `btnSavePet:` method

We are now ready to start implementing the method that will be responsible for saving our pet record information when the user presses the **Save** button.

Open the `PetDetailsViewController.m` implementation file, located within the **Veterinary** folder, and enter in the following code snippet:

```
- (IBAction)btnSavePet:(id) sender {

    // Set a pointer to our Pet database table schema
    if (!self.currentPet) {
        self.currentPet = (PetDetails *) [NSEntityDescription
            insertNewObjectForEntityForName:@"PetDetails"
            inManagedObjectContext:managedObjectContext];
    }

    // Convert the Pet Photo Image to an NSData format
    // to be stored in the SQLite Database.
    NSData *currentPetPhoto = [NSData
        dataWithData:UIImagePNGRepresentation(petPhoto.image)];

    // Set up our date Format when storing the date of birth
    NSDateFormatter *dateFormat =
        [[NSDateFormatter alloc] init];
    [dateFormat setDateFormat:@"yyyy-MM-dd"];

    // Initialise our Number Formatter for Decimal Values
    NSNumberFormatter *numberFormat =
        [[NSNumberFormatter alloc] init];
    [numberFormat
        setNumberStyle:NSNumberFormatterDecimalStyle];
    NSNumber *cPetAge = [numberFormat
        numberFromString:petAge.text];

    // Grab each of the form fields and assign to each
    // of their attributes
```

```

[self.currentPet setPetName:self.petName.text];
[self.currentPet setPetAge:cPetAge];

// Check to ensure that we have a value for the Date
// Of Birth and then set it to be in the correct format.
NSDate *date = [dateFormat
    dateFromString:petDateofBirth.text];
[self.currentPet setPetDOB:date];

[self.currentPet setPetType:self.petType.text];
[self.currentPet setPetBreed:self.petBreed.text];
[self.currentPet setPetGender:self.petGender.text];

NSNumber *cPetWeight = [numberFormat
    numberFromString:petWeight.text];
[self.currentPetset PetWeight:cPetWeight];
[self.currentPetset PetChipID:self.petChipID.text];
[self.currentPetset PetPhoto:currentPetPhoto];
[self.currentPetset VetAddress:self.vetAddress.text];

// Obtain the Owner Details
[self.currentPet setOwnerName:self.ownerName.text];
[self.currentPet setOwnerAddress:self.ownerAddress.text];
[self.currentPet
    setOwnerPhoneNo:self.ownerHomePhone.text];
[self.currentPet
    setOwnerMobileNo:self.ownerMobilePhone.text];

// Catch any errors during the saving of the
// Pet Details Record.
NSError *error;
if (![managedObjectContext save:&error])
{
    // Display Error message stating that the record
    // could not be saved.
    NSLog(@"There was a problem saving the pet details...");
}
// Close the Pet Details form and return back to
// our Pet Listing.
[self dismissViewControllerAnimated:YES completion:nil];
}

```


In the preceding code snippet, we create a managed object context that is then used to create a new managed object using the `PetDetails` entity description. The getters and setters for each of the schema fields of the managed object are then called to set each of the attributes values of the managed object, before the context is finally instructed to save the changes to the persistent store with a call to the context's `save` method.

We then add the new pet information details object to our existing list of pet records and then refresh the table view, using the `reloadData` method to show that the new item was added, and then close the **Add New Pet Details** screen.

Implementing the `btnCancel:` method

Next, we need to implement the **Cancel** button. This will be responsible for closing the screen, and returning you back to the **Veterinary Clinic Application** table view when pressed.

Open the `PetDetailsViewController.m` implementation file, located within the `VeterinaryClinic` folder, and enter in the following code snippet:

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

In the preceding code snippet, we use the `dismissViewControllerAnimated` method, which is only made available in iOS 5 and later. This method is used to close the current modal screen that was sent by our **Veterinary Clinic Application** table view screen.

Implementing the `btnAddPhoto:` method

Next, we need to implement the **Photo Album** button. This will be responsible for displaying the camera roll view within a popover when pressed.

Open the `PetDetailsViewController.m` implementation file, located within the `VeterinaryClinic` folder, and enter in the following code snippet:

```
// called when the user presses the Photo Library button
- (IBAction)btnAddPhoto:(id) sender {

    // Create image picker controller
    self.imagePicker= [[UIImagePickerController alloc] init];

    // Set source to the Photo Library
```

```

self.imagePicker.delegate = self;
self.imagePicker.sourceType =
    UIImagePickerControllerSourceTypePhotoLibrary;
Self.imagePicker.allowsEditing = NO;

self.popoverController = [[UIPopoverController alloc]
    initWithContentViewController:self.imagePicker];
self.popoverController.delegate = self;
[self.popoverController
    presentPopoverFromBarButtonItem:sender
    permittedArrowDirections:UIPopoverArrowDirectionUp
    animated:YES];
}

```

In the preceding code snippet, we declare an instance of `UIImagePickerController` and initialize the properties to only display the images from the photo album. Next, we declare a popover controller that will be passed through the image picker as the view. The view controller is then designated as the delegate for the popover object before the popover is displayed to the user. The sender object passed through to this method references the **Photo Library** button in the toolbar. This object is passed through the popover controller's `presentPopoverFromBarButtonItem:` method, so that the popover is positioned directly above, and pointing to the button when displayed.

Implementing the `btnCameraPhoto:` method

Next, we need to implement the **Camera Photo** button. This will be responsible for displaying the camera roll view within a popover when pressed.

Open the `PetDetailsViewController.m` implementation file, located within the `VeterinaryClinic` folder, and enter in the following code snippet:

```

// Display the iOS Device' Camera using the
// backview as the default.
- (IBAction)btnCameraPhoto:(id)sender {

    // Create image picker controller
    self.imagePicker = [[UIImagePickerController alloc] init];

    // Set source to the Camera
    self.imagePicker.delegate = self;
    self.imagePicker.sourceType =
        UIImagePickerControllerSourceTypeCamera;
    self.imagePicker.cameraDevice =
        UIImagePickerControllerCameraDeviceRear;
    [self presentViewController:imagePicker animated:YES
        completion:nil];
}

```

In the preceding code snippet, we start by creating a `UIImagePickerController` instance, and then make the delegate point to itself and set the media source as the camera. Next, we set the value of the `cameraDevice` property to use the rear camera. Finally, we display the camera interface, and the `UIImagePickerController` object is released.

```
#pragma mark - Image Picker Delegate Methods
// On cancel, only dismiss the picker controller
- (void)imagePickerControllerDidCancel:(UIImagePickerController
*)picker
{
    [imagePicker dismissModalViewControllerAnimated:YES];
}
```

In the preceding code snippet, we start by declaring a delegate object for our image picker controller, `imagePickerControllerDidCancel`. This delegate will be responsible for handling and taking care of closing the popover or camera session, without making an image selection, or taking a picture whenever the **Cancel** button has been pressed.

```
// Calls once the user has chosen an image
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info {
    UIImage *photoImage = [info
objectForKey:UIImagePickerControllerOriginalImage];
    petPhoto.image = photoImage;

    [self.imagePicker dismissViewControllerAnimated:YES
completion:nil];
}
```

In the preceding code snippet, the `didFinishPickingMediaWithInfo` method dismisses and releases the image picker popover and identifies the type of media passed from the image picker controller. If it is an image, it is then displayed to the `petPhoto.image` object property of the user interface. If this is a new image, it is saved to the camera roll. The `didFinishPickingMediaWithInfo` is called when the user has finished taking or selecting an image.

Implementing the Delete row method

Next, we need to implement the `Delete` method. This will be responsible for removing a pet detail record from the **Veterinary Clinic** table view.

Open the `PetsViewController.m` implementation file, located within the `VeterinaryClinic` folder, and modify the `tableView:(UITableView *)tableViewcommitEditingStyle:` method, by entering in the following highlighted code:

```

// Override to support editing the table view.
- (void)tableView:(UITableView *)tableViewcommitEditingStyle:(
    UITableViewCellEditingStyle)editingStyleforRowAtIndexPath:(
    NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        // Get the item to delete from our row
        PetDetails *itemToDelete = [self.petListArray
            objectAtIndex:indexPath.row];

        // Delete the item in Core Data
        [self.managedObjectContext
            deleteObject:itemToDelete];

        // Delete the item from our array
        [petListArray removeObjectAtIndex:indexPath.row];

        // Commit the deletion
        NSError *error;
        if (![self.managedObjectContext save:&error])
        {
            NSLog(@"There was a problem deleting the pet
                information %@", [error domain]);
        }
        // Delete the row from the data source
        [tableView deleteRowsAtIndexPaths:[NSArray
            arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
    }
}

```

In the preceding code snippet, we determine the type of action currently being performed within the table view, which is determined by the `UITableViewCellEditingStyle` class. We then compare against the `UITableViewCellEditingStyleDelete` constant variable, and if the condition is met, we remove the selected pet details at the selected row from our `Pet` array and database, then refresh the Table view data source. If any errors have been detected during the removal process, they logged out to the **Debug** window.

Finishing up

We just have a few more things to implement before we have a complete working application. We will need to implement a couple more methods that will handle the transition between our **Veterinary Clinic** and our **Add New Pet Details** screens when the **+** and **Edit** buttons are pressed.

1. Firstly, we need to import our `PetDetailsViewController.h` interface file at the top of our `PetsViewController.m` implementation file; so let's do this now.

Open the `PetsViewController.m` implementation file, and enter in the following highlighted code snippet:

```
// PetsViewController.m
// VeterinaryClinic
// Created by Steven F. Daniel on 10/02/12.
// Copyright (c) 2012 GenieSoft Studios. All rights
// reserved.
```

```
#import "PetsViewController.h"
#import "PetDetailsViewController.h"
```

```
@implementation PetsViewController
```

```
@synthesize fetchedResultsController;
@synthesize managedObjectContext;
@synthesize petListArray;
```

2. Next, we need to handle the transition between the **Veterinary Clinic Application** screen and the Navigation controller, to determine when a transition has been made on a segue within the Storyboard.
3. Next, with the `PetsViewController.m` implementation file still open, enter in the following code snippet:

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender
{
    UINavigationController *navigationController =
        segue.destinationViewController;
    PetDetailsViewController
        *petDetailsViewController= [[navigationController
```

```

        viewController objectAtIndex:0];
    petDetailsViewController.managedObjectContext =
        self.managedObjectContext;

    if ([
        segue.identifier isEqualToString:@"EditPetDetails"])
    {
        // Get the row we selected to view
        NSInteger selectedIndex= [[self.tableView
            indexPathForSelectedRow]row];
        // Set the title of our form
        petDetailsViewController.title = @"Editing Pet
            Details";
        // Pass the picture object from the table
        // that we want to view.
        [petDetailsViewController setCurrentPet:[petListArray
            objectAtIndex:selectedIndex]];
    }
}

```

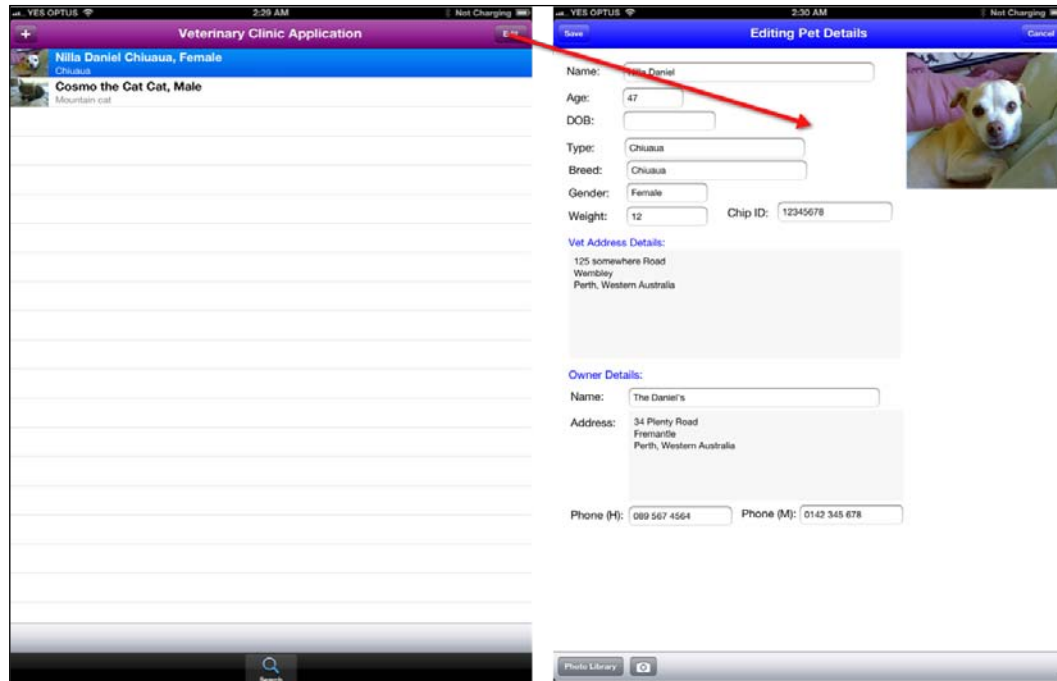
In the preceding code snippet, we use the `prepareForSegue:` method to determine whenever a transition to segue takes place. A check is required to be made on the identifier of the segue to determine if we are calling the **Edit Pet Details** screen, then pass the selected pet information to the `petDetailsViewController` class, and update the header for the form to show that we are currently in the **Edit** mode.

If the determination has been made that editing has not taken place, it will call and display a blank form.

4. Next, we set the Navigation controller of the segue to be the navigation controller of the destination screen, and then cycle through each of the view controller within the navigation controller properties to get the `PetDetailsViewController` instance, before finally setting the data source property of the form to be the currently active connection.

Congratulations, we have finally implemented the methods for our `VeterinaryClinic` application.

- Next, build and run the application by choosing **Product | Run** from the **Product** menu, or alternatively pressing *Command + R*. The following screenshot shows the application running on the iOS device with the first item in the list being selected.



From the preceding screenshot, we can see that when the item has been selected and the **Edit** button has been pressed, the details for the selected row are passed to the **Add New Pet Details** form, all fields are read from the database, and all relevant fields are then populated to the form.

Summary

In this chapter, we learned how to create a `VeterinaryClinic` application, using Core Data to separate our data model from the rest of the application using Model-View-Controller design. We visually designed our `PetDetails` entity, which contained the attributes representing each pet's name, age, date of birth, type of pet, breed, gender, and so on, and programmatically interacted with the data model using `NSManagedObject` and `NSFetchedResultsController` to fetch information from the data store and populate this within `UITableView`.

We also looked into segues in a bit more detail and how we can use these to easily utilize an existing previously created view, and pass information back-and-forth based on the current record that has been selected from the table view controller.

In the next chapter, we will look at how to create an application that will allow us to incorporate and interact with Facebook, directly within our application. We will look at how to register our iOS application with Facebook, and how to install the Facebook SDK.

8

Social Networking Application

When Mark Zuckerberg launched Facebook from his Harvard dormitory room on February 4, 2004, he never could have imagined how big his dream would have become, used by millions of people around the world. On May 24, 2007, Zuckerberg announced the Facebook Platform, a development platform for programmers to create social applications within Facebook.

When Facebook launched the development platform, numerous applications had been built, and already had millions of users playing them. The Social Networking application utilizes the Facebook collection of APIs that enable you to connect to Facebook and send application request notifications, so that you can add them to your list of friends. In this chapter, we will take a look at how to download the Facebook iOS SDK and register your iOS application, so that it can be used with Facebook.

We will then start by creating a simple application and look at how we can add the Facebook iOS SDK into our project, so that the user can sign in to their Facebook account in order to send notification requests as well as submit news feeds directly to their timeline home page. Finally, we will look at how to implement the **Single Sign-On (SSO)** feature of the Facebook iOS SDK that allows the user to sign into your application using their Facebook identity.

In this chapter we will:

- Learn how to download the Facebook iOS SDK
- Learn how to register your iOS application with Facebook
- Build a simple Social Networking application that interacts with Facebook
- Implement a method to use the SSO feature
- Implement a method to integrate with the various social channels

- Learn how to implement and use the Graph API
- Learn how to implement and handle errors within Facebook

We have an exciting project ahead of us, so let's get started.

Overview of the technologies

The Facebook application that we will be developing makes reference to the Facebook iOS SDK. This framework contains all of the method objects and APIs that are required to enable you to interact with Facebook and send notification requests, or simply post messages to the current person's timeline. We will be taking a look into one of the most fundamental features of the Facebook iOS SDK that is the SSO.

This lets your users sign in to your app using their Facebook identity. With the initial release of the SDK, the `authorize` method always opened an inline dialog box containing `UIWebView` in which the authorization UI was shown to the user, and required users to enter their credentials separately for each app they authorized.

In the updated version of the SDK, this has been changed, and no longer requires users having to re-enter their credentials for every application on the device they want to authorize.

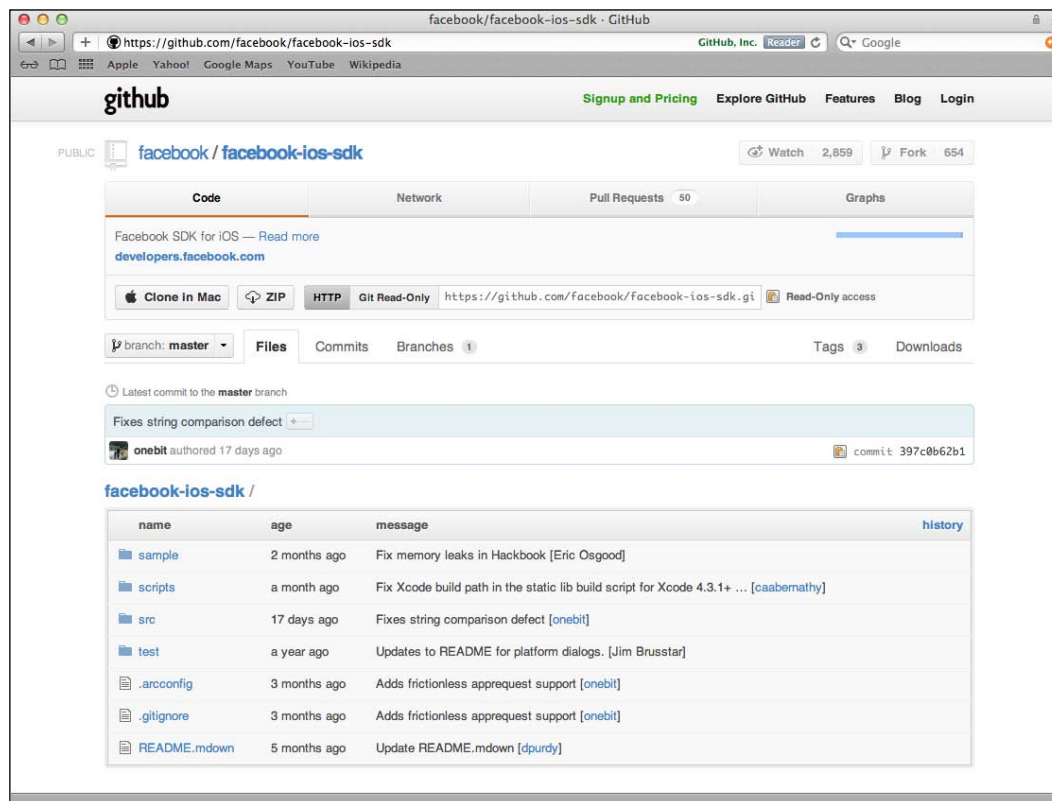
Using the Facebook iOS SDK allows you to do the following:

Facebook iOS SDK types	Description
Authentication and authorization	This prompts users to log in to Facebook and grant permissions to your application.
Make API calls	This allows you to fetch user profile data, as well as any information related to the user's friends using JSON API calls.
Display dialog	This allows you to interact with the user via a <code>UIWebView</code> view. This is extremely useful for enabling interactions with Facebook, without requiring upfront permissions.

Now that we have a reasonable understanding of what the Facebook iOS SDK encompasses and are comfortable with the different types of technologies that we will be dealing with, our next step is to proceed with the download and install the Facebook iOS SDK.

Downloading the Facebook iOS SDK

Before we can start building our Social Networking application, we need to first download the Facebook iOS SDK at <http://github.com/facebook/facebook-ios-sdk/>. This SDK provides you with all of the functionality required to make your application interact with Facebook. You have the option to download the file in the various formats: Clone in Mac, ZIP, and HTTP.

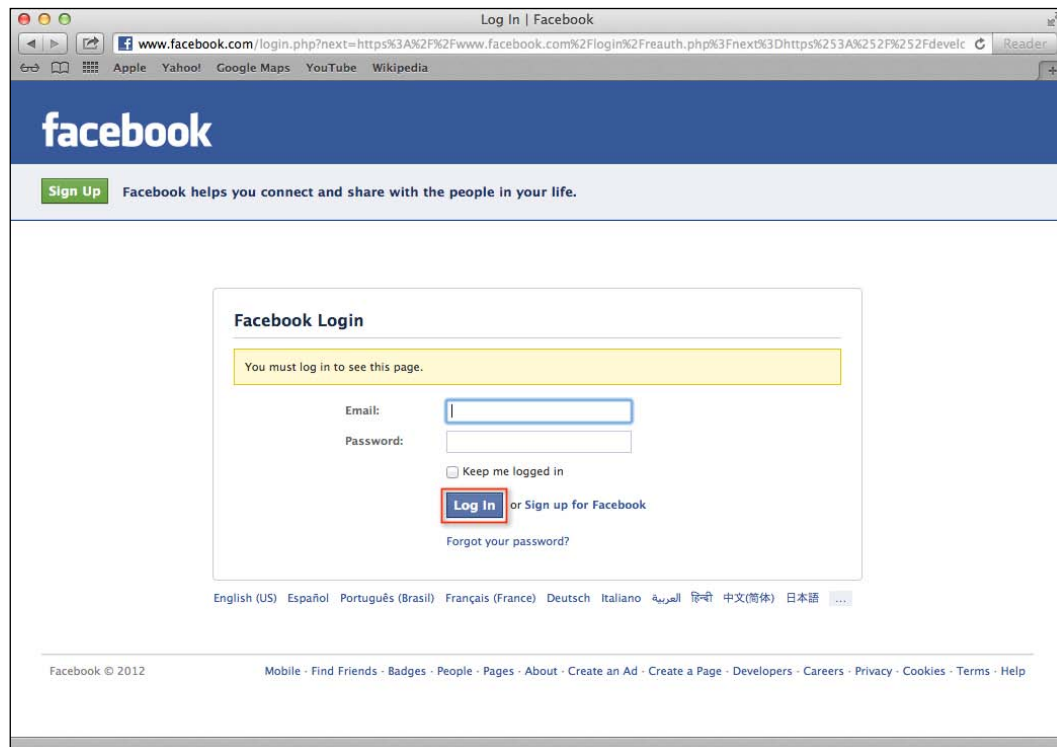


Once you have downloaded the Facebook iOS SDK, the next step is to register your iOS app with Facebook. This will allow your application to interact with Facebook, and allow the visitors of your application to post messages to your timeline or send friend requests.

Registering your iOS app with Facebook

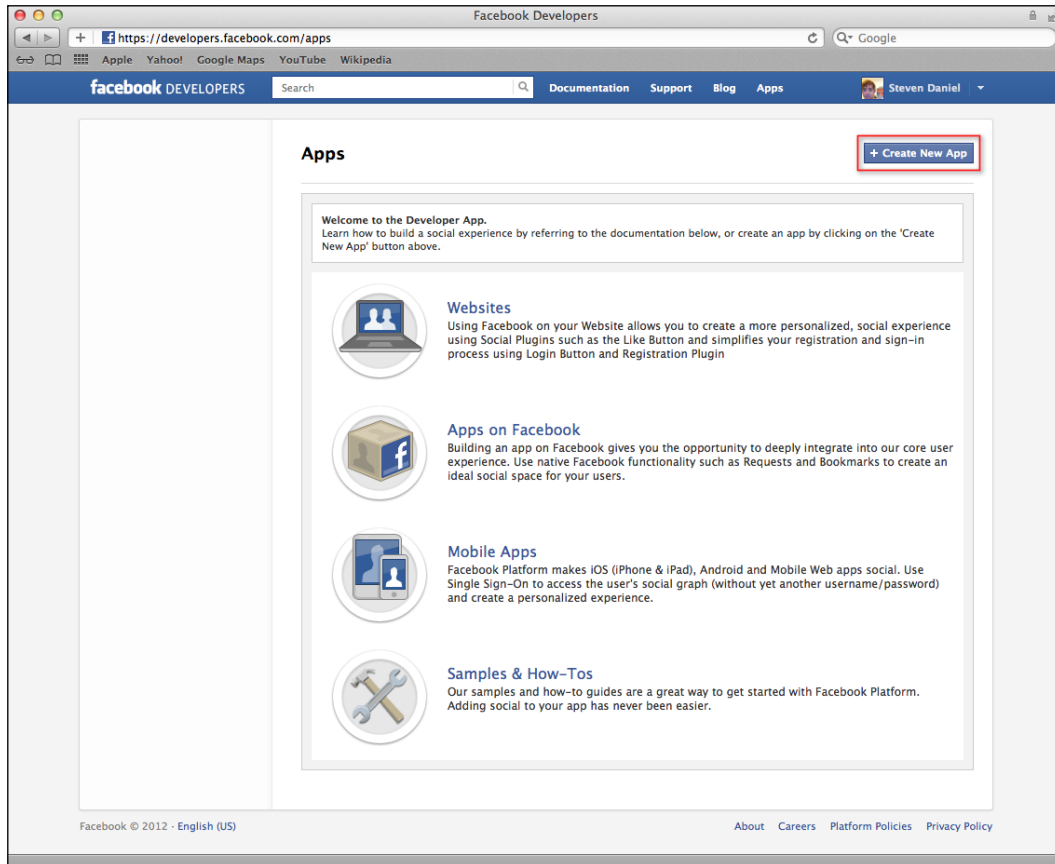
Before we can start integrating our application with the Facebook platform, we will need to register the application with Facebook's mobile website and provide some basic application information. To begin follow these steps:

1. Open your browser and enter in `http://developers.facebook.com/apps`.



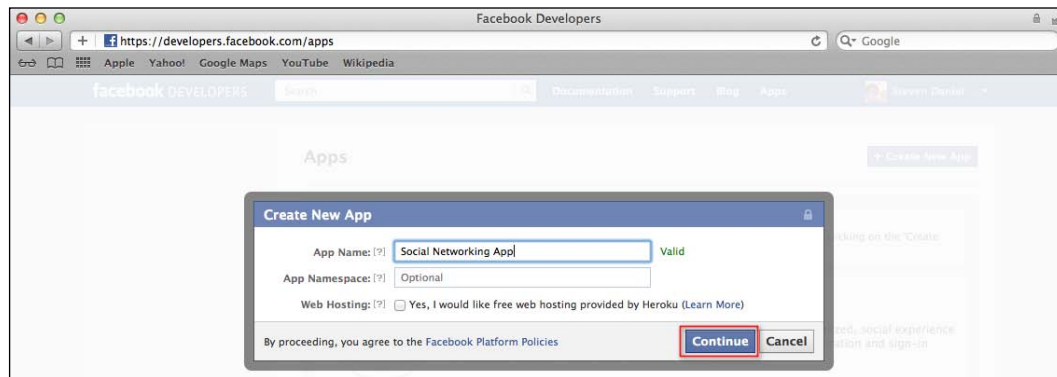
2. Next, enter in your Facebook account credentials.
3. Then, click on the **Log In** button.

- Next, click on the **+Create New App** button from the **Apps** page.

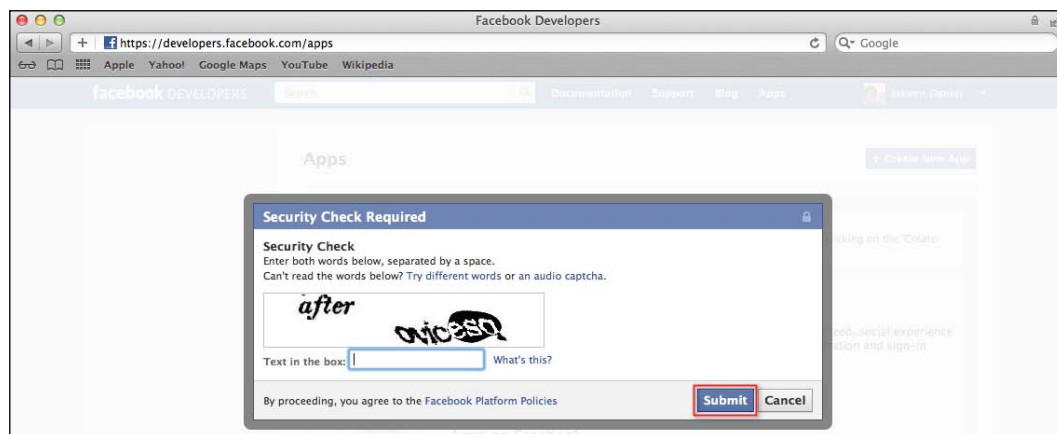


- Next, enter in Social Networking App for the **App Name** field.

- Click on the **Continue** button to proceed with the next step of the wizard. This will be used and displayed whenever you post or send a notification message to your friends.



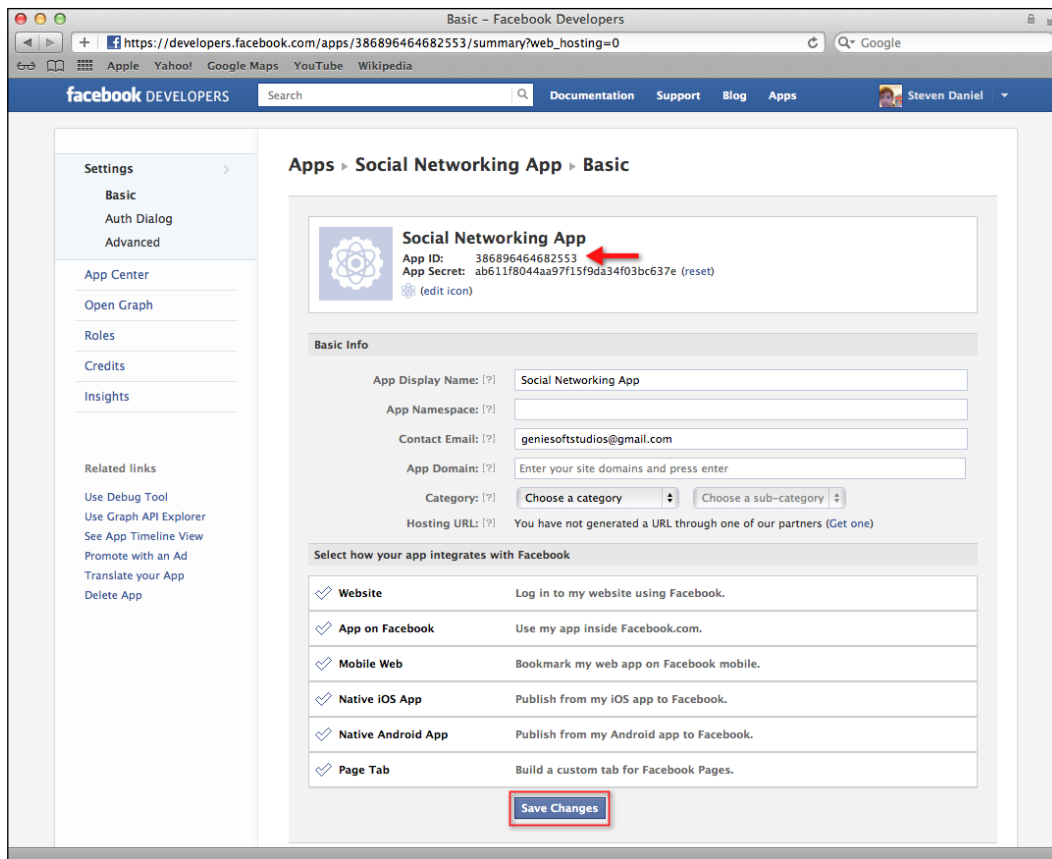
- Next, you will be prompted to enter in the **Security Check** words before you can proceed with the next step.



- Enter in the words displayed on your screen and click on the **Submit** button to continue. The words displayed will be different each time this screen is displayed.



If you enter in the words incorrectly, you may end up with your account being blocked. If this is the case, you will need to contact Facebook directly to have this unlocked.



9. This is the final screen that allows you to make any final changes before you commit your changes. Once you are satisfied with all changes, click on the **Save Changes** button.



The **App ID** is an important field that we will be using in our iOS application to communicate with Facebook, and is highlighted by an arrow in the preceding screenshot.

Building the Social Networking application

The ability to communicate with friends and family through phone or e-mails is a way of us staying in touch to let them know what we have been up to. This can be in many forms, such as through e-mails, phone, or more commonly by using social networking sites, such as Facebook and Google+.

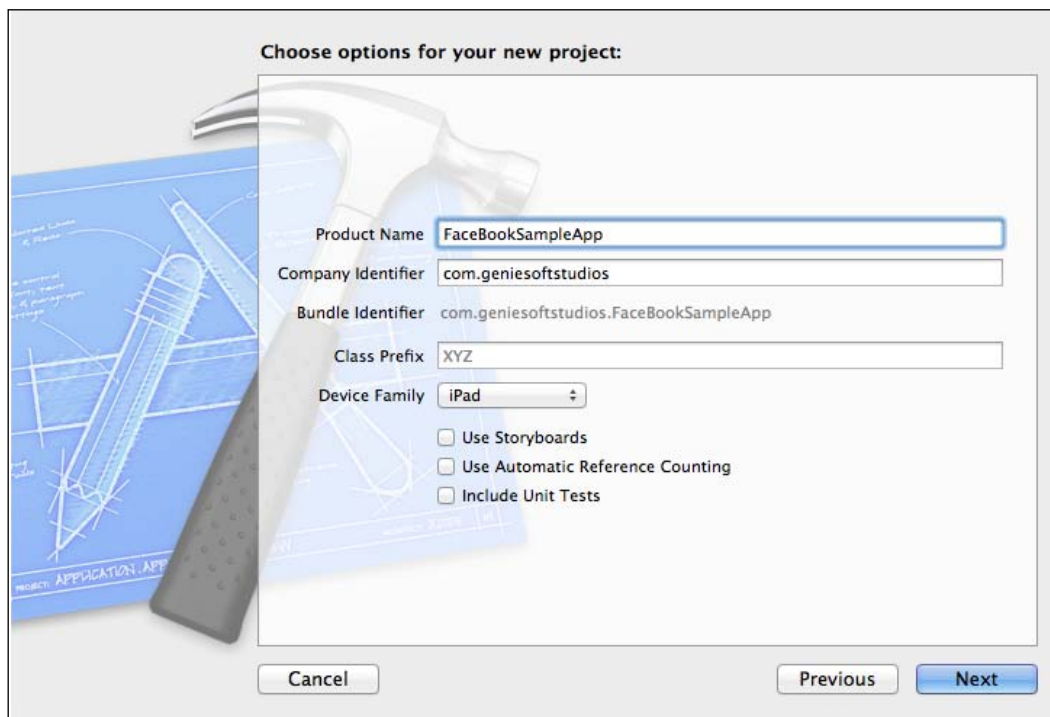
In this section, we will take a look at how to create an application that will do just that, so it can run on an iOS device, enabling us to send notification requests and post messages to our wall, to let any visitors know what we have been up to. We will be using the Facebook iOS SDK and make use of the methods and protocols to communicate with the Facebook development platform.

Before we can proceed, we first need to create our `FacebookSampleApp` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 2, Task Priorities – Building a TaskPriorities iOS App*, under the section named *Building the TaskPriorities app*.

It is very simple to create this in Xcode. Just follow the steps listed here.

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.
3. Select the **Single View Application** template from the list of available templates.
4. Click on the **Next** button to proceed with the next step in the wizard.
5. Next, enter in `FaceBookSampleApp` as the name for your project.
6. Select **iPad** from under the **Device Family** drop-down list.
7. Ensure that the **Use Storyboards** checkbox has not been selected.
8. Ensure that **Use Automatic Reference Counting** checkbox has not been selected.
9. Ensure that the **Include Unit Tests** checkbox has not been selected.

10. Click on the **Next** button to proceed with the next step in the wizard.



11. Specify the location where you would like to save your project.
12. Then, click on the **Create** button to continue and display the Xcode workspace environment.

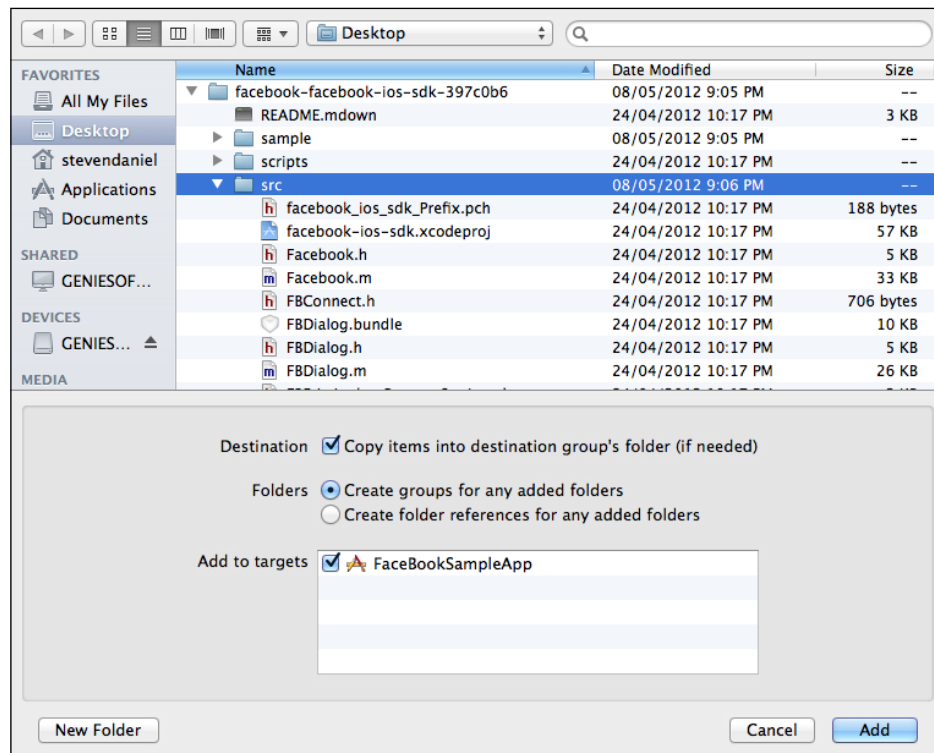
Now that we have downloaded our Facebook iOS SDK, registered our App ID with Facebook, and created our `FaceBookSampleApp` project, we need to start building our user interface that will be responsible for allowing us to communicate with Facebook and post messages to the current user's timeline.

Adding the Facebook iOS SDK to our project

Our next step is to include the Facebook iOS SDK as part of our Facebook Sample App project to enable us to communicate with the Facebook platform. To add the Facebook iOS SDK, select **Project Navigator Group**, and then follow these simple steps as outlined below:

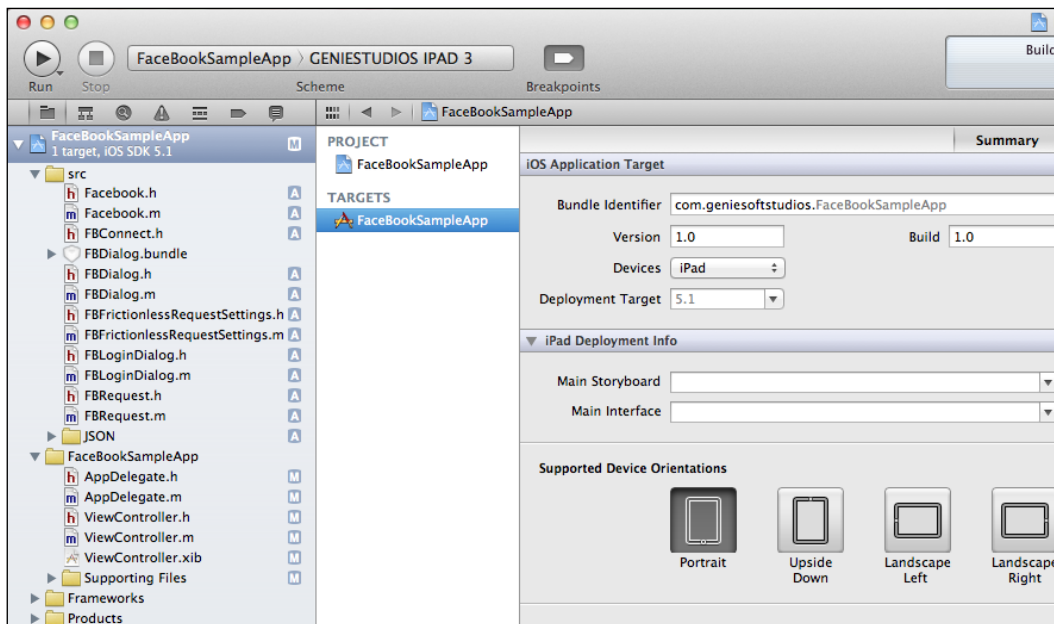
1. From the **Project Navigator** window, select the `FacebookSampleApp` folder.

2. Choose **Add Files to "FacebookSampleApp" ...** or press *Option + Command + A*.



3. Next, select the `src` folder from the `facebook-facebook-ios-sdk-397c0b6` folder.
4. Ensure that you have selected the **Copy items into destination group's folder (if needed)** checkbox.
5. Click on the **Add** button to proceed with importing the project source files into the `FacebookSampleApp` project.

- Once the Facebook iOS SDK has been imported into your project, your solution should contain the following files:



- Exclude `facebook_ios_sdk_prefix.pch` and `facebook-ios-sdk.xcodeproj` from the `src` folder.

Now that we have successfully added the Facebook SDK into our `FaceBookSampleApp` application, our next task is to start building our application.

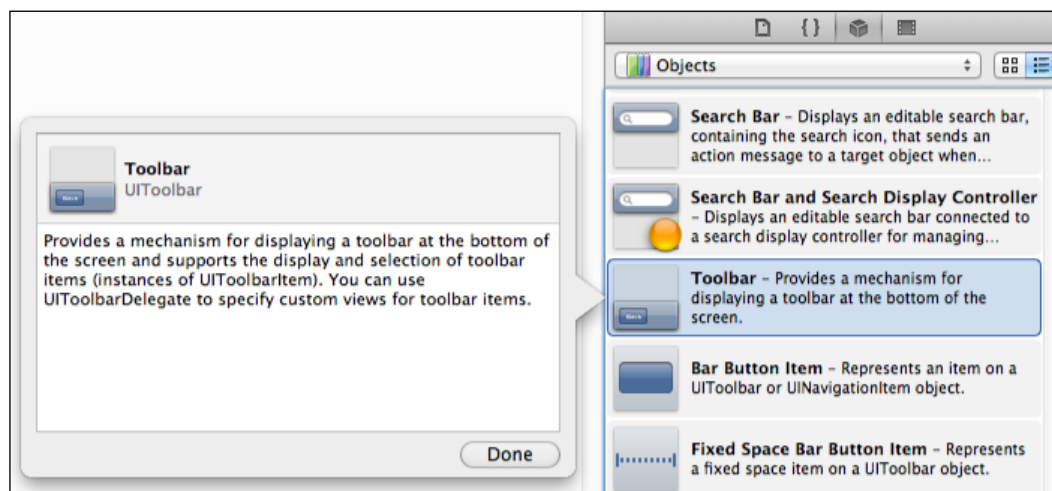


For more information on the Facebook iOS SDK, refer to the Facebook Developer Documentation at <http://developers.facebook.com/docs/guides/mobile/>.

Creating the main application screen

Our next step is to build the user interface for our Facebook Sample application. This screen will be very simple and will consist of just a View controller and a toolbar.

1. Select the `ViewController.xib` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UIToolbar) **Toolbar** control, and add this to the top of our view.



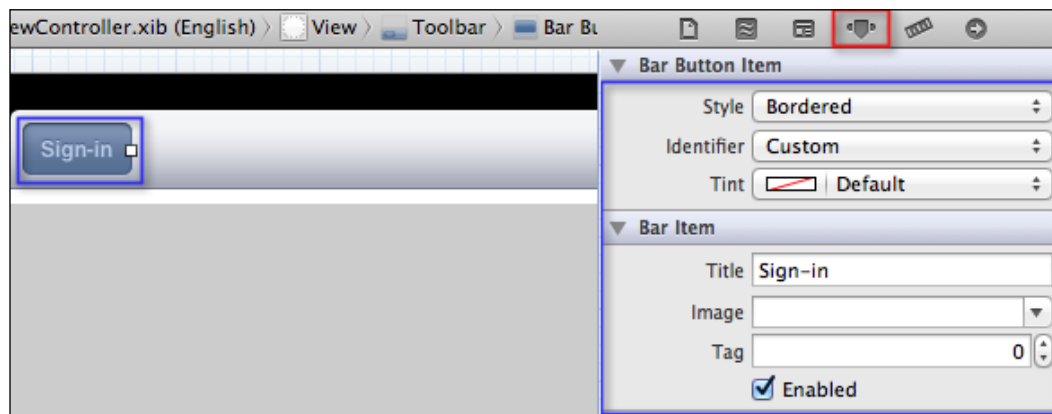
Now that we have added our `UIToolbar` toolbar control to our view controller, our next step is to start adding the **Sign-in**, **Sign-Out**, and **Action** buttons. So let's proceed with the next section.

Adding the Sign-in button

Our next step is to modify the button within our previously added toolbar; this button will be responsible for checking for a valid Facebook session and authorizing the necessary permissions to Facebook. This can be achieved by following these simple steps:

1. Select the `ViewController.xib` file from **Project Navigator**.
2. Next, select the **Item** button located within our toolbar that we previously added.
3. From the **Attributes Inspector** section, change the Identifier property to **Custom**.
4. Change the **Style** property to **Bordered**.

- Then, change the **Title** property to **Sign-in**.



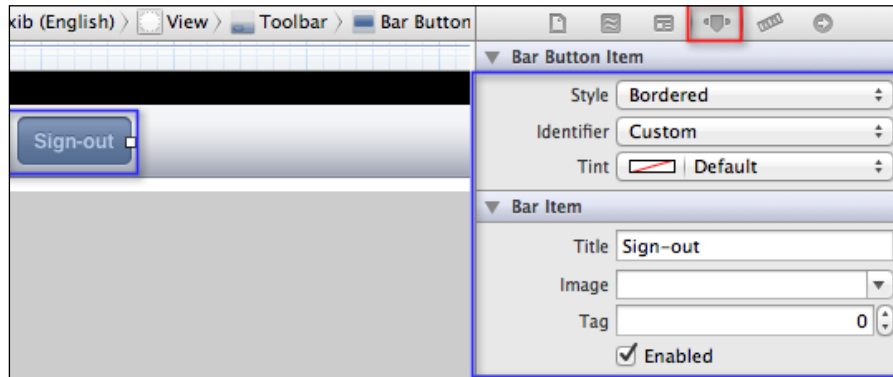
Now that we have added our **Sign-in** button to our **View** controller, our next step is to add the **Sign-out** button that will be responsible for signing out of Facebook and releasing the session object when the button is clicked. So let's proceed with the next section.

Adding the Sign-out button

Now that we have added our button to sign in to Facebook and instantiate our session object, our next step is to add another button that will be responsible for allowing the user to sign out of Facebook when this has been clicked. This can be achieved by following these simple steps:

- Select the `ViewController.xib` file from **Project Navigator**.
- From **Object Library**, select-and-drag a **Bar Button Item** (`UIBarButtonItem`) control after the **Sign-in** button, located within `UIToolBar`.
- From the **Attributes Inspector** section, change the **Identifier** property to **Custom**.
- Change the **Style** property to **Bordered**.

- Then, change the **Title** property to `Sign-out`.

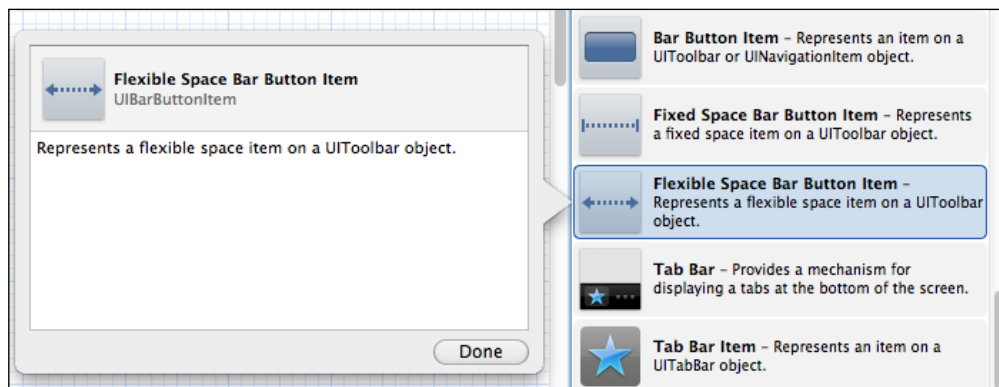


Now that we have added our **Sign-out** button to our `FaceBookApp` View controller, our next step is to add a **Flexible Space Bar Button Item** (`UIBarButtonItem`) control that will be used to fill in the space between the **Sign-out** and the **Action** buttons, which will be responsible for allowing us to send notifications, post messages to the current user's timeline, as well as obtaining specific information about the current logged on user, when this has been clicked.

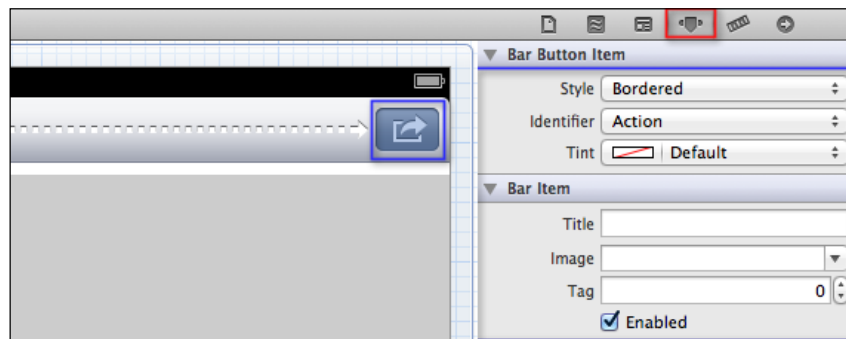
Adding the Action button

Now that we have added our button to sign into Facebook and instantiate our session object, our next step is to add another button that will be responsible for allowing the user to perform a variety of tasks to Facebook when this has been clicked. This can be achieved by following these simple steps:

- From **Object Library**, select-and-drag a **Flexible Space Bar Button Item** (`UIBarButtonItem`) control after the **Sign-out** button within `UIToolbar`.



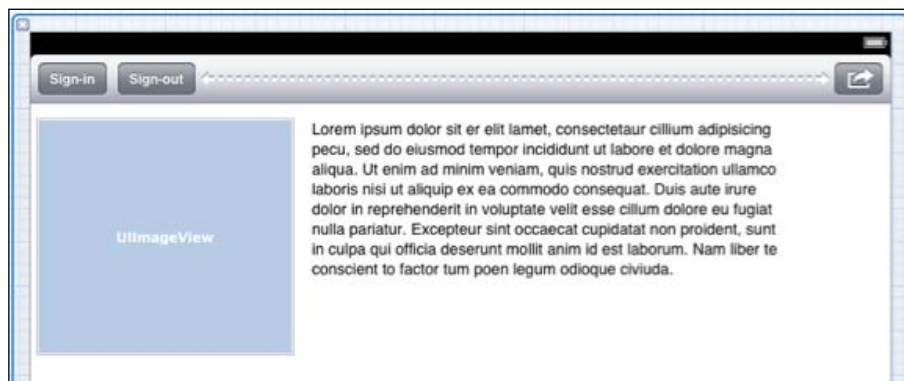
2. Next, from **Object Library**, select-and-drag a **Bar Button Item** (UIBarButtonItem) control after the **Flexible Space Bar Item** control, within UIToolBar.
3. From the **Attributes Inspector** section, change the **Identifier** property to **Action**.
4. Then, change the **Style** property to **Bordered**.



Our next step is to start adding some controls that will be used to display the current user's profile picture as well as some user information.

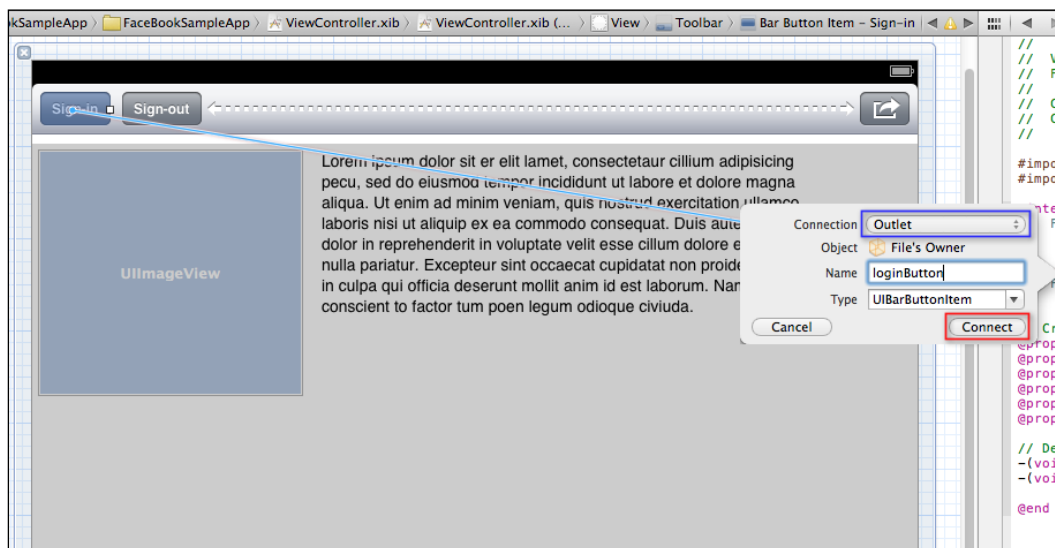
1. Next, from **Object Library**, select-and-drag a **Image View** (UIImageView) control, and place it underneath the toolbar. Resize the control accordingly, so that it has enough space to fit the profile picture.
2. Then, from **Object Library**, select-and-drag **TextView Control** (UITextView), place it to the right of the profile picture, and resize the control accordingly so that it has enough space to display the information.

If you have followed the steps correctly, the completed **View Controller Details** screen should look similar to the following screenshot. Feel free to adjust yours accordingly.



Our next step is to create the outlets for the **Sign-in**, **Sign-out**, and **Action** buttons, as well as our **UIImageView**, and **UITextView** form fields. Creating these will allow us to access these controls from within our code and make modifications to the control properties. To create an **Outlet**, follow these simple steps:

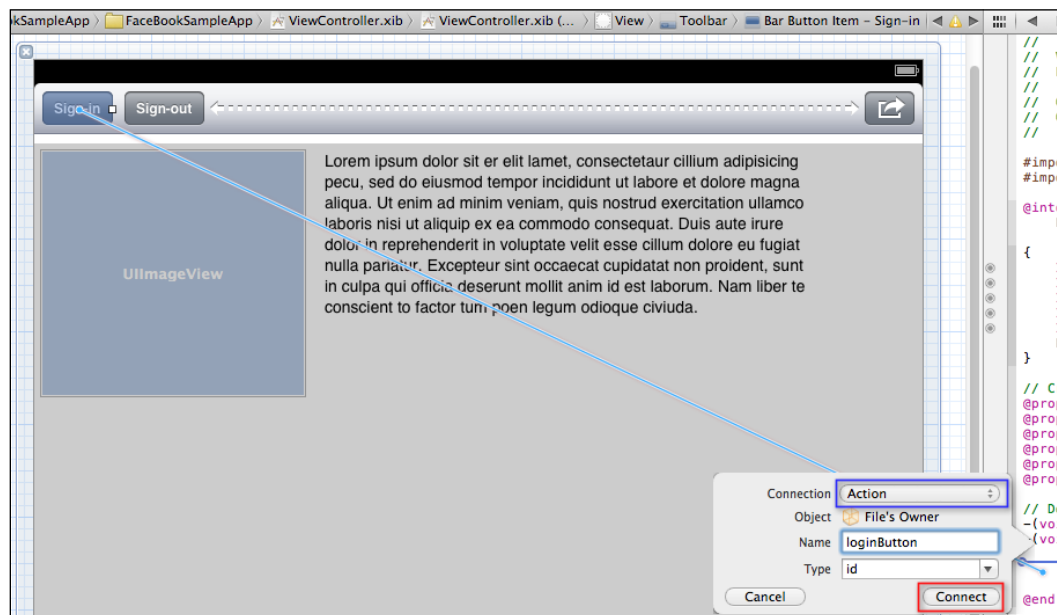
1. Open **Assistant Editor** by choosing **Navigate | Open In Assistant Editor** or press *Option + Command + ,*.
2. Ensure that the **ViewController.h** interface file is displayed to the left of **ViewController.xib**.
3. Select the **Sign-in** (**UIBarButtonItem**) control, then hold down the *Control* key and drag it into the **ViewController.h** interface file.
4. Choose **Outlet** from the **Connection** drop-down list for the connection to be created.
5. Enter in **loginButton** for the name of the **Outlet** to be created.



6. Repeat steps 3 to 5 to create the **IBOutlet**s for the **Sign-out**, **Action**, **UIImageView**, and **UITextView** controls, while providing the following namings for each as follows: **logoutButton**, **postMessage**, **imgPhoto**, and **userInfoDetails**.

Now that we have created the instance variable Outlets for our controls, we need to create the associated Actions for those Outlets events. Creating these actions allows an event to be fired when the button has been pressed. To create an Action, follow these simple steps:

1. With the `ViewController.h` interface file still displayed to the left of the `ViewController.xib` view controller, select the **Sign-in** (UIBarButtonItem) control, then hold down the *Control* key and drag it into the `ViewController.h` interface file.
2. Choose **Action** from the **Connection** drop-down list for the connection to be created.
3. Enter in `loginButton` for the name of the Action to be created.



4. Repeat steps 1 to 3 to create the IBActions for the **Sign-out** and **Action** controls, while providing the following naming for each: `logoutButton` and `postMessage`, respectively.

Now that we have successfully connected up each of our controls and created the required outlets and associated action methods, we can start taking a look at building the Facebook functionality into our Facebook sample application, so that it has the ability to send notifications, post messages to the current user's timeline, as well as retrieve profile information using JSON web method calls.

Building the Facebook app functionality

Well done! You have finally made it this far; we have successfully finished building the user interface for our Facebook Social Networking application. Our next step is to start implementing the methods that will be used for our **Sign-in**, **Sign-out**, and **Action** buttons.

Implementing SSO within your app

One of the most brilliant features of the Facebook iOS SDK is the SSO feature.

This feature lets users to sign in to your application using their personal Facebook login credentials. If a user is already signed into the Facebook iOS application on their device, they won't need to provide this again.

The process of using SSO works by redirecting the user to the Facebook iOS application on their device, and presenting them with an authentication dialog box, showing only those permissions that your application has been configured to use. Once the user has allowed those permissions requested by your iOS app, they will be redirected back to your application with the appropriate access token.

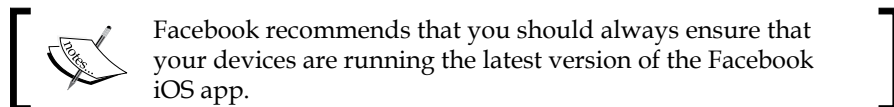


When using the Facebook SSO process, certain things can behave slightly differently depending on what version of the Facebook iOS app has been installed on the user's iOS device.

The following explains what happens when the Facebook SSO process is run under these certain conditions.

- If the iOS application is running with a version of iOS that supports multitasking and running version 3.2.3 or greater of the Facebook iOS app, the Facebook SDK will attempt to open the authorization dialog box within the Facebook app. After the user grants or declines authorization, the user is redirected back to the calling application, passing back with it an authorization token, expiration, and any other parameters the Facebook OAuth authentication server may happen to return.
- If the iOS device is running with a version of iOS that supports multitasking, and isn't running version 3.2.3 or greater of the Facebook iOS app, the Facebook SDK will open the authorization dialog box within Safari. After the user grants or declines authorization, Safari will redirect the user back to the calling application. This process is similar to the Facebook app authorization, and allows for multiple apps to share the same Facebook user `access_token` through the Safari cookie.

- If the iOS application is running a version of iOS that does not support multitasking, then the SDK will use the old mechanism of popping up an inline `UIWebView` web view control, prompting the user to log in and grant access.



Adding SSO into your iOS application is very easy, and we will be taking a look at how this can be achieved in the next section.

Implementing the Application Delegate class

We are now ready to start adding additional content to our `AppDelegate` class, so that it can handle and communicate easily with Facebook through the use of the SSO process. We will first need to import some important header files, as well as declare some object variables that we will be using within this delegate class, and to be called from other class modules. We will also need to extend our class, so that we can use the Facebook Session objects and the Facebook Dialog objects.

1. Open the `AppDelegate.h` interface file, located within the `FacebookSampleApp` folder, and enter in the following highlighted code:

```
// AppDelegate.h
// FaceBookSampleApp
// Created by Steven F. Daniel on 10/05/12.
// Copyright (c) 2012 GenieSoft Studios. All rights
//reserved.

#import <UIKit/UIKit.h>
#import "FBConnect.h"
#import "FBRequest.h"

@class ViewController;

@interface AppDelegate : NSObject<UIApplicationDelegate,
    FBSessionDelegate, FBDialogDelegate>
{
    Facebook *facebook;
}

// Create the required class Setters and Getters
@property (strong, nonatomic) UIWindow *window;
```

```
@property (strong, nonatomic) ViewController
    *viewController;
@property (nonatomic, retain) Facebook *facebook;

@end
```

In the preceding code snippet, we import the interface file header information for our `FBConnect.h` and `FBRequest.h` interface files, so that we can access their class methods. We then extended our class to include each of the following protocols: `FBSessionDelegate` and `FBDialogDelegate`, as well as its methods. We then declared an instance variable called `facebook` that will enable us to access the Facebook class methods. In our final step, we add a property instance of the Facebook class to create the class getters and setters.

2. Next, open the `AppDelegate.m` implementation file, located within the `FacebookSampleApp` folder, and enter in the following highlighted code sections:

```
// AppDelegate.m
// FaceBookSampleApp
// Created by Steven F. Daniel on 10/05/12.
// Copyright (c) 2012 GenieSoft Studios. All rights
// reserved.

#import "AppDelegate.h"
#import "ViewController.h"

@implementation AppDelegate
```

```
@synthesize window = _window;
@synthesize viewController = _viewController;
@synthesize facebook;
```

In the preceding code snippet, we need to synthesize our facebook variable that we defined within the `AppDelegate.h` interface file. This is so that we can make our implementation file aware of this, so that we can access the object properties and methods.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary
    *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]] autorelease];

    // Override point for customization after
```

```

// application launch.
self.viewController = [[[ViewController alloc]
initWithNibName:@"ViewController" bundle:nil]
autorelease];

self.window.rootViewController = self.viewController;

// Do any additional setup after loading the view,
// typically from a nib.
self.facebook= [[Facebook alloc]
initWithAppId:@"YOUR_APPID_HERE"
andDelegate:self];


// Check and retrieve authorization information
NSUserDefaults *defaults = [NSUserDefaults
standardUserDefaults];
if ([defaults objectForKey:@"FBAccessTokenKey"] &&
[defaultsobjectForKey:@"FBExpirationDateKey"]) {
self.facebook.accessToken = [defaults
objectForKey:@"FBAccessTokenKey"];
self.facebook.expirationDate = [defaults
objectForKey:@"FBExpirationDateKey"];
}

// Check to ensure that we have a valid
// session object
if (![self.facebook isValidSession]) {
[self.facebook authorize:nil];
}

[self.window makeKeyAndVisible];
return YES;
}

```

In the preceding code snippet, we need to initialize our Facebook object to invoke the SSO by passing in the application AppID that we created when we registered our iOS mobile app, as well as the Graph API and Platform Dialogs from within our app. Once the object has been instantiated, we need to check for any previously saved access token information and then use this saved information to set up a valid session, by assigning the saved information to the Facebook access token and expiration date properties.

 To ensure your application works with your own AppID, you will need to replace the YOUR_APPID_HERE string after the initWith AppId with your own created Facebook App ID.

This is to ensure that your app does not redirect to the Facebook application and invoke the authorization dialog box, if the application already has a valid `access_token`. Next, we check for a valid session and if it is not valid, we call the `authorize` method which will log the user in and prompt the user to authorize the application.

```
- (BOOL) application: (UIApplication *) application
    handleOpenURL: (NSURL *) url
{
    return [self.facebook handleOpenURL:url];
}

- (BOOL) application: (UIApplication *) application
    openURL: (NSURL *) url sourceApplication: (NSString *) sourceApplication
    annotation: (id) annotation
{
    return [self.facebook handleOpenURL:url];
}
```

In the preceding code snippet, we need to declare two methods that will be called by the iOS when the Facebook application redirects to the app during the SSO process. These methods provide the app with the user's credentials. You will notice that we have declared two different methods; this is to handle different versions of the iOS app. The `handleOpenURL` method is for versions prior to 4.2, and the `openURL` one is for versions 4.2 and greater.

```
- (void) fbDidLogin
{
    // Check and retrieve authorization information
    NSUserDefaults *defaults = [NSUserDefaults
        standardUserDefaults];

    [defaults setObject:[self.facebook accessToken]
        forKey:@"FBAccessTokenKey"];
    [defaults setObject:[self.facebook expirationDate]
        forKey:@"FBExpirationDateKey"];

    [defaults synchronize];
}
```

In the preceding code snippet, we implement the Facebook `fbDidLogin` method of `FBSessionDelegate`. After the SSO process has successfully signed in and the Facebook app redirects back to the calling application, we save the user's credentials using the `FBAccessTokenKey` and `FBExpirationDateKey` keys, and then save these into the user preferences `NSUserDefaults`:

```
- (void) fbDidLogout
{
    // Remove saved authorization information
    // if it exists
    NSUserDefaults *defaults = [NSUserDefaults
                               standardUserDefaults];

    if ([defaults objectForKey:@"FBAccessTokenKey"]) {
        [defaults
         removeObjectForKey:@"FBAccessTokenKey"];
        [defaults
         removeObjectForKey:@"FBExpirationDateKey"];
        [defaults synchronize];
    }

    UIAlertView *alertView = [[UIAlertView alloc]
                              initWithTitle:@"FaceBookSampleApp"
                              message:@"Your session has logged out."
                              delegate:nil
                              cancelButtonTitle:@"OK"
                              otherButtonTitles:nil,
                              nil];

    [alertView show];
    [alertView release];
}
```

In the preceding code snippet, we implement the Facebook `fbDidLogout` method of `FBSessionDelegate`. After the SSO process successfully signs out of the iOS app, the callback method gets called. Next, we need to check to see if we have a successful access token key prior to removing the stored user's credentials, using the `FBAccessTokenKey` and `FBExpirationDateKey` keys. Next, we remove those details from the user preferences using the `NSUserDefaults` object. Finally, we create an instance of the `UIAlertView` dialog box to notify the user that a successful logout has happened.

```
// Called when the session has expired.
- (void) fbSessionInvalidated {
    UIAlertView *alertView = [[UIAlertView alloc]
                              initWithTitle:@"FaceBookSampleApp"
```



```
        message:@"Your session has expired."
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil,
        nil];

        [alertView show];
        [alertView release];
        [self fbDidLogout];
    }
```

In the preceding code snippet, we implement the Facebook `fbSessionInvalidated` method of `FBSessionDelegate`. When a request is made to post a new message to the current user's timeline or to send notifications to your friends, the `fbSessionInvalidated` method is called to ensure that a valid session exists. This uses the `session` object, created by the SSO process when your application signed in. If the session state has expired, we declare an instance of the `UIAlertView` class to display a message to the user, before finally making a call to the `fbDidLogout` method to ensure that all of the required access tokens are removed cleanly.

Implementing the View Controller class

We are now ready to start adding additional content to our View Controller class. We will need to import some important header files, as well as declare some object variables that we will be using throughout our application. We will also need to extend our class, so that we can conform with the `ActionSheet` and `Facebook Request Dialog Delegate` protocols.

1. Open the `ViewController.h` interface file, located within the `FacebookSampleApp` folder, and enter in the following highlighted code sections:

```
// ViewController.h
// FaceBookSampleApp
// Created by Steven F. Daniel on 10/05/12.
// Copyright (c) 2012 GenieSoft Studios. All rights
// reserved.

#import <UIKit/UIKit.h>
#import "AppDelegate.h"

@interface ViewController :
    UIViewController<UIActionSheetDelegate,
    FBRequestDelegate>
{
```

```

    AppDelegate *mainDelegate;
    IBOutlet UIBarButtonItem *loginButton;
    IBOutlet UIBarButtonItem *logoutButton;
    IBOutlet UIBarButtonItem *postMessage;
    IBOutlet UIImageView *imgPhoto;
    IBOutlet UITextView *userInfoDetails;
    Facebook *facebook;
}

// Create the required class Setters and Getters
@property (nonatomic, strong) UIBarButtonItem
    *loginButton;
@property (nonatomic, strong) UIBarButtonItem
    *logoutButton;
@property (nonatomic, strong) UIBarButtonItem
    *postMessage;
@property (nonatomic, strong) UIImageView *imgPhoto;
@property (nonatomic, strong) UITextView
    *userInfoDetails;
@property (nonatomic, retain) Facebook
    *facebook;
@property (nonatomic, retain) AppDelegate
    *mainDelegate;

// Declare our instancemethods
- (void) SendNotificationRequest;
- (void) PostMessageToWall;

@end

```

In the preceding code snippet, we import the interface file header information for our AppDelegate.h interface file, so that we can access their class methods. We extend our class, so that we can include each of the following class protocols and methods for UIActionSheetDelegate and FBRequestDelegate. Next, we declare a new Facebook object, which will be used by our View Controller to access the Facebook properties. Finally, we declare an AppDelegate object variable to connect to the AppDelegate class and use the already instantiated Facebook object.

2. Open the `ViewController.m` implementation file, located within the `FacebookSampleApp` folder, and modify the `viewDidLoad` method, as shown in the following code snippet:

```
// ViewController.m
// FaceBookSampleApp
// Created by Steven F. Daniel on 10/05/12.
// Copyright (c) 2012 GenieSoft Studios. All rights
// reserved.
//

#import "ViewController.h"
#import "FBRequest.h"

@interface ViewController ()

@end

@implementation ViewController

@synthesize loginButton, logoutButton, postMessage, imgPhoto,
userInfoDetails;
@synthesize facebook, mainDelegate;

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Set up our delegate object
    self.mainDelegate= (AppDelegate *)[[UIApplication
        sharedApplication]delegate];
    self.facebook= mainDelegate.facebook;

    // Initialize our form fields
    userInfoDetails.text = @"";
}
```

In the preceding code snippet, we import the interface file header information for our `FBRequest.h` interface file, so that we can access the request dialog class methods. Next, we declare an application delegate object `mainDelegate` that points to the properties and class methods within our delegate class. Next, we initialize our `facebook` object within our view controller to use the same object instance as the one which we instantiated within our delegate class. Finally, we initialize our user info details `UITextView` control.

Adding the LogOut functionality to your app

When developing applications for the iOS platform, it is always better to provide an easy way for your user to log out cleanly from your application. This helps ensure that your application properly clears out or releases any objects and clear application state objects.

When working with the Facebook integration within your application, you simply make a call to the `logout` method of the Facebook class. This method clears the application state, and makes a server request to invalidate the current session `access_token`.

Open the `ViewController.m` implementation file, located within the `FacebookSampleApp` folder, and modify the `logoutFacebook` method, as shown in the following code snippet:

```
// Handle when the logout button is pressed.
- (IBAction)logoutFacebook:(id)sender {
    [self.facebook logout:self.mainDelegate];
}
```

In the preceding code snippet, we call the `logout` method of the Facebook class, and then pass in our `mainDelegate` object. When this method is called, it will call the `fbDidLogout` method of the `FBSessionDelegate` within our `AppDelegate` class to handle any post-logout actions and releasing of objects as well as for notifying the user that a successful logout has taken place.



When making a call to the `logout` method, your application's permissions will not be revoked; it will simply clear the value of your application's `access_token`.

If a user who has previously logged out of your application decides to run it again, they will simply see a notification that they are logging back into your application, not a notification requesting for permissions.

For more information on the Facebook protocol methods, refer to the following URL: <http://developers.facebook.com/docs/reference/iossdk/#protocols>.

Requesting additional permissions

When using Facebook integration within your application, you can specify additional permissions to be used by your application. When you launch your application without specifying additional permissions, your application uses the default permissions; by that I mean your application gets the ability to read only the user's basic information and this includes certain properties of the `User` object, such as id, name, picture, gender, and their locale.

If you want to read additional data or publish data back to Facebook, you will need to request these additional permissions. These additional permissions fall into the following sections:

Requested permission	Description
Basic information (no permissions)	When a user authorizes your application and you don't specify additional permissions, your application will only have access to the user's basic information. This includes certain properties, such as their id, name, gender, locale, and their profile picture.
User and friend permissions	As a part of the authorization process, you can also request for additional access to your user's profile. You can access information, such as their birthday, activities, check-ins, and education history. The user must, however, authorize this at startup in order to continue and authorize your application.
Extended permissions	If you are using the Enhanced Authorization Dialog, the extended permissions will be presented to the user. These type of permissions allow you to read your user's friend lists, read the user's mail inbox, access your user's friend requests, and create and modify events on the user's behalf.
Open graph permissions	These types of permissions allow your application to publish actions to the Open Graph API and enables it to retrieve any actions that have been published by any other application.
Page permissions	These types of permissions allow you to retrieve <code>access_tokens</code> for pages and applications that the user administrates, and is only compatible with the Graph API.

Now that we have covered a bit about permissions, let's take a look at how we can implement these additional permissions into our iOS application.

Open the AppDelegate.m implementation file, and apply the highlighted sections to the didFinishLaunchingWithOptions:(NSDictionary *)launchOptions method as shown in the following code snippet:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]] autorelease];

    // Override point for customization after
    // application launch.
    self.viewController = [[[ViewController alloc]
        initWithNibName:@"ViewController" bundle:nil]
        autorelease];

    self.window.rootViewController = self.viewController;

    // Do any additional setup after loading the view,
    // typically from a nib.
    self.facebook= [[Facebook alloc]
        initWithAppId:@"YOUR_APPID_HERE" andDelegate:self];

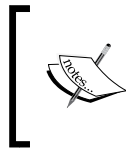
    // Check and retrieve authorization information
   NSUserDefaults *defaults = [NSUserDefaults
        standardUserDefaults];
    if ([defaults objectForKey:@"FBAccessTokenKey"] &&
        [defaults objectForKey:@"FBExpirationDateKey"]) {
        self.facebook.accessToken = [defaults
            objectForKey:@"FBAccessTokenKey"];
        self.facebook.expirationDate = [defaults
            objectForKey:@"FBExpirationDateKey"];
    }

    // Set up the permissions to use for this App
    NSArray *permissions = [[NSArray alloc]
        initWithObjects:
            @"user_likes",
            @"user_birthday",
            @"user_interests",
            @"read_stream"
            , nil];

    // Check to ensure that we have a valid
    // session object
    if (![self.facebook isValidSession]) {
```

```
[self.facebook authorize:permissions];  
}  
[permissions release];  
  
[self.window makeKeyAndVisible];  
return YES;  
}
```

In the preceding code snippet, we declare an `NSArray` object variable `permission` that will be used to store each of our permissions that we want to request. We then pass this variable into our `authorize` method of the `facebook` object, before finally releasing the memory allocated by the object.



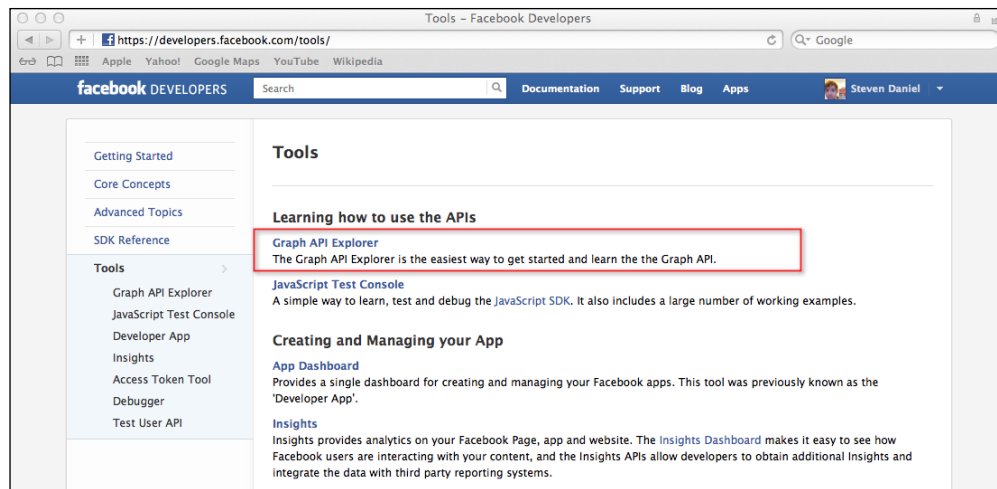
For more information on the full list of available permissions that are available to you, refer to the Facebook Permissions reference at the following URL: <https://developers.facebook.com/docs/authentication/permissions/>.

Using the Graph API

The Graph API is the core of Facebook, and represents a simple social graph pertaining to people and each of the connections they have, by representing each of the objects in the graph (for example; people, photos, events, and pages) and the connections between them (for example; friend relationships, shared content, and photo tags). You can access the Graph API by passing the Graph Path to the `request` method.

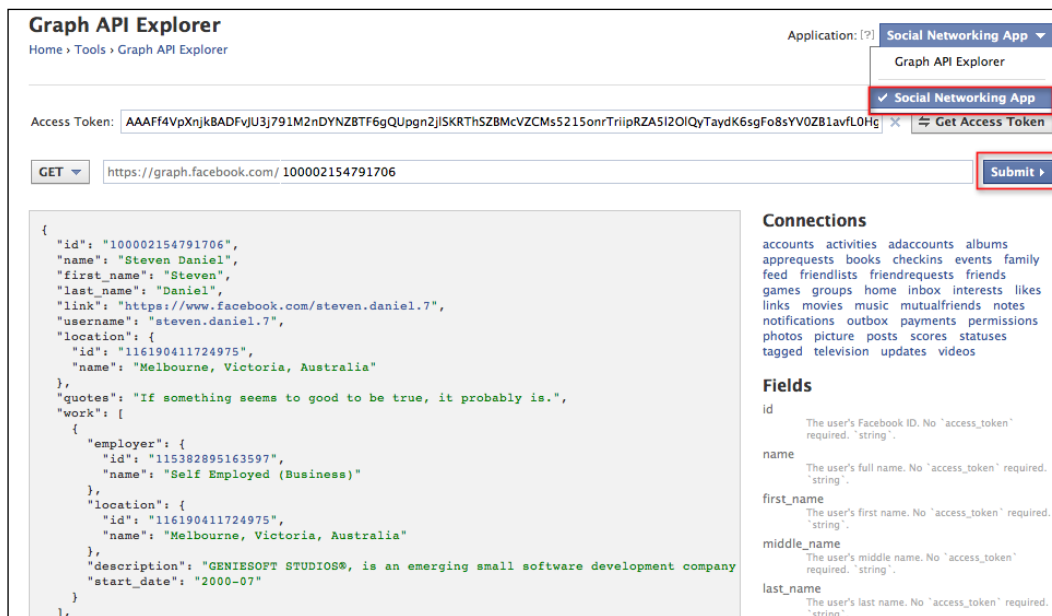
Within the Facebook Developer's website, you can access the Graph API Explorer to learn more about what types of information are returned, and see a visual representation of the data schema that the **Facebook Query Language (FQL)** produces.

1. Log in to the **Facebook Developers** website at the following website address: <http://developers.facebook.com/tools/>.
2. Next, from the **Tools** section, click on the **Graph API Explorer** link.



This will display the **Graph API Explorer** window and display an explanation of each of the data fields returned within the center view.

- From the **Application** section, select **Social Networking App** from the drop-down menu.



- Then, click on the **Submit** button to retrieve all the basic information for the current user ID.

You have seen how easy it is to use the Graph API Explorer to retrieve a visual representation of the data for various types of information. Let's take a look at how we can access information about the currently logged in user.

Open the `ViewController.m` implementation file, located within the `FacebookSampleApp` folder, and enter in the method as shown in the following code snippet:

```
#pragma mark - Facebook GraphAPI Method
- (void) getGraphAPIData
{
    // Make a call using the Facebook Query Language to
    // get the current user details.
    NSMutableDictionary *params = [NSMutableDictionary
    dictionaryWithObjectsAndKeys:
        @"SELECT quotes, uid, name, pic FROM
        user WHERE uid=me()", @"query",
        nil];

    [self.facebook requestWithMethodName:@"FQL.query"
    andParams:params
    andHttpMethod:@"POST"
    andDelegate:self];
}
```

In the preceding code snippet, we declare an `NSMutableDictionary` object variable `param` that will be used to pass an SQL query to the FQL object. This enables us to use an SQL-style interface using the Graph API to query the data. We then call the `requestWithMethodName` method of the `facebook` object and set up `type` to be `FQL.query`, which tells the request that we are passing in a query string and returns the contents as a dictionary array object.

```
// This method gets called when the Graph API
// call has completed.
- (void) request: (FBRequest *) request didLoad: (id) result
{
    if ([result isKindOfClass:[NSArray class]]) {
        result = [result objectAtIndex:0];
    }

    // This callback can be a result of getting the user's
    // basic information or getting the user's permissions.
    if ([result objectForKey:@"name"]) {
        // Retrieve back the basic user information.
    }
}
```

```

NSString *concatString = [[NSString alloc]
initWithFormat:@"ID: %@\nName: %@\nQuotes: %@\n",
[result objectForKey:@"uid"],
[result objectForKey:@"name"],
[result objectForKey:@"quotes"]];

// Get the profile image
UIImage *image = [UIImage imageWithData:[NSData
dataWithContentsOfURL:[NSURL URLWithString:[result
objectForKey:@"pic"]]]];
self.imgPhoto.image = image;
self.userInfoDetails.text = concatString;
}
}

```

In the preceding code snippet, when our `requestWithMethodName` method completes, it calls the `didLoad` method of the `request` method. This method parses the result using a JSON call. Next, we check to see the type that has been returned. If multiple results are returned, an `NSArray` object is returned, otherwise an `NSDictionary` object is returned for single result values. We then set our result to point to the first position within the array, and then retrieve each of the fields for the `uid`, `name`, and `quotes`. We then declare a `UIImage` variable `image` and then typecast the profile picture to be of type `UIImage`, before assigning this to our `imgPhoto` control on our form, as well as displaying the relevant profile details.

```

// This method is called when an error has occurred
// while retrieving GraphAPI details.
-(void)request:(FBRequest *)request didFailWithError:(NSError
*)error
{
    NSLog(@"An error occurred obtaining details: %@",error);
}

```

In the preceding code snippet, if any JSON parsing errors are determined when our `requestWithMethodName` method completes, the `didFailWithError` method is called. Any error information is contained within the `NSError` variable `error` object.



For more information on the Graph API and FQL Query language, please refer to the Facebook API and FQL reference material at the following locations:

- <https://developers.facebook.com/docs/reference/api/>
- <https://developers.facebook.com/docs/reference/fql/>

Integrating with social channels

The Facebook iOS SDK provides you with an easy way of making your applications integrate with the Facebook social channels. Using these social channels allows your users to submit posts to their timeline, or send notification requests to your friends.

The iOS SDK provides you with a method to integrate through the social channels using the Facebook platform dialogs. The following table lists the dialogs that are currently supported by Facebook:

Social channel dialogs	Description
Feed dialog	This dialog is used for publishing posts to a user's news feed.
Requests dialog	This type of dialog allows you to send a request to one or more of your friends.

When using Facebook requests, these social channel dialogs provide you with a great way of allowing users to invite their friends to your iOS application or even accept gifts from your friends. Requests are sent using the Request dialog, and if the user's iOS device supports **push** notifications, they will receive a push notification via the Facebook iOS application whenever a notification request is sent.

Sending requests provides you with a great way of promoting your iOS apps on Facebook to increase download sales. Now that we have an understanding of what requests are, let's take a look at how we can implement this within our iOS application.

1. Open the `ViewController.m` implementation file, located within the `FacebookSampleApp` folder, and enter in the method as shown in the following code snippet:

```
#pragma mark - Facebook Method
// Method to send a notification request to a
// group of friends.
- (void)sendNotificationRequest
{
    NSMutableDictionary *params =
        [NSMutableDictionary dictionaryWithObjectsAndKeys:
         @"invites you to check out some great stuff.",
         @"message",
         @"Check this out", @"notification_text",
         nil];

    // Display the Facebook Request Notifications DialogBox
    [self.facebook dialog:@"apprequests"
     andParams:params
     andDelegate:self.mainDelegate];
}
```

```

    }

    // FBDialogDelegate
    - (void)dialogDidComplete:(FBDialog *)dialog {
        NSLog(@"dialog completed successfully");
    }

```

In the preceding code snippet, we declare an `NSMutableDictionary` object variable `params` that will be used to pass the message and the notification text, using the `@message` and `@notification_text` parameters. We then use the `dialog` method of our `facebook` object, and tell the dialog that we want to use the `apprequests` dialog. Finally, we declare the method called `dialogDidComplete`, which gets called if the requests dialog gets successfully displayed to the user.

The use of incorporating **News feeds** within your application allows you to post information to the current user's main timeline page. This page is normally shown immediately upon the user signing in to Facebook. Let's take a look at how we can implement this within our iOS application.

2. Open the `ViewController.m` implementation file, located within the `FacebookSampleApp` folder, and enter in the method as shown in the following code snippet:

```

// Method to post a message to the current user's Wall.
- (void)postMessagetoWall
{
    NSMutableDictionary *params =
        [NSMutableDictionary dictionaryWithObjectsAndKeys:
         @"Testing FacebookSampleApp Feed Dialog", @"name",
         @"Using Feed Dialogs within iOS are great.",
         @"caption",
         @"Click to check out my BlockHeadz game on the
         AppStore",
         @"description",
         @"http://itunes.apple.com/app/block-headz
         /id386884355?mt=8#",
         @"link",
         @"http://geniesoftstudios.com/blog/wp-
         content/uploads/2011/03/blockhead.png", @"picture",
         nil];

    // Display the Facebook feed dialog with our array.
    [self.facebook dialog:@"feed"
     andParams:params
     andDelegate:self.mainDelegate];
}

```

In the preceding code snippet, we declare an `NSMutableDictionary` object variable `parameters` that will be used to pass the message and the notification text, using the `@name`, `@caption`, `@description`, `@link`, and `@picture` properties. These define what information is displayed when posting the message to the user's timeline. Next, we use the `dialog` method of our `facebook` object, and tell the dialog that we want to use the **feed** dialog, since we are posting details to the timeline.

How to handle errors

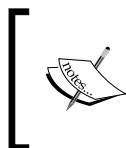
Handling errors within your application when using the Facebook SDK is extremely easy. Should any errors occur within your iOS application, the `FBRequestDelegate` and the `FBDialogDelegate` protocols will immediately handle them.

```
// This method is called when an error has occurred while retrieving
// GraphAPI details.
- (void)request:(FBRequest *)request didFailWithError:(NSError
*)error
{
    NSLog(@"An error occurred obtaining details: %@",error);
}
```

In the preceding code snippet, the `didFailWithError` method gets called upon whenever an error during the requesting of information using the Graph API occurs. Should any errors occur, this information will be returned in the error object.

```
- (void)dialog:(FBDialog*)dialog didFailWithError:(NSError *)error
{
    NSLog(@"An error occurred obtaining details: %@",error);
}
```

In the preceding code snippet, the `didFailWithError` method is invoked if there is an error during the dialog process. Should any errors occur, you can access this information using the error object.



For more information on how to handle errors when using the Facebook iOS SDK, you can refer to the Facebook iOS Reference documentation at the following URL: <https://developers.facebook.com/docs/mobile/ios/build/#errors>.

Implementing the `postMessageButton:` method

Our next step is to start implementing a method that will be responsible for sending notification messages, posting messages to the user's timeline, as well as obtaining a user's profile information using the Graph API when the user presses the **Action** button.

Open the `ViewController.m` implementation file, located within the `FacebookSampleApp` folder, and enter in the following code snippet:

```
// Called when the user presses the Post Message Button
- (IBAction)postMessage:(id)sender {

    // Define an instance of our action sheet
    UIAlertController *actionSheet;

    // Initialize our action sheet with the
    // different mapping types.
    actionSheet = [[UIAlertSheet alloc]
        initWithTitle:@"Choose from the list below"
        delegate:self
        cancelButtonTitle:@"Cancel"
        destructiveButtonTitle:@"Close"
        otherButtonTitles:@"Send Notification",
        @"Submit new post",
        @"Obtain User Details",nil];

    // Set our Action Sheet style and then
    // display it to the user.
    actionSheet.actionSheetStyle =
        UIBarStyleBlackTranslucent;
    [actionSheet showInView:self.view];
}

// Delegate that handles the chosen action sheet options
- (void)actionSheet:(UIAlertSheet
    *)actionSheet clickedButtonAtIndex:(NSInteger)buttonIndex
{
    // Determine the chosen item
    switch (buttonIndex) {
        case 1: [self sendNotificationRequest]; break;
        case 2: [self postMessageToWall]; break;
        case 3: [self getGraphAPIData]; break;
        default: break; // Catch the Close button and exit.
    }
}
```

In the preceding code snippet, we declare and instantiate an `actionSheet` object that is based on the `UIActionSheet` class, and then initialize our action sheet to display the different types of actions we want to perform, to have displayed as the list of options to choose from. Next, we proceed to set the style for our action sheet using the `actionSheetStyle` property of the `UIActionSheet` class, and then display the action sheet into the current view using the `showInView:self.view` method. In our next part, we define a delegate method to determine the button that was pressed from the action sheet and use the `clickedButtonAtIndex` method of the `actionSheet` property. We then check the value of the `buttonIndex` variable to determine the index of the button that was pressed.

Implementing the `loginButton:` method

Next, we need to implement the **Login** button. This allows our application to display the Facebook login page where we can provide our login credentials, and then returns back to us the Social Networking iOS application.

Open the `ViewController.m` implementation file, located within the `FacebookSampleApp` folder, and enter in the following code snippet:

```
// Handle when the login button is pressed.
- (IBAction)loginButton:(id) sender
{
    if (![self.facebook isValidSession])
    {
        NSLog(@"facebook session");
        NSArray *permissions = [[NSArray alloc]
            initWithObjects:@"email",@"publish_actions", nil];
        [self.facebook authorize:permissions];
        [permissions release];
    }
    else
    {
        NSLog(@"session still valid");
    }
}
```

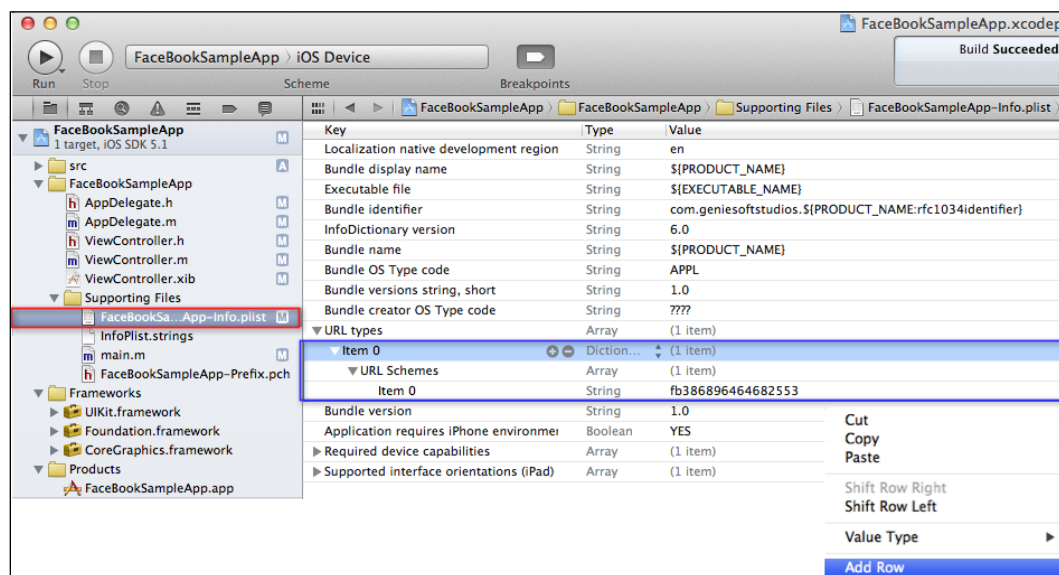
In the preceding code snippet, we use the `isValidSession` method of the `facebook` object to determine if we still have a valid connection to the `facebook` instance. If it proves that our session has expired, we initialize and pass to our `authorize` method of the `facebook` class permissions to request for accessing the user's e-mail, and allow the iOS to publish to the **Open Graph API** actions. Finally, we release the memory that has been allocated by our `permissions` object.

Finishing up

We just have a few more things to implement before we have a complete working application. We will need to make some changes to our application's property list to enable SSO support when the application is run.

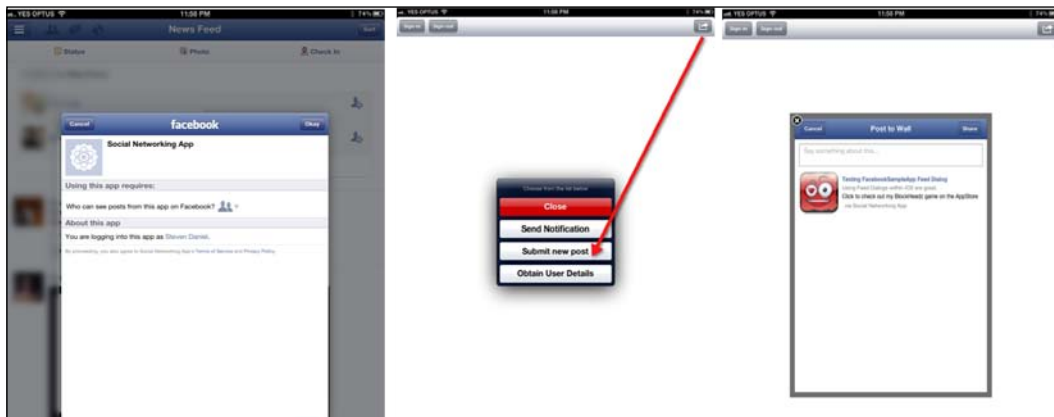
In order to make our application enable SSO, we will need to modify our application's property list file. This can be achieved by following these simple steps:

1. Select the `FacebookSampleApp-info.plist` file from within **Project Navigator**.
2. Next, right-click within the center of the panel, then select **Add Row** from the pop-up list.
3. Add a new entry called `URL Types`, and set its type to **Array**.
4. Right-click and select **Add Row** inside the `URL types`.
5. Then, create a new item called `Item 0`, and set its type to **Dictionary**.
6. Next, create a new entry called `URL Schemes`, and set its type to **Array**.
7. Then, create a new item called `Item 0`, and set its type to **String**.
8. Finally, using the Facebook AppID value when we registered our mobile application, enter this in the Value field. You must prefix this with `fb` followed by your App ID for this to work correctly.



As you can see from the preceding screenshot, we have modified our `.plist` file so that it can support SSO. We specifically created and registered two URL variables: **URL Types** and **URL Schemes**, so that it can uniquely identify your application with iOS.

Congratulations, we have finally implemented the methods for our Facebook Social Networking application. Next, build and run the application by choosing **Product | Run** from the **Product** menu, or alternatively press *Command + R*. The following screenshot shows the application running on the iOS device:



From the preceding screenshot, you can see that when we first load our Social Networking application, we receive the authorization dialog box with the permissions that we have requested. Once the user has pressed on the **Okay** button, the dialog box will disappear, and our iOS application will be displayed (as shown in the second screenshot). We then press the **Action** button and select the **Submit new post** button to display the post new message to our wall, which is shown in the final screenshot.

Summary

In this chapter, we learned how to create a simple Social Networking application using the Facebook iOS SDK. We looked at how to register our mobile application with Facebook, and how to then download the Facebook iOS SDK, and then import this into our project. We then looked at how we can implement the SSO feature within our application, and how we can go about using the Facebook methods and APIs to communicate with Facebook to post directly to the current user's timeline, as well as sending notifications to friends.

We also looked into the Open Graph API, and how we can use the Facebook Query Language to pass SQL Query-like syntax to retrieve information about the current user. To end the chapter, we looked at how we can use the various methods that have been made available to us within the Facebook iOS SDK to handle errors within our iOS applications cleanly.

In the next chapter, we will look at how to create an application that will allow us to work with external displays to display output to Apple TV using AirPlay. We will look at the different types of transition effects that we can incorporate into our application, to create a photo slideshow application.

9

External Displays using Airplay and Core Image

Since the release of iOS 4.2, developers have been able to use Airplay to stream Photos, videos, and audio to an Apple TV capable device. With the release of iOS 5, this has been greatly improved, and makes it even easier to wirelessly mirror everything on your iPad 2 to an HDTV through Apple TV.

The Core Image framework is a hardware-accelerated framework that provides an easier way for you to enhance your photos and videos by creating some amazing effects using your camera and image editing applications. Core Image provides several built-in filters: color effects, distortions, and transitions, as well as several advanced features: auto-enhance, red-eye reduction, and facial recognition.

In this chapter, we will be taking a closer look at these frameworks and how we can use them within our applications, to apply image filter effects using the `CIFilter` class, as well as implementing Airplay to allow us to output content to another device using Apple TV. Finally, we will learn how we can output content to an external monitor, and then adjust the screen resolution using the `UIScreen` class.

In this chapter we will:

- Get an overview of the types of technologies that we will be using
- Learn about the AirPlay and Core Image frameworks
- Create a simple AirPlay and Core Image application
- Implement methods to access the iOS device's camera and photo library
- Implement the method to apply image filter effects using the `CIFilter` class
- Implement methods to apply transition effects using the `CATransition` class
- Implement methods to output content to an Apple TV and external VGA

We have an exciting project ahead of us; so let's get started.

Overview of the technologies

The **External Displays** application makes reference to the following frameworks **MediaPlayer**, **CoreImage**, and **QuartzCore**. With the release of iOS 5, the Media Player framework has been updated to give developers the ability to easily incorporate Airplay into their applications, and lets you stream audio and video content from any iOS device to any Airplay device that is capable of playing audio and video to a nearby Apple TV receiver.

The Core Image framework is an extensible image processing technology architecture that has been built into Mac OS X v10.4 and iOS 5.0. This framework leverages the programmable graphics hardware to provide near real-time, pixel-accurate image processing of graphics, as well as video processing. The Core Image framework comes with over 100 built-in filters that are ready-to-use by filter clients who want to support image processing in their applications.

The Core Image **Application Programming Interface (API)** is a component of the Quartz Core framework that provides access to several built-in image filters for both video and still images, as well as providing support for creating custom filters.

In this section, we will take a look at how we can create a simple application to playback video content on an iOS device and how to output this to an Apple TV device. We will also look at how we can apply filter effects and transitions to images and output this to an external display using the Apple **Video Graphics Adaptor (VGA)**.

Building the ExternalDisplays application

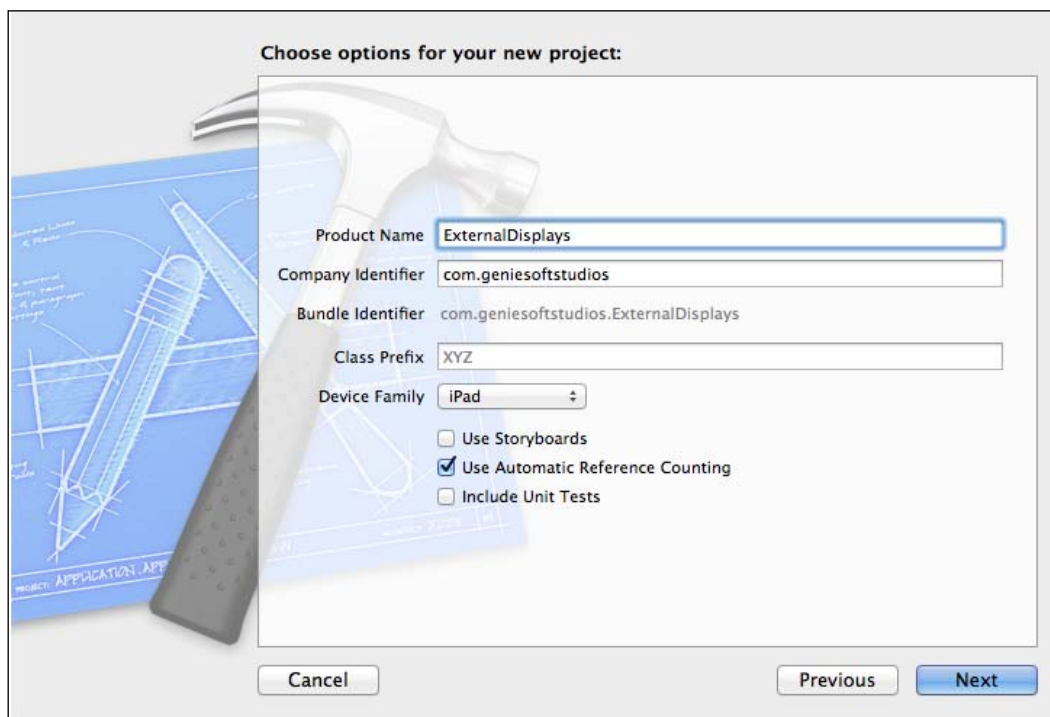
Playing videos is one of the most common tasks that can be done on any iOS device; all videos must be played and displayed in full-screen. Before we can play any videos, we need to add the **Media Player** framework into our project. With Core Image, Apple has provided more than 100 image-processing filters to make it easier for you to provide support to these within your own applications, to enhance the sharpness of images or even red-eye removal from photos.

Before we can proceed, we first need to create our `ExternalDisplays` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 3, VoiceRecorder App – Audio Recording and Playback*, under the section named *Building the VoiceRecorder App*.

It is very simple to create this in Xcode. Just follow the steps listed here.

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode project**, or **File | New Project**.

3. Select the **Single View Application** template from the list of available templates.
4. Click on the **Next** button to proceed with the next step in the wizard.
5. Next, enter in `ExternalDisplays` as the name for your project
6. Select **iPad** from under the **Device Family** drop-down list.
7. Ensure that the **Use Storyboards** checkbox has not been selected.
8. Ensure that the **Use Automatic Reference Counting** checkbox has been selected.
9. Ensure that the **Include Unit Tests** checkbox has not been selected.
10. Click on the **Next** button to proceed with the next step in the wizard.



11. Specify the location where you would like to save your project.
12. Then, click on the **Create** button to continue and display the Xcode workspace environment.

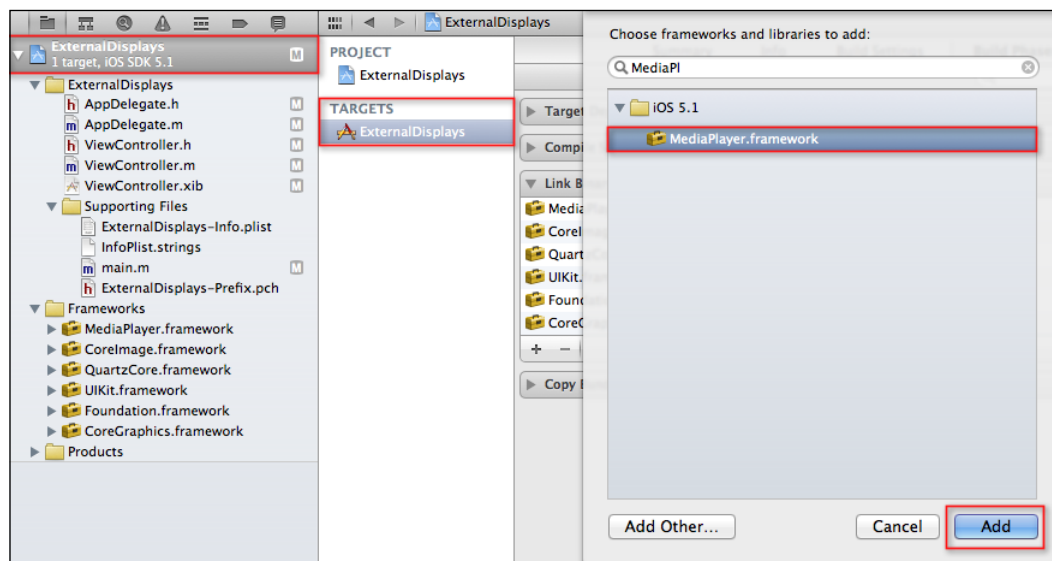
Adding the Media Player framework to our project

Now that we have created our `ExternalDisplays` project, we need to add each of the necessary frameworks to our project that will enable us to playback video and apply different image filter effects and transitions.

To add the Media Player framework, select **Project Navigator Group**, and then follow these simple steps:

1. Click and select your project from **Project Navigator**.
2. Then select your project target from under the **TARGETS** group.
3. Select the **Build Phases** tab.
4. Expand the **Link binary with Libraries** disclosure triangle.
5. Use **+** to add the library you want.
6. Select `MediaPlayer.framework` from the list of available frameworks. You can also search if you can't find the framework you are after, from within the list.

If you are still confused as to how to go about adding the frameworks, take a look at this screenshot, which highlights the areas that you need to select (surrounded by a rectangle).

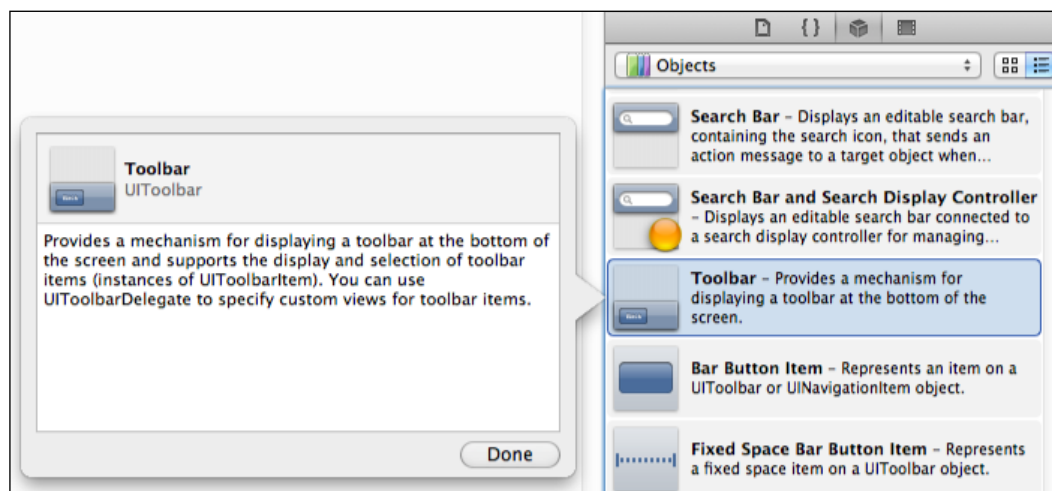


7. Repeat steps 1 to 6 to add the `CoreImage.framework` and `QuartzCore.framework` frameworks to the `ExternalDisplays` project.

Creating the main application screen

Now that we have successfully added our frameworks into our project, we need to start building our user interface that will allow us select a photo or video from the photo library that will allow for the playback of videos and apply filter effects to the images. This screen will be very simple and will consist of just a View controller and a toolbar.

1. Select the `ViewController.xib` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UIToolbar) **Toolbar** control, and add it to the top of our view.

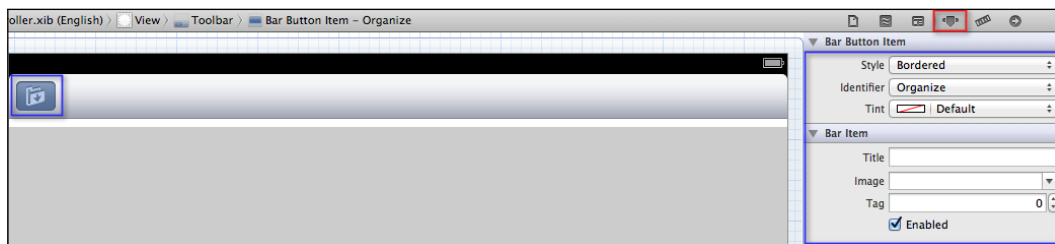


We have added our `UIToolbar` toolbar control to our view controller; our next step is to start adding the button objects that make up our user interface. So let's proceed to the next section.

Adding the Browse button

Our next step is to add a **Browse** button to `UIToolBar`; this will be responsible for allowing you to choose a photo image or video from the iOS device's library. This can be achieved by following these simple steps:

1. Select the `ViewController.xib` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control to the top left-hand corner of `UIToolBar`.
3. From the **Attributes Inspector** section, change the Identifier property to **Organize**.
4. Change the **Style** property to **Bordered**.



Now that we have added our **Browse** button to our View controller, our next step is to add the **Camera** button that will be responsible for allowing you to take a photo using the iOS device camera when the button is clicked. So let's proceed with the next section.

Adding the Camera button

Now that we have added our **Browse** button, our next step is to add another button that will be responsible for allowing the user to use the iOS camera to take photos and record a video when it has been clicked. This can be achieved by following these simple steps:

1. Select the `ViewController.xib` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UIBarButtonItem`) **Bar Button Item** control after the **Browse** button, located within `UIToolBar`.
3. From the **Attributes Inspector** section, change the **Identifier** property to **Camera**.
4. Change the **Style** property to **Bordered**.

Now that we have added our **Camera** button to our View controller, our next step is to add a (UIBarButtonItem) **Flexible Space Bar Button Item** control that will be used to fill in the space between the **Browse** button and the **Camera** button.

Adding the Play Video button

Now that we have added our **Camera** button, our next step is to add another button that will be responsible for playing video content when it has been clicked. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control after the UIBarButtonItem control, located within UIToolBar.
2. From the **Attributes Inspector** section, change the **Identifier** property to **Play**.
3. Change the **Style** property to **Bordered**.

Adding the Transitions button

Now that we have added our **Play Video** button, our next step is to add another button that will be responsible for applying filter effects and transitions to images when it has been clicked. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control after the **Play** UIBarButtonItem control, located within UIToolBar.
2. From the **Attributes Inspector** section, change the **Identifier** property to **Action**.
3. Change the **Style** property to **Bordered**.

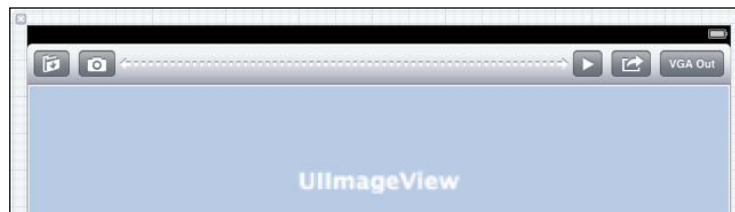
Adding the VGA Out button

Now that we have added our **Transitions** button, our next step is to add our final button that will be responsible for outputting content to an external VGA device, using the VGA cable extension for the iPad when this has been clicked. This can be achieved by following these simple steps:

1. From **Object Library**, select-and-drag a (UIBarButtonItem) **Flexible Space Bar Button Item** control after the **Camera** button, located within UIToolBar.
2. Next, from **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control after ActionUIBarButtonItem, located within UIToolBar.
3. From the **Attributes Inspector** section, change the **Identifier** property to **Custom**.

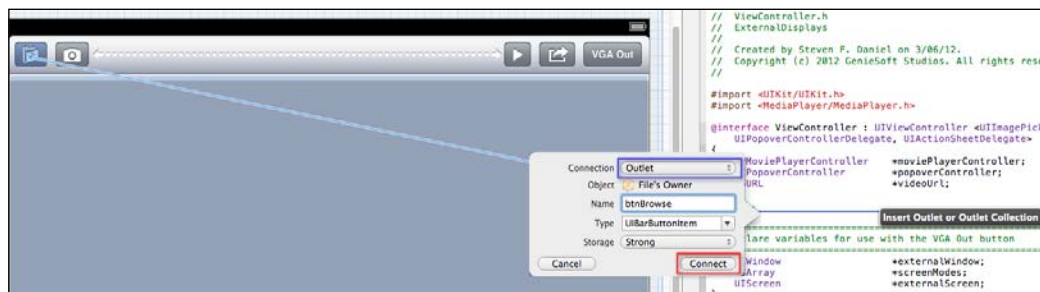
4. Change the **Style** property to Bordered.
5. Then, change the **Title** property to VGA Out.
6. Next, from **Object Library**, select-and-drag an (UIImageView) **Image View** control, and place it underneath the toolbar.
7. Resize the control so that it fills the container window of the View controller.

If you have followed the steps correctly, the completed **View Controller** screen should look similar to the following screenshot; feel free to adjust yours accordingly:



Our next step is to create the outlets for the **Browse**, **Camera**, **Play Video**, **Transitions**, and **VGA Out** buttons, as well as our UIImageView form fields. Creating these outlets will allow us to access these controls from within our code and make modifications to the control properties. To create an Outlet, follow these simple steps:

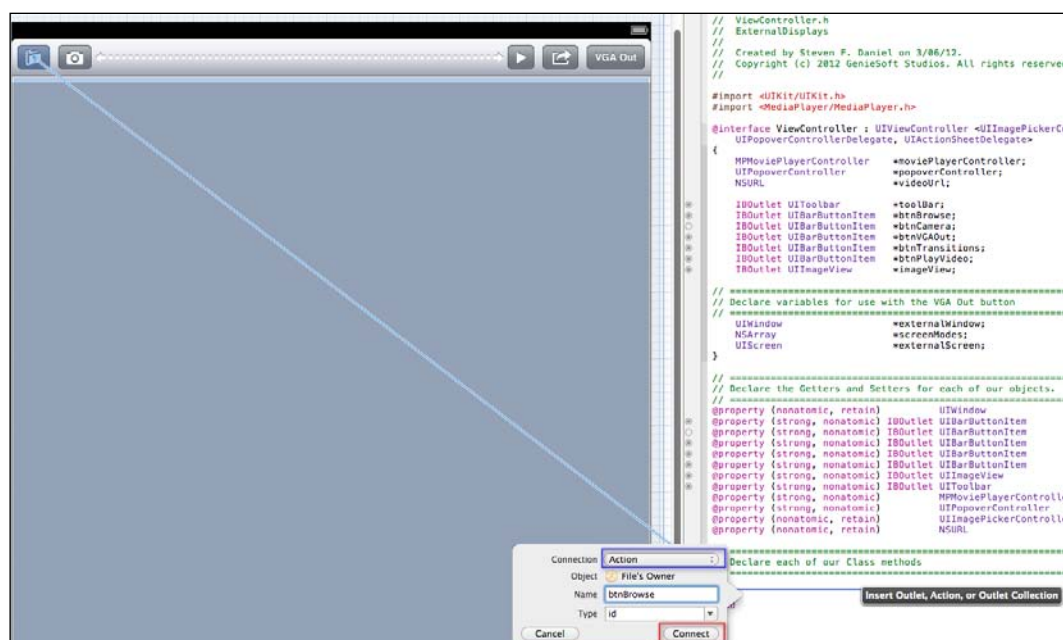
1. Open **Assistant Editor** by choosing **Navigate | Open In Assistant Editor**, or press **Option + Command + ,**.
2. Ensure that the `ViewController.h` interface file is displayed inside the **Assistant Editor** window.
3. Select the **Browse** (UIBarButtonItem) control, then hold down the **Control** key, and drag it into the `ViewController.h` interface file.
4. Choose **Outlet** from the **Connection** drop-down list, for the connection to be created.
5. Enter in `btnBrowse` for the name of the property to be created.



- Repeat steps 3 to 5 to create the `IBOutlet`s for the **Camera**, **Play Video**, **Transitions**, **VGA Out**, **Toolbar**, and **UIImageView** controls, while providing the following namings for each as follows: `btnCamera`, `btnPlayVideo`, `btnTransitions`, `btnVGAOut`, `imageView`, and `toolbar`.

Now that we have created the instance variable Outlets for our controls, we need to create the associated Actions for those Outlets events. Creating these actions allows an event to be fired when the button has been pressed. To create an Action, follow these simple steps:

- With the `ViewController.h` interface file still displayed to the left of the `ViewController.xib` View controller, select the **Browse** (`UIBarButtonItem`) control, then hold down the **Control** key, and drag it into the `ViewController.h` interface file.
- Choose **Action** from the **Connection** drop-down list, for the connection to be created.
- Enter in `btnBrowse` for the name of the property to be created.



- Repeat steps 1 to 3 to create the `IBActions` for the **Camera**, **Play Video**, **Transitions**, and **VGA Out** controls, while providing the following namings for each as follows: `btnCamera`, `btnPlayVideo`, `btnTransitions`, and `btnVGAOutput`.

Now that we have successfully connected each of our controls, and created the required outlets and associated action methods, we can start taking a look at building the functionality into our application.

Functionality

Well done! You have made it this far; we have successfully finished building the user interface for our `ExternalDisplays` application. Our next step is to implement the methods for each of our button controls. This gives us the ability to choose an image from the iOS device's photo library or take a photo using the camera, and then apply filter effects to those images. We will also look at implementing the methods that will be responsible for outputting the content to both an Apple TV and external monitor device.

Implementing the View Controller class

We are now ready to start adding additional content to our View Controller class. We will need to import some important header files, as well as declare some object variables that we will be using throughout our application.

We will also need to extend our class so that we can use the `ImagePickerController`, `NavigationController`, `PopoverController`, and `ActionSheet` delegate classes.

1. Open the `ViewController.h` interface file, located within the `ExternalDisplays` folder, and enter in the following highlighted code sections:

```
// ViewController.h
// ExternalDisplays
// Created by Steven F. Daniel on 3/06/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

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

@interface ViewController:
    UIViewController<UIImagePickerControllerDelegate,
    UINavigationControllerDelegate,
    UIPopoverControllerDelegate, UIActionSheetDelegate>
{
    MPMoviePlayerController *moviePlayerController;
    UIPopoverController *popoverController;
    NSURL *videoUrl;

    IBOutlet UIToolbar *toolBar;
}
```

```

IBOutlet UIBarButtonItem *btnBrowse;
IBOutlet UIBarButtonItem *btnCamera;
IBOutlet UIBarButtonItem *btnVGAOut;
IBOutlet UIBarButtonItem *btnTransitions;
IBOutlet UIBarButtonItem *btnPlayVideo;
IBOutlet UIImageView *imageView;

// Declare variables for use with the VGA Out button
UIWindow *externalWindow;
NSArray *screenModes;
UIScreen*externalScreen;
}

// Declare the Getters and Setters for each of our
// objects.
@property (nonatomic, retain) UIWindow *externalWindow;
@property (strong, nonatomic) IBOutlet UIBarButtonItem
    *btnBrowse;
@property (strong, nonatomic) IBOutlet UIBarButtonItem
    *btnCamera;
@property (strong, nonatomic) IBOutlet UIBarButtonItem
    *btnVGAOut;
@property (strong, nonatomic) IBOutlet UIBarButtonItem
    *btnTransitions;
@property (strong, nonatomic) IBOutlet UIBarButtonItem
    *btnPlayVideo;
@property (strong, nonatomic) IBOutlet UIImageView
    *imageView;
@property (strong, nonatomic) IBOutlet UIToolbar
    *toolBar;

@property (strong, nonatomic)
    MPMoviePlayerController *moviePlayerController;
@property (strong, nonatomic)
    UIPopoverController *popoverController;
@property (nonatomic, retain)
    UIImagePickerController *imagePicker;
@property (nonatomic, retain) NSURL *videoUrl;

// Declare each of our Class methods
- (IBAction)btnBrowse:(id)sender;
- (IBAction)btnCamera:(id)sender;
- (IBAction)btnPlayVideo:(id)sender;
- (IBAction)btnTransitions:(id)sender;
- (IBAction)btnVGAOutput:(id)sender;

@end

```

In the preceding code snippet, we start by importing our interface header information for our `MediaPlayer.h` interface file to allow for the playback of audio and video content, as well as setting up instance variables to both our Movie Player and Popover Controllers that will be used to playback video content and select images from the iOS device's photo library.

We then extended our class, to include each of the following class protocols: `UIImagePickerControllerDelegate`, `UINavigationControllerDelegate`, `UIPopoverControllerDelegate`, and `UIActionSheetDelegate`. This is done so that we can access each of their respective properties and methods. Finally, we declared a series of variables that will enable us to output the content to an external monitor, as well as the `NSArray` object `screenModes` which will contain each of the allowable screen resolutions.

2. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder, and modify the `viewDidLoad` method, as shown in the following code snippet:

```
// ViewController.m
// ExternalDisplays
// Created by Steven F. Daniel on 3/06/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "ViewController.h"
#import "QuartzCore/QuartzCore.h"

@interface ViewController ()

@end

@implementation ViewController

@synthesize btnBrowse,btnCamera,btnPlayVideo btnVGAOut;
@synthesize btnTransitions, toolBar;
@synthesize popoverController,imagePicker,imageView;
@synthesize moviePlayerController;
@synthesize externalWindow,videoUrl;

// Initialize our view and the objects when it is loaded.
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Do any additional setup after loading the
    // view, typically from a nib.
}
```

```

[self.toolbar setTintColor:[UIColor purpleColor]];

self.externalWindow.hidden = YES;
self.btnPlayVideo.enabled = NO;
self.btnTransitions.enabled = NO;
self.btnVGAOut.enabled = NO;
}

```

In the preceding code snippet, we import the interface file header information for our `QuartzCore.h` interface files, so that we can access the core image and transition class methods. Next, we initialize our toolbar object within our view controller, set the background color of our toolbar to purple, and then initialize each of the buttons.

Implementing the `btnBrowse:` method

Our next step is to implement the **Browse** button. This method will be responsible for allowing you to choose an image from the iOS device's photo library and display this using the camera roll view within a popover.

Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder and modify the `btnBrowse` method, as shown in the following code snippet:

```

// Called when the user presses the Photo Library button
- (IBAction)btnBrowse:(id)sender
{
    // Create image picker controller
    self.imagePicker= [[UIImagePickerController alloc] init];

    // Checks the device to make sure that the Photo Library
    // is available.
    if ([UIImagePickerController
        isSourceTypeAvailable:
        UIImagePickerControllerSourceTypePhotoLibrary]) {

        // Set source to the Photo Library
        self.imagePicker.delegate = self;
        self.imagePicker.sourceType =
            UIImagePickerControllerSourceTypePhotoLibrary;
        self.imagePicker.mediaTypes =
            [UIImagePickerController
            availableMediaTypesForSourceType:imagePicker.sourceType];
        self.imagePicker.allowsEditing = NO;
    }
}

```



```
self.popoverController = [[UIPopoverController alloc]
    initWithContentViewController:self.imagePicker];
popoverController.delegate = self;

[self.popoverController
    presentPopoverFromBarButtonItem:sender
    permittedArrowDirections:UIPopoverArrowDirectionUp
    animated:YES];
}
else {
    NSLog(@"Unable to access the Photo Library.");
}
}
```

In the preceding code snippet, we first check to see if we are able to access the iOS device's **Photo Library** using the `isSourceTypeAvailable` property of the `UIImagePickerController` class. Next, we initialize the properties of the `imagePicker` class to only display images from the photo library, and then declare a popover controller that will be passed through the image picker as the view.

The view controller is then designated as the delegate for the popover object before the popover is displayed to the user. The sender object passed through to this method references the **Photo Library** button in the toolbar. The object is passed through the popover Controller's `presentPopoverFromBarButtonItem:` method so that the popover is positioned directly above, and points to the button when displayed.

Implementing the `btnCamera:` method

Our next step is to implement the **Camera** photo button that will be responsible for displaying the camera view within the popover when pressed.

Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder, and enter in the method as shown in the following code snippet:

```
// Display the iOS Device' Camera using the backview as
// the default.
- (IBAction)btnCamera:(id)sender
{
    // Checks the device to make sure that it has a camera.
    if ([UIImagePickerController
        isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeCamera]) {

        // Create image picker controller
        self.imagePicker = [[UIImagePickerController alloc] init];
```

```

    // Set source to the Camera
    self.imagePicker.delegate = self;
    self.imagePicker.sourceType =
        UIImagePickerControllerSourceTypeCamera;
    self.imagePicker.cameraDevice =
        UIImagePickerControllerCameraDeviceRear;
    self.imagePicker.allowsEditing = NO;
    [self presentViewController:self.imagePicker
        animated:YES completion:nil];
}
else {
    NSLog(@"Unable to access the camera.");
}
}

```

In the preceding code snippet, we first check to see if we are able to access the iOS device camera using the `isSourceTypeAvailable` property of the `UIImagePickerController` class. We then create a new instance of our `UIImagePickerController` class. Next, we make the delegate point to itself and then set the `sourceType` property to use the camera, and then set the value of the `cameraDevice` property to use the rear camera. Finally, we display the camera interface, and the `UIImagePickerController` object is dismissed.

```

#pragma mark - Image Picker Delegate Methods
- (void)imagePickerControllerDidCancel:(UIImagePickerController
    *)picker
{
    [self.imagePicker dismissModalViewControllerAnimated:YES];
}

```

In the preceding code snippet, we start by declaring a delegate method for our image picker controller `imagePickerControllerDidCancel`. This delegate will be responsible for handling and taking care of closing the popover or camera session without making an image selection, or taking a picture whenever the **Cancel** button has been pressed.

```

// This method is called when the user has chosen an item
// from the image picker.
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info {

    // Determine the media type of the chosen item
    // from the image picker.
    NSString *mediaType = [info
        objectForKey:UIImagePickerControllerMediaType];
}

```

```
// Determine if we have chosen a Movie file from library.
if ( [ mediaType isEqualToString:@"public.movie" ])
{
    self.videoUrl= [info
        valueForKey:UIImagePickerControllerMediaURL];
    self.moviePlayerController= [[MPMoviePlayerController alloc]
        initWithContentURL:self.videoUrl];
    UIImage *thumbnail = [self.moviePlayerController
        thumbnailImageAtTime:0.0
        timeOption:
            MPMovieTimeOptionNearestKeyFrame];
    [self.imageView setImage:thumbnail];

    // Enable our video button but disable the image
    // transitions button.
    self.btnTransitions.enabled = NO;
    self.btnPlayVideo.enabled = YES;
}
// Else we have chosen an image file from library.
else if ([ mediaType isEqualToString:@"public.image" ] )
{
    self.moviePlayerController= nil;
    self.videoUrl= nil;
    self.btnPlayVideo.enabled = NO;
    self.btnTransitions.enabled = YES;
    UIImage *photoImage = [info
        objectForKey:UIImagePickerControllerOriginalImage];
    self.imageView.image = photoImage;
}
// Enable our VGA Out buttons
self.btnVGAOut.enabled = YES;

// Dismiss the PopOver Dialog
[self.popoverController dismissPopoverAnimated:NO];
}
```

In the preceding code snippet, we start by checking the type of media that has been chosen from the photo library using the `UIImagePickerControllerMediaType` property of `UIImagePickerController`, and then enable our video button but disable the image transition button. Next, we check to see if we selected a movie, and then obtain the file location of the chosen file, using `UIImagePickerControllerMediaURL`.

Next, we create a thumbnail image representation of the video, and set the image property of the `imageView` control. If we have determined an image has been chosen, we set the image property of the `imageView` control to the selected image, and enable the **VGA Out** button, before dismissing the `popoverController` control.

Implementing the `btnPlayVideo:` method

Our next step is to implement the **Play** button that will be responsible for playing our chosen movie from the camera roll when pressed.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder, and enter in the method as shown in the following code snippet:

```
// Handle Playback of the chosen movie when the Play
// Movie
// button is pressed.
- (IBAction)btnPlayVideo:(id)sender
{
    [self.navigationController
     dismissModalViewControllerAnimated:NO];

    // Determine to see if we have chosen a video from the
    // Photo Library.
    if (self.videoUrl == NULL)
    {
        // Display an alert message to the user.
        UIAlertView *alertView = [[UIAlertView alloc]
                                   initWithTitle:@"Video Playback"
                                   message:@"No video has been selected from the
                                   library." delegate:self
                                   cancelButtonTitle:nil
                                   otherButtonTitles:@"OK", nil];

        [alertView show];
    }
    else
    {
        // Initialise our moviePlayer Controller with
        // the path of the selected video.
        [self.moviePlayerControllersetContentURL:[NSURL
            fileURLWithPath:[videoUrlpath] isDirectory:NO]];
        [[NSNotificationCenter defaultCenter]
         addObserver:self
         selector:@selector(moviePlaybackComplete:)
         name:MPMoviePlayerPlaybackDidFinishNotification
```

```
        object:self.moviePlayerController];

// Add the movie player controller to the view
[self.view addSubview:self
    .moviePlayerController.view];

// Initialize the movie player properties.
self.moviePlayerController.fullscreen= YES;
self.moviePlayerController.scalingMode =
    MPMovieScalingModeAspectFit;

// Play the video
[self.moviePlayerController play];
    }
}
```

In the preceding code snippet, we start by dismissing our popover control from our current view to prevent it from being displayed when the video is played. Next, we check value of our `videoUrl` variable, to ensure that we have a value for our video file location, and display a message using the `UIAlertView` class; this is to prevent the application from crashing.

2. Next, we create an (NSURL) `fileURLWithPath` object variable that converts our `videoURL` variable to an object, which is what `MPMoviePlayerController` needs when it is being initialized. We then add the `MPMoviePlayerController` view to our View controller, so that it will appear on the screen, then specify that we want to display this in full-screen, and finally tell `moviePlayerController` to commence playback. Next, we send a dispatch notification method called `MPMoviePlayerPlaybackDidFinishNotification` to `NSNotificationCenter` to tell it what to do when the movie playback completes, as shown in the highlighted code in the previous snippet.
3. When we playback the video content within our iOS applications, you will sometimes need to modify the `scalingMode` property of the `MPMoviePlayerController` object. By setting this property, it will determine how the movie image adapts to fill the playback size that you have defined.

```
// Method to handle once the video has finished playback.
- (void)moviePlaybackComplete:(NSNotification
*)notification
{
    self.moviePlayerController= [notification object];
    [[NSNotificationCenter defaultCenter]
        removeObserver:self
        name:MPMoviePlayerPlaybackDidFinishNotification
```

```

        object:self.moviePlayerController];
    [self.moviePlayerController.view removeFromSuperview];
}

```

In this code snippet, we passed an object to the notification method. This is whatever we have passed in the previous code snippet, due to the `moviePlayerController` object. We start by retrieving the object using the `[notification object]` statement, and then reference it with the new `MPMoviePlayerController` pointer.

4. We then send a message back to the `NSNotificationCenter` method that removes the observer we previously registered with our `btnPlayVideo` method. We finally proceed with removing `moviePlayerController` from our display. In the next section, we will look at the steps involved in modifying our application, so that it can be displayed on a TV screen using Apple TV.



For more information on the comparison between the different scaling modes, refer to `MPMoviePlayerController` Class Reference at the following URL: http://developer.apple.com/library/ios/#documentation/mediaplayer/reference/MPMoviePlayerController_Class/Reference/Reference.html#//apple_ref/doc/c_ref/MPMoviePlayerController.

Using AirPlay to present application content to Apple TV

Starting with iOS 4.3, Apple decided to provide its developers with one of the most impressive frameworks ever imagined, which would allow developers to integrate AirPlay features into their own applications. With just a few lines of code, any iOS application can be modified to have the ability to stream video directly out to an Apple TV device.

To enable the AirPlay functionality, we will need to enable a special property on our `MPMoviePlayerController` object, by setting the `allowsAirPlay` property to `YES`.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder, and locate the following statement within the `btnPlayVideo` method:

```

// Add the movie player controller to the view
[self.view addSubview:self.moviePlayerController.view];

```

2. Next, add the following code snippet:

```
if ([self.moviePlayerController
    respondsToSelector:@selector(setAllowsAirPlay:)]) {
    [self.moviePlayerController setAllowsAirPlay:YES];
}
```

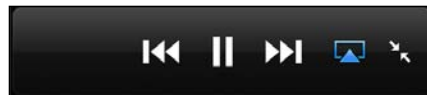
In this code snippet, we use the `respondsToSelector:` method of the `MPMoviePlayerController` object to cater for older iOS devices that don't support the `allowsAirPlay` property.

If you try to use this feature on a device that does not support it, it will cause a run-time error exception to occur, which will crash your application. In order to offer AirPlay only to those devices that support it, we need to place a conditional statement around the statement, which will check to see if the `MPMoviePlayerController` object supports the `allowsAirPlay` option.

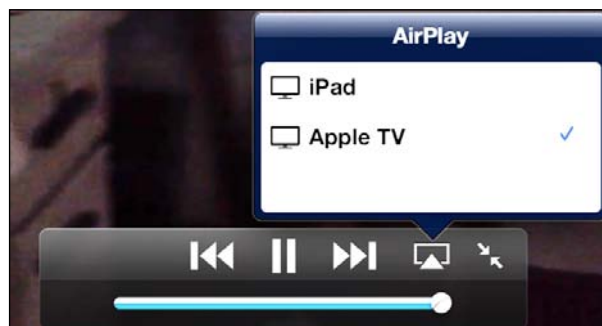


When this is set, it will cause an additional icon to appear within the movie player controller pane. You have no control, programmatically, over this icon placement.

3. Next, build and run your application, and click on the **Play** button. The following screenshot shows what this icon looks like when AirPlay has been enabled:



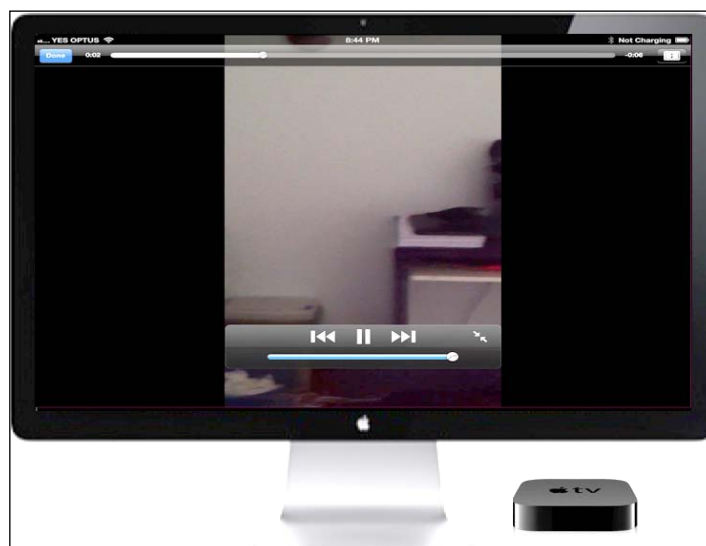
4. When the AirPlay icon has been enabled, you will be presented with a pop-up list of detected output device options to choose from.



5. If you choose the **Apple TV** option, as shown in the preceding screenshot, the output on your iOS device will disappear, and you will be notified that the video is being played on the Apple TV device. This is shown in the following screenshot:



6. Finally, you will see your video being displayed on an Apple TV device, as shown in the following screenshot:



As you can see, by following a few simple steps, you can easily incorporate the functionality needed to turn your existing applications into Airplay-aware applications.

The following list provides you with a few considerations to keep in mind when implementing AirPlay within your applications.

- Apple has only made this feature available on its most recent devices with the AirPlay 4.3 SDK, so there is no AirPlay support for iPhone 3G devices.
- When launching an AirPlay-enabled application, you will need to ensure that both your iOS device and your Apple TV software are running the same version of the OS, otherwise you could run into some problems.
- In order for your iOS devices to find other Apple AirPlay-enabled devices, you will need to ensure that you are on the same Wi-Fi network that your AirPlay devices are connected to.



For more information about the `MPMoviePlayerController` Class framework, refer to the following Apple Developer Documentation at the following URL: http://developer.apple.com/library/ios/#documentation/mediaplayer/reference/MPMoviePlayerController_Class/Reference/Reference.html#//apple_ref/doc/c_ref/MPMoviePlayerController.

Implementing the `btnTransitions:` method

Our next step is to implement the **Transitions** button that will enable us to choose a filter effect from a list of options, and have this applied to our loaded image within the `imageView` control.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder, and enter in the method as shown in the following code snippet:

```
// Displays our list of image filter effects
// within an ActionSheet
- (IBAction)btnTransitions:(id)sender
{
    // Initialise our Action Sheet with options
    UIAlertController *actionSheet = [[UIAlertSheet alloc]
        initWithTitle:@"Available Transitions"
        delegate:self
        cancelButtonTitle:@"Cancel"
        destructiveButtonTitle:@"Close"
        otherButtonTitles:@"Hue Adjust",
```

```

        @"Vibrance",@"Color Invert",
        @"Ripple Effect", nil];

    // Display the actionsheet to the view.
    [actionSheet showInView:self.view];
}

```

In the preceding code snippet, we declare, create, and initialize an `actionSheet` variable that sets up a list of filter options that can be chosen from, and then applied to an image. It is worth mentioning that the `UIActionSheet` class adopts the `UIActionSheetDelegate` protocol.

2. Next, we need to create the `actionSheet` method that will handle and apply the required filter type to the image based on the button index chosen within the list.
3. Enter in the following code snippet for this method.

```

// Delegate which handles the processing of the option
// buttons selected
- (void)actionSheet:(UIActionSheet *)actionSheet
    clickedButtonAtIndex:(NSInteger)buttonIndex {
}

```

The preceding code snippet will be used to determine what button has been selected from the action sheet options panel. This is derived by the `buttonIndex` property that is passed into this function. In the next section, we will look at how to apply these image effects.

Understanding the Core Image framework

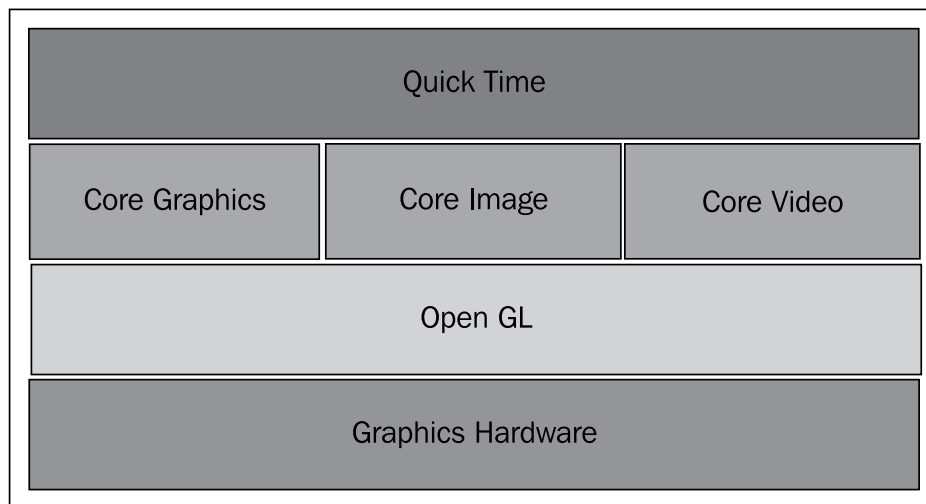
The Core Image framework is an extensible image processing technology architecture that has been built into Mac OS X v10.4 and iOS 5.0. This framework comes with over 100 built-in filters that are ready-to-use, and leverages the programmable graphics hardware to provide near real-time, pixel-accurate image processing of graphics, as well as video processing.

The Core Image filter reference describes these filters. The list of built-in filters can change. So, for this reason, Core Image provides you with the methods that let you query the system for these available filters. You can also load filters that third-party developers package as image units. The Core Image API is a part of the `QuartzCore` framework (`QuartzCore.framework`), and provides access to built-in image filters for both video and still images, and provides support for creating custom filters.

You can use Core Image from the Cocoa and Carbon frameworks by linking to this the Core Image framework. By using the Core Image framework, you can perform the following types of operations, by using filters that are bundled in Core Image or that you or another developer would create:

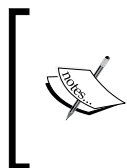
- Crop images and correct color, such as perform white point adjustment
- Apply color effects, such as sepia tone and blur or sharpen Images
- Composite images and warp or transform the geometry of an image
- Generate color, Gaussian gradients, and other pattern images
- Add transition effects to images/video
- Provide real-time color adjustment on video

The following screenshot gives you a general idea of where the Core Image framework fits in with other graphics technologies within the Mac OS X operating system.



As you can see, the Core Image framework has been integrated with these technologies, allowing you to use them together to achieve a wide range of results. You can use Core Image to process images created in Quartz 2D (Core Graphics) and textures created in OpenGL. You can also apply Core Image filters to videos played using Core Video.

The Core Image comes with over 100 built-in filters, ready-to-use by filter clients who want to support image processing in their application. The Core Image filter reference describes these filters; the list of built-in filters can change, so for that reason, Core Image provides you with the methods that let you query the system for these available filters. You can also load filters that third-party developers package as image units.



For more information on the built-in filters that are available in the Core Image API, refer to the following Apple Developer Documentation: <http://developer.apple.com/library/mac/#documentation/graphicsimaging/reference/CoreImageFilterReference/index.html>.

Applying image filter effects using the CIColor class

The CIColor class is used when you want to apply effects to images. These can be when you want to pixelate an image, or to handle red eye removal from your images. You can use the CIColor objects in conjunction with other Core Image classes, such as the CIFilter, CIContent, CIVector, and CIColor classes. In order to take advantage of the built-in Core Image filters when processing images, you can create CIColor objects with data supplied from a variety of sources, including Quartz 2D images and Core Video image buffers, using CIColorBufferRef.

The CIColor object has image data associated with it, but it is not an image. A CIColor object has all the information necessary to produce an image, but Core Image doesn't actually render an image until it is told to do so. This method allows Core Image to operate as efficiently as possible.

The `CIColor` class contains a number of parameters, which are explained in the following table:

CIColorParameters	Description
Filter category	This specifies the type of effect – blur, distortion, generator, and so on – or its intended use – still images, video, non-square pixels, and so on. A filter can be a member of more than one category.
Display name	This is the name that should be shown in the user interface.
Filter name	This is the name you use to access the filter programmatically.
Input parameters	These can contain one or more input parameters that let you control how processing is done.
Attribute class	Every input parameter that you create contains an attribute class that specifies its data type, such as <code>NSNumber</code> . An input parameter can optionally have other attributes, such as its default value, the allowable minimum and maximum values, the display name for the parameter, and any other attributes that are described in <code>CIFilter</code> .

If you take, for instance, the color monochrome filter, this contains three input parameters: the image to process, a monochrome color, and the color intensity. You supply the image and have the option to set a color and color intensity.

Most filters, including the color monochrome filter, have default values for each non-image input parameter. Core Image uses the default values to process your image, if you choose not to supply your own values for the input parameters.

Filter attributes are stored as key-value pairs. The **key** is a constant that identifies the attribute, and the **value** is the setting associated with the key. Core Image attribute values are typically one of the following data types:

- **Strings:** These are used for things, such as display names.
- **Floating-point numbers:** They are used to specify scalar values, such as intensity levels and radii.
- **Vectors:** They can have two, three, or four elements, each of which is a floating-point number. These are used to specify positions, areas, and color values.
- **Colors:** They specify color values and a color space to interpret the values in.
- **Images:** They are lightweight objects that specify images.
- **Transforms:** They specify an affine transformation to apply to `image`.
`CIColorContext`.

In the next section, we will take a look at how we can use some of these techniques when creating the various types of color effects to our `CIFilterEffects` application, when a filter type has been selected from our action sheet list of options.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder, locate the `-(void)actionSheet:(UIActionSheet *)actionSheet clickedButtonAtIndex:(NSInteger)buttonIndex` method.
2. Enter the following code snippet after the function declaration.

```
CIContext *context = [CIContext contextWithOptions:nil];
CIImage *cImage = [CIImage imageWithCGImage:[self.imageView.image
CGImage]];
CIImage *result = nil;
CIFilter *filterType = nil;
CATransition *animation = nil;
```

In the preceding code snippet, we declared a `CIContext` variable `context`. This variable will be used for rendering the `cImage` image object to the view. We then declare a `cImage` variable object of type `CIImage`, which contains a pointer to the image within our `imageView`.

Next, we declare a `CIImage` result variable that will be used to apply the image filter changes, and then output this modified `imageView` control. We then declare a `CIFilter` variable called `filterType`, which will contain the type of filter effect to use. Finally, we declare a `CATransition` variable called `animation` that will be responsible for handling the transition animations for our `UIView` layer.

3. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder.
4. Next, locate the `-(void)actionSheet:(UIActionSheet *)actionSheet clickedButtonAtIndex:(NSInteger)buttonIndex` method.
5. Enter the following code snippet after the variable declarations that we applied in the previous code snippet.

```
// Changes the overall Hue, or tint, of the source
// pixels.
if (buttonIndex == 1) {
    filterType = [CIFilterfilterWithName:@"CIHueAdjust"];
    [filterType setDefaults];
    [filterType setValue:cImage forKey: @"inputImage"];
    [filterType setValue:[NSNumber numberWithInt: 2.094]
    forKey: @"inputAngle"];

    result = [filterType valueForKey: @"outputImage"];
}
```

In the preceding code snippet, we start by declaring a `CIFilter` variable called `filterType`. This will be used to denote the type of filter that we want to apply to our image. Next, we assign a `cImage` variable of type `CIImage`, which points to the chosen image within our `UIImageView` control, and assign this to be the `inputImage`. We then assign the level of the hue to apply to the image, by setting the value of the `inputAngle` property. Once we have done all of this, we apply the hue adjustment to the image and return this to our `UIImage` result, based on the `outputImage` property, and then output this back to our `UIImageView` control.



The values for the `inputAngle` property must have a starting range from a minimum value of `-3.14` to a maximum value of `3.14`. There is also a default value of `0.00`.

Next, we will take a look at the **Vibrance** option and see what happens when this Core Image filter has been chosen from the list of options within our action sheet.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder.
2. Next, enter in the following code snippet underneath the previous code block that we applied in the previous code snippet:

```
// Adjusts the saturation of an image while keeping
// pleasing
// skin tones.
else if (buttonIndex == 2) {
    filterType = [CIFilter filterWithName:@"CIVibrance"];
    [filterType setDefaults];
    [filterType setValue: cImage forKey: @"inputImage"];
    [filterType setValue: [NSNumber numberWithIntFloat: 1.00]
        forKey: @"inputAmount"];

    result = [filterType valueForKey: @"outputImage"];
}
```

In the preceding code snippet, we start by declaring a `CIFilter` variable called `filterType`. This will be used to denote the type of filter that we want to apply to our image. Next, we assign a `cImage` variable of type `CIImage`, which points to the chosen image within our `UIImageView` control, and assign this to be `inputImage`. Finally, we assign the level of saturation to apply to the image, by setting the value of the `inputAmount` property.



The values for the `inputAmount` property must have a starting range from a minimum value of `-1.00` to a maximum value of `1.00`. There is also a default value of `0.00`.

Next, we will take a look at the **Color Invert** option, and see what happens when this Core Image filter has been chosen from the list of options within our action sheet.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder.
2. Next, enter in the following code snippet underneath the previous code block that we applied in the previous code snippet.

```
// Inverts the colors in an image
elseif(buttonIndex == 3) {
    filterType = [CIFilter filterWithName:@"CIColorInvert"];
    [filterType setDefaults];
    [filterType setValue: cImage forKey:@"inputImage"];

    result = [filterType valueForKey:@"outputImage"];
}
```

In the preceding code snippet, we start by declaring a `CIFilter` variable called `filterType`. This will be used to denote the type of filter that we want to apply to our image. Next, we assign a `cImage` variable of type `CIImage`, which points to the chosen image within our `UIImageView` control, and assign this to be `inputImage`.

Next, we need to add the code that will be used to output the updated image once this has been applied based on our Core Image filters.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder.
2. Next, enter in the following code snippet underneath the previous code block that we applied in the previous code snippet.

```
// Only process when button index is based on the
// list of options(ignore the Close button).
if (buttonIndex > 0 && buttonIndex < 4) {
    self.imageView.image =
        [UIImage imageWithCGImage:[context
            createCGImage:result
            fromRect:CGRectMake(0, 0,
                self.imageView.image.size.width,
                self.imageView.image.size.height)]];
}
```


In the preceding code snippet, we start by checking to ensure that we are not processing our **Close** button. This is a general way of safeguarding our application to prevent it from crashing. Next, we use the `imageWithCGImage` method to create and return an image object representing the specified Quartz image, then displaying this image back to our `UIImageView` control, and setting it to be displayed to the width and height of the image view. In the next section, we will take a look at how we can apply transition effects to an image while making use of the Quartz Core framework.

Applying transitions to images

Transitions are typically used to apply some sort of effect to an image. These effects are rendered over time and require that you set up a timer event. In this section, we will take a look at how to go about applying a water ripple effect to an image.

Fortunately, you don't need to worry, as there is already a ripple effect component that comes as part of the Quartz Core framework, and this will take advantage of the graphics hardware acceleration when rendering this effect.

Open the `ViewController.m` implementation file that is located within the `ExternalDisplays` folder, and add the following code statement underneath the `Color Invert` code block:

```
// Applies the Ripple Effect transition to the view.
else if (buttonIndex == 4) {
    animation = [CATransition animation];
    [animation setDelegate:self];
    [animation setDuration:3.0f];
    [animation setType:@"rippleEffect" ];
    [self.view.layer addAnimation:animation forKey:NULL];
}
```

In this code snippet, we start by declaring a variable called `animation` that will be responsible for handling the transition animations for our `UIView` layer. In the next step, we specify the duration of our ripple effect that will be used to define how long, in seconds, a single iteration of an animation will take to display. Next, we set up a timing function. This will be used to specify the `UIViewAnimationCurveEaseInOut` type as the type of animation that we want to use.

This causes the animation to start off slowly, then accelerate through the middle of its duration, and then begin to slow down towards the end of its iteration, and is the default curve for most animations. In the next step, we specify the type of animation that we want to use is the `rippleEffect` transition effect. Finally, we apply the animation effect to our view.

The following screenshot displays the output with the water rippling effect applied. You will notice how it curves from the inside out, more like a vacuum effect:



As you can see, by using both the Core Image and Quartz Core frameworks, you can create some fantastic visual effects within your applications, and bring them to life.



For more information on the Quartz Core frameworks, refer to the following URL: http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/CoreImaging/ci_intro/ci_intro.html.

For more information on the Core Image filters, please refer to the following URL:

<http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Reference/CoreImageFilterReference/index.html>.

Presenting content out to an external monitor device

In this section, we will be taking a look at how we can display an image from our iOS device out to an external monitor using the VGA port add-on for the iPad. We will look at how we can dynamically obtain a list of the available screen modes, currently supported by the external device using the `UIScreen` class. The user will then be able to choose a screen resolution from the list, and have the image resized to the selected screen mode.

Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder, and modify the `btnVGAOut` method, as shown in the following code snippet:

```
// Method to output an image to an external screen.
- (IBAction)btnVGAOutput:(id)sender
{
    // Determine if we have found any external screens.
    if ([[UIScreen screens] count] > 1) {
        externalScreen = [[UIScreen screens] objectAtIndex:1];
        screenModes = externalScreen.availableModes;
        UIAlertView *alertView = [[UIAlertView alloc]
            initWithTitle:@"External Screen Size"
            message:@"Choose a screen resolution"
            delegate:self cancelButtonTitle:nil
            otherButtonTitles:nil];

        // Fill our alertView with the available screen modes
        for (UIScreenMode *mode in screenModes) {
            CGSize modeScreenSize = mode.size;
            [alertView addButtonWithTitle:[NSString
                stringWithFormat:@"%0.0f x %0.0f pixels",
                modeScreenSize.width, modeScreenSize.height]];
        }
        // Display the alert dialog to the view.
        [alertView show];
    }
    else
    {
        // We didn't manage to locate any external devices.
        NSLog(@"Unable to locate any external screens.");
    }
}
```

In this code snippet, we use the `screens` property of the `UIScreen` class, then use the `count` property to determine if any external devices have been detected, and then set the `objectAtIndex:1` property, this means that we are using an external screen. Next, we populate our `NSArray` variable `screenModes` with the list of available screen modes that are supported by the external device. Finally, we create an `alertView` dialog and dynamically populate it with all of the determined available screen modes, before finally displaying the alert dialog to the screen.

Alternatively, if no external devices were found, we log out an error message to the console window.

Next, we need to add the code that will be used to output the image to the external device and resize it based on the screen resolutions selected.

1. Open the `ViewController.m` implementation file, located within the `ExternalDisplays` folder.
2. Next, enter in the following code:

```
// Handles displaying of the chosen screen mode to the
// external display.
- (void)alertView:(UIAlertView *)alertView
  clickedButtonAtIndex:(NSInteger)buttonIndex
{
    // Get the chosen screen mode from the list.
    UIScreenMode *desiredMode = [screenModes
    objectAtIndex:buttonIndex];

    // Create a new window instance for our external
    //screen.
    externalWindow = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];

    externalScreen.currentMode = desiredMode;

    // Create a new imageView control
    UIImageView *externalImage = [[UIImageView alloc]
    initWithFrame:[UIScreen mainScreen]
    applicationFrame]];

    // Copy the image from our view to the externalWindow
    [externalImage setImage:imageView.self.image];
    [externalWindow addSubview:externalImage];
    [externalWindow setScreen:externalScreen];

    // Resize the external window to the size of the
    // chosen screen resolution.
```

```
CGRectrect= CGRectZero;
rect.size = desiredMode.size;
[externalWindow setFrame:rect];
[externalWindow setClipsToBounds:YES];
[externalWindow setHidden:NO];
[externalWindow makeKeyAndVisible];

// Release the memory allocated to the objects
screenModes = nil;
externalScreen = nil;
}
```

In the preceding code snippet, we determine what button has been selected from the `alertView` options panel. This is derived by the `buttonIndex` property that is passed into this function. Next, we create a new window instance for our external screen by declaring the `externalWindow` variable object, and then set the current mode for the external window based on the screen resolution chosen.

3. Finally, we create a new `imageView` control, and set the image of the current view to the external window, before resizing the external window size to the size of the chosen screen resolution. We then make this window visible to the user, so that it shows up on the monitor screen. At the end we dereference our `screenModes` and `externalScreen` objects.

Implementing the `shouldAutorotateToInterfaceOrientation:` method

Now, we need to implement a method that will be responsible for preventing our device from having our application being displayed within the various views.

1. Open the `ViewController.m` implementation file, and then locate the `shouldAutorotateToInterfaceOrientation:` method.
2. Next, enter in the following code snippet:

```
- (BOOL) shouldAutorotateToInterfaceOrientation: (UIInterfaceOrientation) interfaceOrientation
{
    // Forces the device to stay in Portrait mode.
    return (interfaceOrientation ==
            UIInterfaceOrientationPortrait ||
            interfaceOrientation ==
            UIInterfaceOrientationPortraitUpsideDown);
}
```

In the preceding code snippet, we force the device to always display in portrait mode when the device has been rotated, then check the `interfaceOrientation` variable of the iOS device, and set this to the value of `UIInterfaceOrientationPortrait`.

Finishing up

Congratulations, we have finally implemented the methods for our `ExternalDisplays` application. Next, build and run the application by choosing **Product | Run** from the **Product** menu, or alternatively press *Command + R*. The following screenshot shows the application running on the iOS device:



In the preceding screenshot, the external screen displays an image, which has been chosen from the iOS device's photo library. Once the user taps on the **VGA Out** button, we are presented with a list of available screen resolutions that have been detected as being supported for the type of VGA monitor. Once a screen mode has been selected, you can see that the image is sent to the monitor, and resized accordingly.

Summary

In this chapter, we learned about the AirPlay and Core Image frameworks, and looked at how we can implement these into our applications to output this to an external device, such as Apple TV and an external monitor using the VGA Adaptor.

We then learned about the Core Image filters class, and how we can apply the different image filter effects to enhance images through the different built-in filters, such as color effects. We then familiarized ourselves with the Quartz Core framework, and looked at how we can use this framework, using the built-in filters, for distortions and transition effects to apply a water ripple effect to an image.

In the next chapter, we will learn about iCloud and the storage APIs that come as part of this technology. We will take a look at building an iCloud application, and understand the different methods that can be used to store and use documents within the Cloud. To end the chapter, we will look at the methods that we can use to detect and handle file-version conflicts.

10

Storing Documents within the Cloud

The *ScratchPad* application allows you to keep a visual record of each note, or reminder for your every day needs. These can be as simple as jotting down some ideas when you are on your way home on the train, creating an itemized list of shopping items, or even creating your daily journal entries. This application records information for each item, and then adds this information into your iCloud account repository, using the iCloud storage APIs.

In this chapter, we will take a look at the features of iCloud and the storage APIs, and see how we can incorporate these into our application, so that it can interact with the iCloud servers to read, write, and edit documents, and provide us with the ability to access these items from all our iOS devices, without the need of having to sync or transfer these files.

Storing documents within a user's iCloud account provides us with an additional layer of security, so even if the user loses their device, these documents can easily be retrieved, provided that they are contained within iCloud storage.

In this chapter we will:

- Build the *ScratchPad* application using Storyboards
- Learn how to store and search for documents within iCloud Storage
- Learn how to handle file-version conflicts
- Learn how to configure and set up provisioning profiles ready for iCloud
- Implement the functionality to add or edit an item in `UITableView`
- Implement the methods to save and cancel items

We have an exciting project ahead of us, so let's get started.

Overview of the technologies

iCloud is a service that comes as part of iOS 5 and helps you synchronize your data across multiple devices, by using a set of central servers that store your documents, apps, music, and photos, while making the latest version of these available to every device compatible with iCloud, including your Mac or PC.

In iOS, each application has its data stored within a local directory, and each application can access data within its own directory, thus preventing other applications from reading or modifying data from other applications. iCloud allows you to upload your local data to its central servers, and receive updates from other devices. This replication of content is achieved by a continuous background process, called a *daemon*, which detects changes made to a resource: document, photo, and so on, and then uploads them to the central storage repository.

In this chapter, we will learn how to implement a background monitoring process called "notifications" that will enable us to keep track of documents within the Cloud repository, and refresh the table view controller as new documents are added or updated. This is particularly useful if a user has deleted a document manually from their iCloud repository, and we need to ensure that our application correctly shows what is currently in the cloud.

Methods to store and use documents within iCloud

iCloud provides you with a common central location for easy access to each document that gets stored within the Cloud, any updates made to the document will be delivered to each iOS device or computer, as long as they are using the same Apple ID used to upload those documents. When a document is uploaded to iCloud, it is not moved there immediately, as the document must first be moved out of the application sandbox into a local system-managed directory, where it can be monitored by the iCloud service.

Once this process has completed, the file is transferred to iCloud and then distributed out to each of the user's iOS devices. Any changes made on one device are initially stored locally and then immediately pushed out to iCloud, using a local daemon service.

This daemon service is used to prevent file conflicts from happening at the same time, and is handled by the file coordinator, which mediates changes made between the application and the local daemon service, responsible for facilitating the transfer of the document to-and-from the iCloud service.

The file coordinator acts much like a locking mechanism for the document, thus preventing your application and the daemon service from applying modifications to the document simultaneously. At the heart of the iCloud locking mechanism are file coordinators and file presenters, which are explained in the following sections.

The file coordinator

Whenever you need to read and write a file, you do so by using a **file coordinator**, which is an instance of the `NSFileCoordinator` class. The job of a file coordinator is to coordinate the reads and writes performed by your application and the sync daemon on the same document. For example, your application and the daemon may both read the document at the same time, but only one may write to the file at any single time.

Also, if one process is reading the document, the other process is prevented from writing to the document, until the earlier process is finished reading the document.

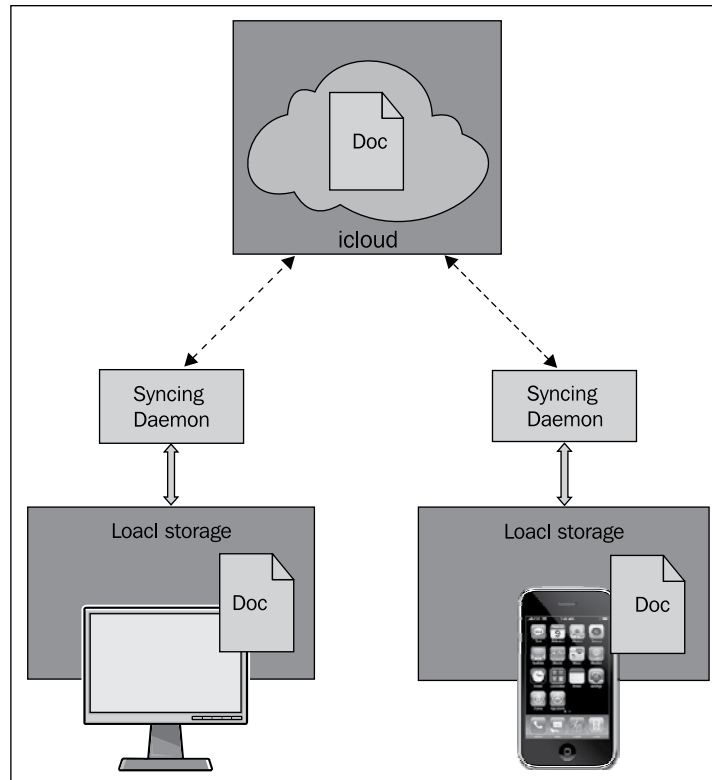
The file presenter

In addition to coordinating operations, file coordinators also work with file presenters to notify applications when changes are about to occur. A **file presenter** is any object that conforms to the `NSFilePresenter` protocol, and takes responsibility for managing a specific file (or directory of files) in an application.

The job of a file presenter is to protect the integrity of its own data structures. It does this by listening for messages from other file coordinators and using those messages to update its internal data structures. In most cases, a file presenter may not have to do anything.

However, if a file coordinator declares that it is about to move a file to a new URL, the file presenter would need to replace its old URL, with the new one provided to it by the file coordinator.

The following screenshot shows the process when changes are made on one device, and having those changes stored locally before being pushed back out to the iCloud service, using a local daemon process:



Whenever your application stores documents to iCloud, it must specify one or more containers in which those documents' contents will be stored, by including the `com.apple.developer.ubiquity-container-identifiers` key value entry within your application's entitlements file. This is covered in the section *Requesting entitlements for iCloud storage* in this chapter.

Using the iCloud storage APIs

The iCloud storage APIs let your application write user documents and data to a central location, and access those items from all of a user's computers and iOS devices. Storing documents in a user's iCloud account provides a layer of security for that user. If the user happens to lose their device, any documents that were saved on it can easily be recovered, if they are contained within iCloud storage. To fully utilize

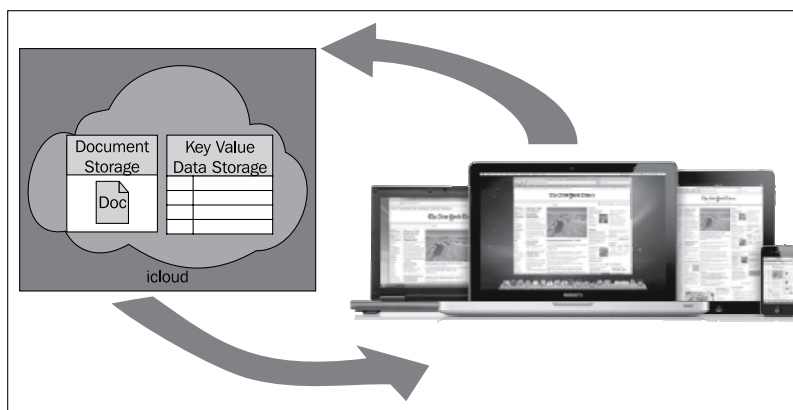
iCloud storage, this can happen in one of the two ways, which information can be accessed. These are explained in the following table:

Storage type	Description
iCloud document storage	Use this feature to store and share user documents and data in the user's iCloud account
iCloud key-value data storage	Use this feature to store and share small amounts of data among instances of your application

Most applications that you create will use the iCloud document storage to share documents from a user's iCloud account as this provides you with the ability to share documents across multiple devices, and manage documents from a given device.

When using the iCloud key-value data store, this is not something that a user will see, as this is handled by your application to share very small amounts of information that are only used by your application. An example of this would be that you can store the time the user logged in to your application, or what screen they are currently viewing.

The following screenshot shows the process involved when creating information in local iCloud storage within your application's sandbox:



For more information about iCloud, you can refer to the Apple Developer Documentation at the following URL: https://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iCloud/iCloud.html#//apple_ref/doc/uid/TP40007072-CH5-SW1.

Handling iCloud file-version conflicts

Handling version conflicts of files is a common issue in software development. With iCloud, we need to be able to handle this when multiple instances of your application are running on multiple devices, and both try to modify the same document.

This will result in a conflict when both devices try to upload the changes made to the file at the same time. At this point, iCloud will end up with two different versions of the same file, and has to decide what to do with them.

The solution to this is to make the most recently modified file the current file, and to mark any other versions of the file as conflict versions. To avoid loss of changes made to those documents, your application will need to prompt the user to choose the appropriate course of action to take.

For example, you might let the user choose which version of the file to keep, or you might offer to save the older version under a new name. You would need to determine the current file's version, using the `currentVersionOfItemAtURL:` class method, and obtain an array of the conflicted versions, by using the class method call to `unresolvedConflictVersionsOfItemAtURL:`.

For each conflicted file version, you will need to perform the appropriate course of action to resolve the conflict, by using any of these actions, listed as follows:

- You have the option to merge the conflicted versions with the current file automatically, if it is practical to do so.
- You can choose to ignore the conflicted versions, which will result in data being lost in those files.
- You can have your application prompt the user to select the appropriate course of action, and let the user decide which of the versions they should keep. Please keep in mind that this should always be your last course of action.

Building the ScratchPad application

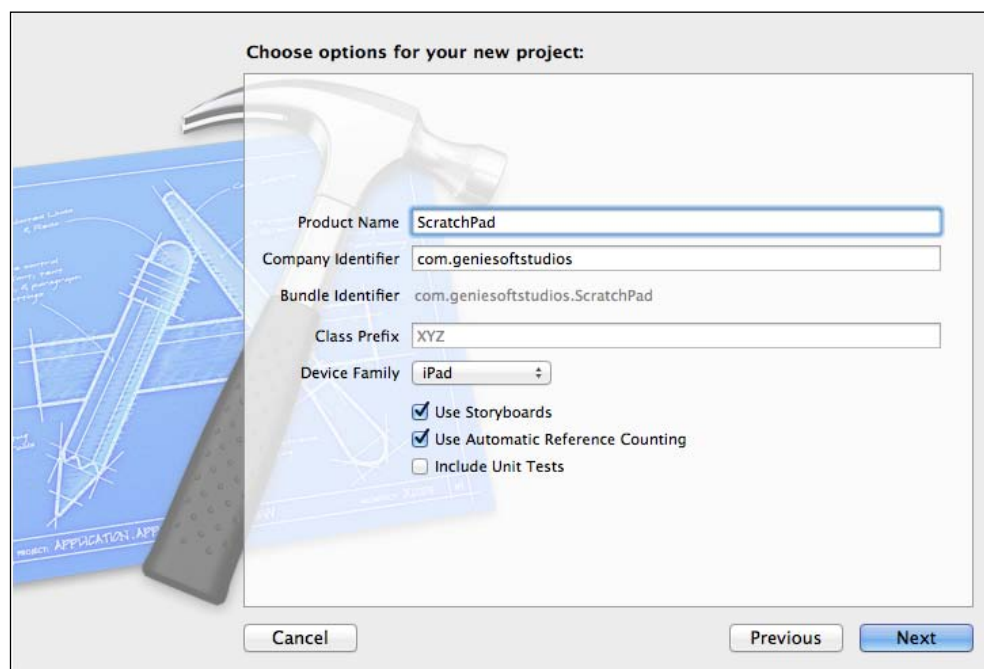
The ability to create notes to remind us of what needs to be done is very common, and forms part of our day-to-day duties. These can be as simple as jotting down some ideas, when you are on your way home on the train, creating an itemized list of shopping items, or even creating your daily journal entries.

In this section, we will take a look at how to create an application that will enable us to create new documents within our iCloud repository, and then have the ability for us to edit this information, and save it back to our iCloud repository. We will also be using the `UITableView` control to populate it with our documents retrieved from our repository.

Before we can proceed, we first need to create our `ScratchPad` project. To refresh your memory on how to go about creating a new project, you can refer to the section that we covered in *Chapter 2, TaskPriorities – Building a TaskPriorities iOS App*, under the section named *Building the TaskPriorities App*.

It is very simple to create this in Xcode. Just follow the steps listed here:

1. Launch Xcode from the `/Xcode4/Applications` folder.
2. Choose **Create a new Xcode Project**, or select **File | New Project**.
3. Select the **Single View Application** template from the list of available templates.
4. Click on the **Next** button to proceed with the next step in the wizard.
5. Next, enter in `ScratchPad` as the name for your project.
6. Select **iPad** from under the **Device Family** drop-down list.
7. Ensure that the **Use Storyboards** checkbox has been selected.
8. Ensure that the **Use Automatic Reference Counting** checkbox has been selected.
9. Ensure that the **Include Unit Tests** checkbox has not been selected.
10. Click on the **Next** button to proceed with the next step in the wizard:



11. Next, specify the location where you would like to save your project.
12. Then, click on the **Create** button to continue and display the Xcode workspace environment.

Creating the main application screen

Now that we have created our `ScratchPad` project, we can start building our user interface that will be responsible for allowing us to create and modify new or existing documents directly into our list and have them stored within the Cloud.

These screens will consist of a Tab Bar controller, Navigation controller, and View controllers. The Navigation controller enables us to create relationships between the other screens within the Storyboard and set up the required connections, known as segues. A segue represents a transition from one screen to another.

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Select the View controller that was added by the template, and then delete it.
3. From **Object Library**, select-and-drag a (UITabBarController) **Tab Bar Controller** control, and add this to our view.

To see how to go about adding `UITabBarController`, **Tab Bar Controller**, you can refer to *Chapter 4, Enhanced Address Book App*, under the section named *Creating the main application screen*.

Adding the table control to hold iCloud document data

Our next step is to add a `UITableViewController` object that will be used to hold and list our task entries. We will need to include a Navigation controller that will be used to navigate back and forth between `UITableViewController` and itself. To see how to go about adding `UITableViewController`, you can refer to *Chapter 4, Enhanced Address Book App*, under the section named *Adding the table control to hold item data*.

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (`UITableViewController`) **Table View Controller** control, and add this to our Storyboard canvas.
3. Next, delete the two `UIViewController` objects that Xcode generated in the Storyboard when you dragged-and-dropped the `UITabBarController` controller.

Next, we need to create a navigation controller between `TabBarController` and `UITableViewController` that we just added. There are two ways that this can be achieved; you can either drag a `UINavigationController` object directly onto the view, or you can let Xcode do this for you automatically:

1. Select the `UITableViewController` object that we just added, and then choose **Editor | Embed In | Navigation Controller** from the **Editor** menu.
2. Next, select `TabBarController`, then hold down *Ctrl*, drag from the Tab Bar controller to the Navigation controller, and release the mouse.
3. Choose **Relationship - viewControllers** from the Storyboard Segues pop-up window.
4. Next, we want to show the bottom toolbar within our Navigation controller. Select the Navigation controller, and from the **Attributes Inspector** dialog box, select the **Shows Toolbar** option.

So far, we have linked up our Tab Bar controller and Navigation controllers, and have configured the properties required for the Navigation controller; our next step is to set up the properties on our Table View controller. Follow these simple steps:

1. Select **Table View Controller** that we just added previously.
2. Next, click on the **Prototype** cell from the **Prototype Cells** section.
3. From the **Attributes Inspector** section, and change the **Style** property to **Subtitle**. This will change the cell's appearance to contain two labels.

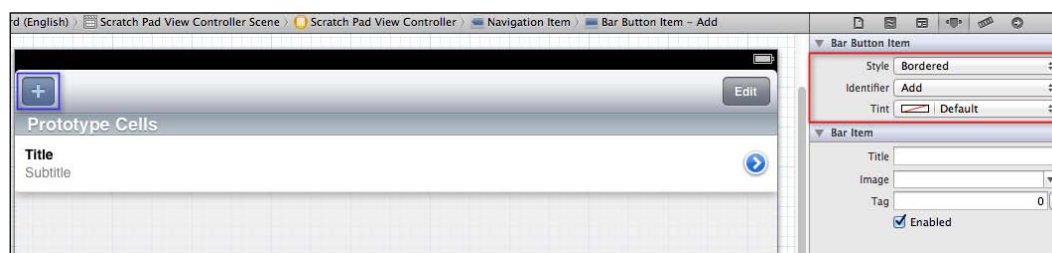
You would have noticed that since we added our Table View controller, Xcode gave us a warning. This generally happens whenever you add a new Table View controller to any new or existing storyboards; and is due to it wanting to use prototype cells as the default. Now, we look at how we can properly configure this:

1. Select the **Identifier** item and enter in `ScratchPadCell` as its unique identifier. You will notice that once this has been entered in, Xcode will stop complaining about the warning message we received earlier on.
2. Set the **Accessory** attribute to show **Detail Disclosure**.
3. Now that we have set up the properties to our Table View controller, it would be a good time to build and run our application to ensure that you have followed the steps correctly, and no program errors exist.
4. Choose **Product | Run** from the **Product** menu, or alternatively press *Command + R*.

Adding the Add button

Our next step is to add a button to our Navigation controller; this will be responsible for displaying an additional screen, providing us with the ability to create new documents that will be added to our iCloud repository. This information will then be displayed within our Table View control. This can be achieved by following these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control to the top left-hand corner of the navigation bar on the **ScratchPad** (UITableViewController) **Table View Controller** screen.
3. From the **Attributes Inspector** section, change the **Identifier** property to **Add**.
4. Then, change the **Style** property to **Bordered**:



Now that we have added our **Add** button to our **Scratch Pad View Controller**, our next step is to add the **Edit** button that will be responsible for editing an existing document within our table view when the button is clicked. So let's proceed with the next section.

Adding the Edit button

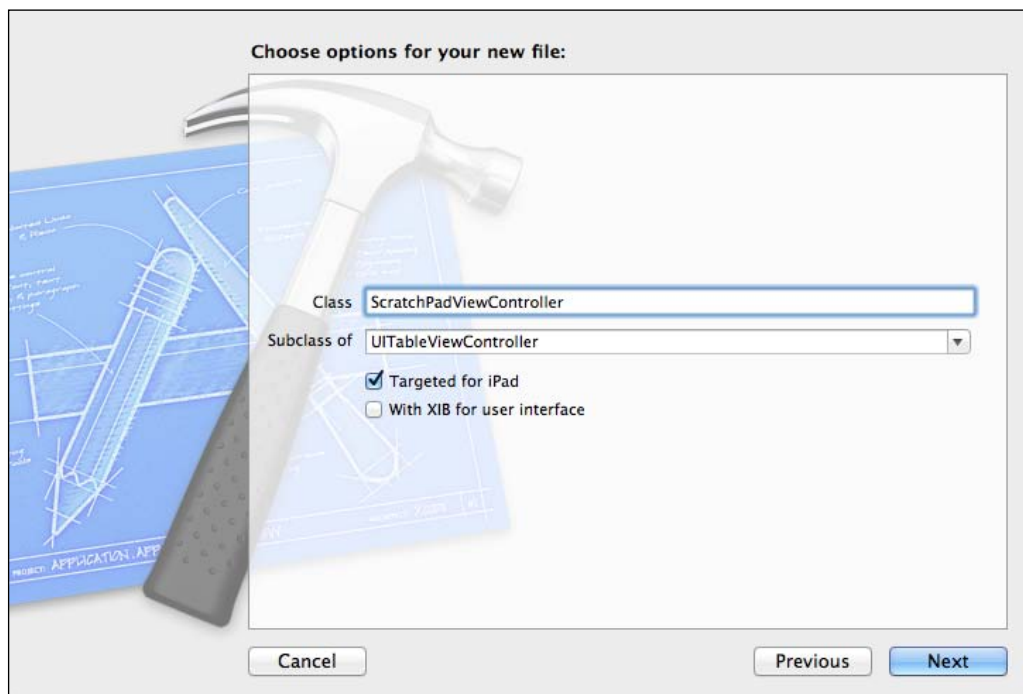
Now that we have added our button to add a new document, our next step is to add another button that will be responsible for allowing the user to make changes to an existing iCloud document record within the table view. This can be achieved by following these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control to the top right-hand corner of the navigation bar on the **Scratch Pad Table View Controller** screen that we added previously.

3. From the **Attributes Inspector** section, change the **Identifier** property to **Edit**.
4. Then, change the **Style** property to **Bordered**.

Now that we have added our **Add** and **Edit** buttons and have properly configured our Table View controller, as well as built our user interface, the next step is to create our very own custom `UIViewController` subclass. This will act as the data source for our table, so that it will know how many rows to display when it retrieves each document from our iCloud repository.

1. Select the `ScratchPad` folder, choose **File | New | File...** or press *Command + N*.
2. Select **Cocoa Touch**, located under the iOS header section.
3. Select the **Objective-C** class template from the list of templates.
4. Click on the **Next** button to proceed with the next step within the wizard.
5. Enter `ScratchPadViewController` within the **Class** field as the name of the file to be created.
6. Ensure that you have selected `UITableViewController` as the type of subclass to be created from the **Subclass of** dropdown.
7. Ensure that you have selected the **Targeted for iPad** option:



8. Click on the **Next** button to proceed with the next step of the wizard.
9. Click on the **Create** button to save the file to the folder location specified.

Now that we have added our View controller class to our ScratchPad application, our next step is to update the class of our UITableViewController to use this class, instead of the default UITableViewController class:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Click-and-select our ScratchPad (UITableViewController) controller.
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `ScratchPadViewController`.

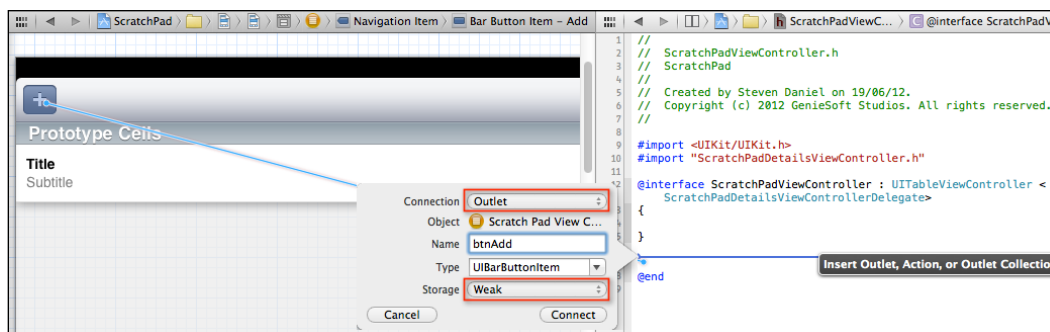
The next step is to create the outlets for our **Add** and **Edit** buttons:

1. Open the **Assistant Editor** window by choosing **Navigate | Open In Assistant Editor**, or press *Option + Command + ,*.
2. Ensure that the `ScratchPadViewController.h` interface file gets displayed.
3. Select the **Add** (UIBarButtonItem) control, hold down the *Control* key, and drag it into the `ScratchPadViewController.h` interface file.



In order to create the IBOutlet properties, it is necessary to drop it outside the curly braces { } of the @interface; as these braces are not there by default, you will need to add them.

4. Enter in `btnAdd` for the name of the property to be created.
5. Choose **Weak** from the **Storage** drop-down list:



6. Repeat steps 6 to 8 to create the outlets for the `btnEdit` button.

Now that we have created the outlet properties, our next step is to create the `NSMutableArray` and `NSMetadataQuery` properties within our `ScratchPadViewController` interface file. This will allow us to look up the document within our application's iCloud repository, and keep a track of each document object properties that have been retrieved.

1. Open the `ScratchPadViewController.h` interface file from within **Project Navigator**, and enter the following highlighted code sections:

```
// ScratchPadViewController.h
// ScratchPad
// Created by Steven Daniel on 19/06/12.
// Copyright (c) 2012 GenieSoft Studios. All rights
// reserved.

#import <UIKit/UIKit.h>
#import "ScratchPadDetailsViewController.h"

@interface ScratchPadViewController :
    UITableViewController<
    ScratchPadDetailsViewControllerDelegate>
{
}

// Declare the Getters and Setters for all objects.
@property (nonatomic,strong) NSMutableArray
    *document;
@property (nonatomic,strong) NSMetadataQuery
    *docQuery;
@property (nonatomic,weak) IBOutlet UIBarButtonItem *btnAdd;
@property (nonatomic,weak)
    IBOutlet UIBarButtonItem*btnEdit;

// Declare each of our class methods.
- (void)getScratchPadDetails;
- (void)getScratchPadData: (NSMetadataQuery *)query;

@end
```

As you can see, in the preceding code, we start by extending our class to include our `ScratchPadDetailsViewController` class protocol delegate, so that we can access the methods that will be used for adding and editing document information. Next, we declare an `NSMutableArray` object to hold each document that is created, so that it can be used as a data source to populate our table view control.

In our final step, we create an `NSMetadataQuery` object that will be used to query and look up the document within our applications iCloud repository, so that it can access the relevant object properties of each file.



For more information about the `NSMutableArray` object, refer to the following URL: http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSMutableArray_Class/Reference/Reference.html.

For more information about the `NSMetadataQuery` object, refer to the following URL: http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSMetadataQuery_Class/Reference/Reference.html.

2. Open the `ScratchPadViewController.m` implementation file from within **Project Navigator**, and enter the following highlighted code sections:

```
// ScratchPadViewController.m
// ScratchPad
// Created by Steven Daniel on 19/06/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "ScratchPadViewController.h"

@interface ScratchPadViewController ()

@end

@implementation ScratchPadViewController

@synthesize document;
@synthesize docQuery;
@synthesize btnAdd = m_btnAdd;
@synthesize btnEdit = m_btnEdit;
```

3. Next, we need to change the table view data source methods that are located within the `ScratchPadViewController.m` implementation file, and enter the following highlighted code snippets:

```
#pragma mark - Table view data source
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections. return 1;
}
```

```

- (NSInteger)tableView:(UITableView
*)tableView numberOfRowsInSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return [self.document count];
}

```

As you can see from the preceding code snippets, we set the number of table sections, and then have the `numberOfRowsInSection` method work out how many rows will exist in each section. This is achieved by using the `count` property of our document array object:

```

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

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:
        @"ScratchPadCell"];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:@"ScratchPadCell"];
    }

    ScratchPad *scratchDoc = [self.document
        objectAtIndex:indexPath.row];

    cell.textLabel.text =
        scratchDoc.fileURL.lastPathComponent;
    cell.detailTextLabel.text = scratchDoc.docContent;

    // Configure the cell...
    return cell;
}

```

Finally, as you can see in the preceding code snippet, we supply the reuse identifier of the `TableViewCell` cell that we set up previously, and then obtain the document properties for each item retrieved from our iCloud repository, including the name of the file and the associated document content, then write these out to each of our cell labels.



When you reference the reuse identifier as a parameter to the method called `dequeueReusableCellWithIdentifier`, it automatically makes a new copy of the prototype, and returns the object back to you.

```
#pragma mark - Table view delegate
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Enable our Edit Button when an item has
    // been selected within our Table View.
    if (indexPath > 0) {
        self.btnEdit.enabled = YES;
    }
    else {
        self.btnEdit.enabled = NO;
    }
}
```

As you can see from the preceding code snippet, we check to see if the `indexPath` property for the selected row is greater than zero. If this is `TRUE`, we enable the **Edit** button within our table view controller. Alternatively, we disable the button.

4. Next, we need to modify the `viewDidLoad` method, located within the `ScratchPadViewController.m` implementation file, and enter the following highlighted code snippets:

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

    // Initialize our table view buttons
    self.btnAdd.enabled = YES;
    self.btnEdit.enabled = NO;

    NSURL *ubiq = [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil];

    if (!ubiq) {
        self.btnAdd.enabled = NO;
        self.btnEdit.enabled = NO;
        NSLog(@"Error connecting to iCloud Service.");
    }
}
```

```

// Initialize and reload our document information.
self.navigationController.navigationBar.tintColor =
    [UIColor redColor];
self.title = @"ScratchPad for iCloud";
self.document = [[NSMutableArray alloc] init];

// Add an observer call to reload the list when
// the application becomes active from the
// background.
[[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(getScratchPadDetails)
 name:UIApplicationDidBecomeActiveNotification
 object:nil];
}

```

In the preceding code snippet, we start by checking for the availability of iCloud as soon as the application starts. Although iCloud is available on all iOS 5 devices, it is a great way to ensure that the user has configured this correctly, and prevents your application from crashing. The `URLForUbiquityContainerIdentifier` method takes the container identifier and returns the URL for the iCloud storage on your local iOS device.

Passing in `nil` to the method, automatically returns the first iCloud container set up for the project. We will look at setting this up in the section *Requesting Entitlements for iCloud Storage*, later on in this chapter. If we can access the iCloud repository, we enable/disable our **Add** and **Edit** buttons accordingly, then initialize and set the color of our navigational controller bar, set the title, and initialize our document `NSMutableArray` object.

Finally, we set up an observer call to listen when our application becomes active, and reload the documents from our iCloud storage repository. This is particularly useful, and prevents your application from crashing if the user manually deletes the document record from their iCloud repository. You still need to ensure that your application performs, and refresh the list to represent what is in the application's iCloud repository.

5. Next, open the `ScratchPadViewController.m` implementation file, and enter the following code snippet for the `getScratchPadDetails` method:

```

- (void) getScratchPadDetails
{
    NSURL *ubiq = [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil];

    if (ubiq)

```



```
{
    self.docQuery= [[NSMetadataQuery alloc] init];
    [self.docQuery setSearchScopes:[NSArray
        arrayWithObject:
            NSMetadataQueryUbiquitousDocumentsScope]];
    NSPredicate *pred = [NSPredicate predicateWithFormat:
        @"%K Like 'Jou*_*'", NSMetadataItemFSNameKey];

    [self.docQuery setPredicate:pred];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(queryDidFinishGathering:)
        name:NSMetadataQueryDidFinishGatheringNotification
        object:self.docQuery];

    [self.docQuery startQuery];
} else {
    NSLog(@"Problem connecting to the iCloud Service.");
}
}
```

In the preceding code snippet, we start by ensuring that we can connect to our iCloud data store, and then set up and initialize our `docQuery` query predicate to look for all occurrences of file names starting with "Jou*_*" using the predicate class method `NSMetadataQueryUbiquitousDocumentsScope`. We then set up an observer `queryDidFinishGathering` notification that gets called when the metadata search finishes gathering all items.

6. Next, open the `ScratchPadViewController.m` implementation file, and enter the following code snippet for the `queryDidFinishGathering:` notification method, as follows:

```
- (void)queryDidFinishGathering:(NSNotification
*)notification {
    NSMetadataQuery *query = [notification object];

    [query disableUpdates];
    [query stopQuery];
    [self getScratchPadData:query];
    [[NSNotificationCenter defaultCenter]
        removeObserver:self
        name:NSMetadataQueryDidFinishGatheringNotification
        object:query];

    self.docQuery= nil;
}
```

In the preceding code snippet, this method is called once the predicate class method, `NSMetadataQueryUbiquitousDocumentsScope:`, has completed gathering all items from the query, as defined within our `getScratchPadDetails` method. We then call the `getScratchPadData:query` method, and extract the required properties associated with the document, before finally removing the notification observer call from the notification center.

7. Next, open the `ScratchPadViewController.m` implementation file, and enter the following code snippet for the `getScratchPadData:query` method:

```
// Populate our documents array with the contents
// for each document returned.
- (void)getScratchPadData:(NSMetadataQuery *)query {

    [self.document removeAllObjects];

    for (NSMetadataItem *item in [query results])
    {
        NSURL *url = [item
            valueForKey:NSMetadataItemURLKey];
        ScratchPad *contents = [[ScratchPad alloc]
            initWithFileURL:url];
        [contents openWithCompletionHandler:^(BOOL success) {
            if (success) {
                [self.document addObject:contents];
                [self.tableView reloadData];
            }
            else
            {
                NSLog(@"failed to open from iCloud");
            }
        }]];
    }
}
```

In the preceding code snippet, we cycle through each item returned by our results query, populate our document array, and then reload the list of documents that have been returned by iCloud. If any issues are experienced accessing the iCloud document repository, an error message will be logged out to the console window.

8. Next, open the `AppDelegate.m` implementation file from within **Project Navigator**, and modify the `didFinishLaunchingWithOptions:` method, as shown in the following code snippet:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary
    *)launchOptions
{
    // Override point for customization after the
    //application launches.
    return YES;
}
```

When using Storyboards, we don't need to create a new `UIWindow` instance, as this will create another white window and place it on top of the Storyboard. So, we just need to clear out everything except the `return YES` statement.

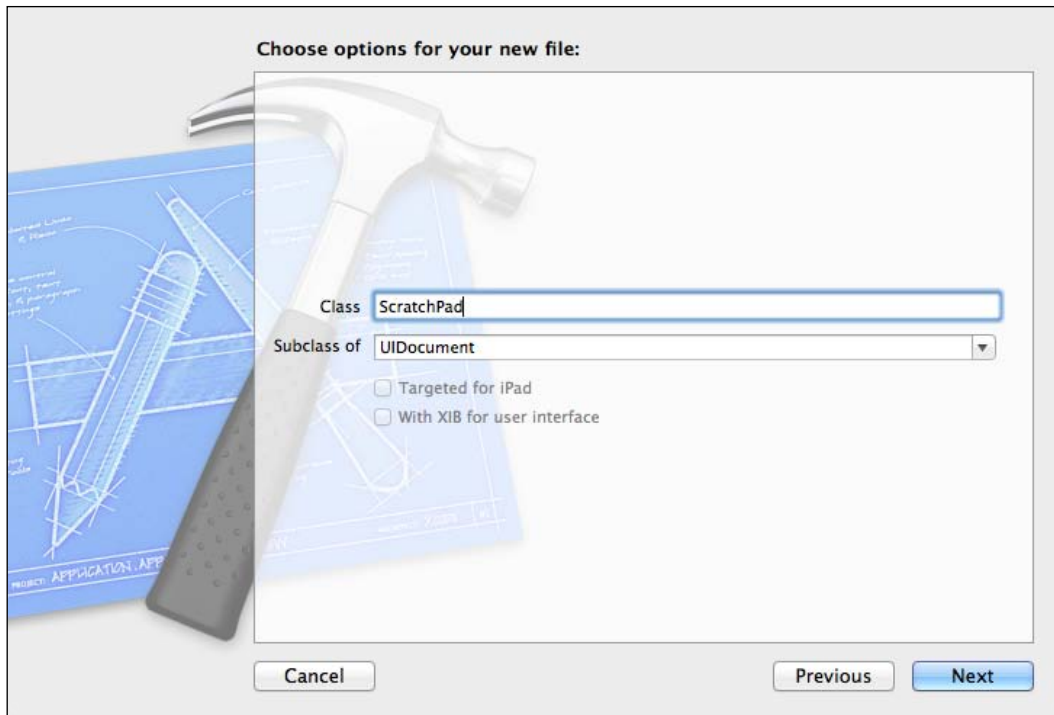
In the preceding section, we saw how to successfully configure our data source, so that it can be used to populate our table view with each document that is retrieved from our application's iCloud repository. We also learned how we can use predicates to search and retrieve filenames using the `Like` statement, as well as how to set up observers to call methods.

Our next step is to use the `UIDocument` class that comes with iOS 5, which will make it much easier for us when working with iCloud documents. This is because it acts as the middleware between the file and the actual data that it contains, and implements the `NSFilePresenter` protocol to handle our entire document processing in the background, so that our application is not blocked when files are opened or saved.

1. From the `ScratchPad` folder, choose **File | New | New File...**, or press *Command + N*.
2. Select the **Objective-C class** template from the list of templates.
3. Click on the **Next** button to proceed with the next step in the wizard.
4. Enter in `ScratchPad` as the name of the file to be created.
5. Ensure that you have selected `UIDocument` as the type to create from the **Subclass of** dropdown.



If you don't see the `UIDocument` class shown within your drop-down list, simply type it in manually.



6. Click on the **Next** button to proceed with the next step of the wizard.
7. Click on the **Create** button to save the file to the folder location specified.
8. Next, open the `ScratchPad.h` interface file, located within the `ScratchPad` folder. Enter the following code snippet:

```
// ScratchPad.h
// ScratchPad
// Created by Steven Daniel on 19/06/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.
```

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

```
@interface ScratchPad : UIDocument
```

```
@property (nonatomic, strong) NSString *docContent;
```

```
@end
```

In the preceding code snippet, we declared an `NSString` property variable `docContent` that will be used to store the contents of each document that gets created or modified.

9. Next, open the `ScratchPad.m` implementation file from within **Project Navigator**, and enter the following code snippet:

```
// ScratchPad.m
// ScratchPad
// Created by Steven Daniel on 19/06/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import "ScratchPad.h"

@implementation ScratchPad

@synthesize docContent;

// Called whenever the application reads data from the
// file.
- (BOOL)loadFromContents:(id)contents ofType:(NSString
*)typeName error:(NSError **)outError
{
    // Initialize our document content
    self.docContent = @"";

    // Check to see if we have any text associated
    // for the document.
    if ([contents length] > 0) {
        self.docContent = [[NSString alloc]
initWithBytes:[contents bytes]
length:[contents length]
encoding:NSUTF8StringEncoding];
    }

    return YES;
}
```

As you can see in the preceding code, we start by synthesizing our document content property, so that our class can access the objects associated with it. Next, we override the `loadFromContents:` method to read the data from the file into our `UIDocument` subclass. The most important parameter to note here is `contents`; this is an `NSData` object containing the actual data that was entered when you created or updated your document model. The background queue of `NSFilePresenter` calls this method whenever the read operation has completed. If the document was saved without entering any information, we assign a default value of `""`:

```
// Called whenever the application saves the content.
- (id)contentsForType:(NSString *)typeName error:(NSError
```

```

        **)outError
    {
        // Check to ensure we have content to save
        // for our document.
        if ([self.docContent length] == 0) {
            self.docContent = @"";
        }

        // Save the document contents and return back the data.
        return [NSData dataWithBytes:[self.docContent
            UTF8String]
            length:[self.docContent length]];
    }

@end

```

As you can see in the preceding code, we override the `contentsForType:` method, which is used when the background queue of `NSFilePresenter` requests a snapshot of the contents of the `UIDocument` subclass. Here we check to ensure that the document contains contents, and if so, we convert our document's data to an `NSData` object, and return this as an `NSData` instance.

In our next section, we will see how we can navigate between screens within the Storyboard. We will learn about segue, and the different types of views they can take on, as well as how to go about providing the ability for additional documents to the current list of documents within our table view.

Navigating between screens using Storyboards

In this section, we will be adding more View controllers to our Storyboard to allow the flexibility of creating and modifying documents within our existing table view.

In order for us to transition between screens within our Storyboard, we need to create a connection, known as a segue. Segues are defined as having the ability to only go one way; they cannot go back to the previous screen, unless a delegate class has been set up. For our new screen, we will be creating a "modal" segue. To begin creating the **Add New Document** screen, follow these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. From **Object Library**, select-and-drag a new (`UIViewController`) **View Controller** control, and add it to our Storyboard to the right of the **Table View** screen.

3. Next, select the `UIViewController` control that we just added, and then choose **Editor | Embed In | Navigation Controller** from the **Editor** menu.
4. Next, select the **+** button that we added previously, and hold down the *Control* key while dragging it to the Navigation controller, and release the mouse button.
5. Finally, select **Modal** from the pop-up list of choices.
6. Repeat steps 4 to 5 to create the relationships for the **Edit** button.

When you select **Modal** from the list of Storyboard segues, a new arrow is placed between the **Table View Controller** screen and the Navigational controller. So, when you press the **+** button, a new screen will be displayed. Next, we need to specify an identifier for our Storyboard Segue that will be responsible for handling the canceling and saving when the **Add New Document** form is closed.

1. Select the segue relationship located between the **Add** button inside the Navigation bar of the **Table View Controller** screen and the Navigation controller for the **New Document** screen.
2. Click on the **Attributes Inspector** button.
3. Change the **Identifier** property to **Add Document**.
4. Change the **Style** property to **Modal**.
5. Change the **Presentation** property to **Form Sheet**.
6. Change the **Transition** property to **Cover Vertical**.

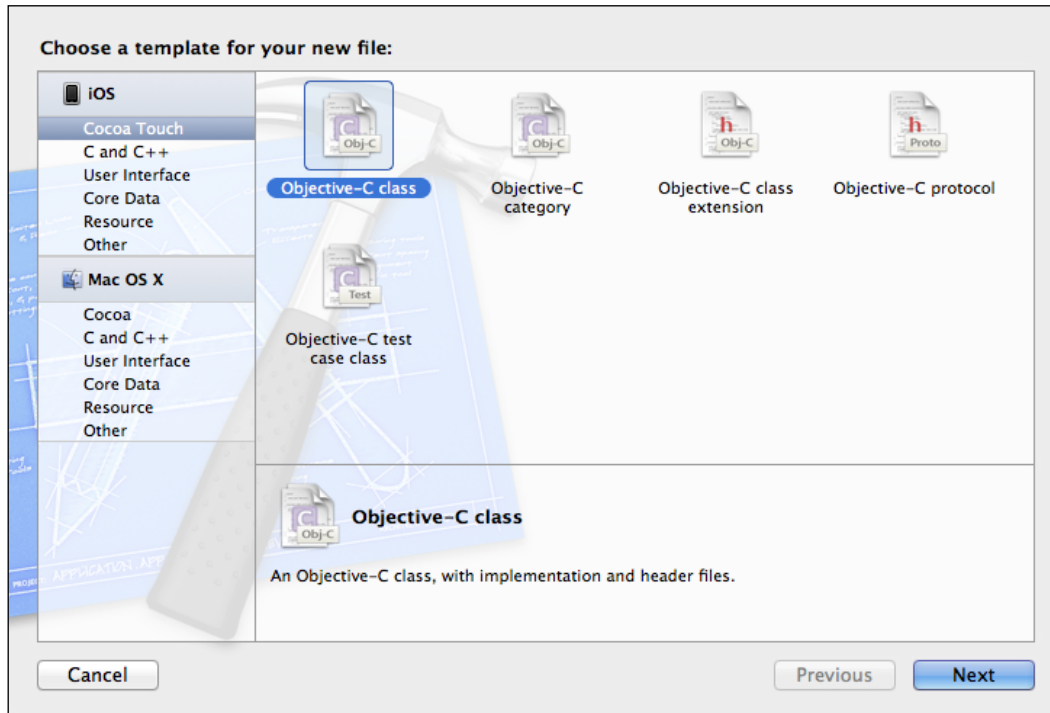
Next, we need to apply the same logic as we did for the **Add New Document** form that will be responsible for calling the same form to handle the editing of an existing document record when the **Edit** button has been pressed.

Repeat steps 1 to 6 to create the segue relationship for the **Edit** button, and set the **Identifier** property to **Edit Document**.

Unfortunately, you won't be able to go back to the previous screen until we create a `UIViewController` subclass, the same as we did for our `ScratchPadViewController`.

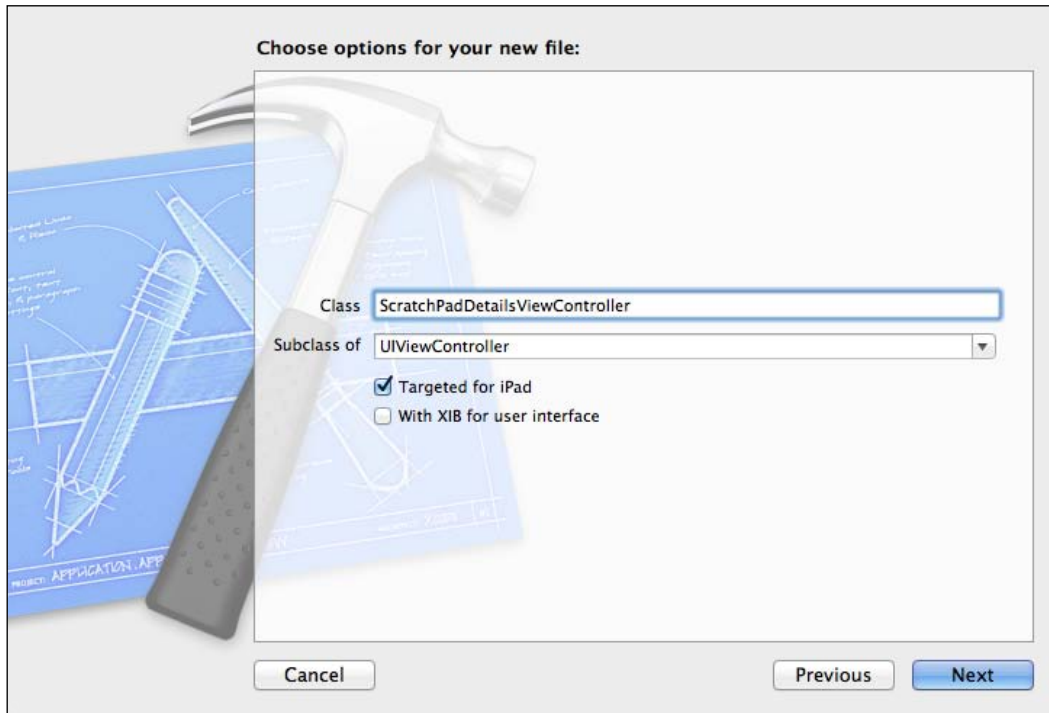
1. From the `ScratchPad` folder, choose **File | New | File...** or press *Command + N*.
2. Select **Cocoa Touch** located under the **iOS** header section.
3. Select the **Objective-C class** template from the list of templates.

- Click on the **Next** button to proceed with the next step in the wizard:



- Enter in `ScratchPadDetailsViewController` as the name of the file to be created.
- Ensure that you have selected `UIViewController` as the type of subclass to be created from the **Subclass of** dropdown.

7. Ensure that you have selected the **Targeted for iPad** option:



8. Click on the **Next** button to proceed with the next step of the wizard.
9. Then, click on the **Create** button to save the file to the folder location specified.

Once you have done this, we need to update the class method of our previously added View controller to use our new **View Controller** subclass. Follow these simple steps:

1. Select the `MainStoryboard.storyboard` file from **Project Navigator**.
2. Click-and-select our newly added ViewController (`UIViewController`) to the right of the `ScratchPadViewController` ViewController.
3. Click on the **Identity Inspector** section, and change the value of the **Custom Class** property to read `ScratchPadDetailsViewController`.
4. Next, from the **Attributes Inspector** section, change the **Title** property to read **Add New Document**.

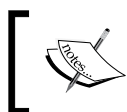
5. From **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control to the top left-hand corner of the navigation bar on the **Add New Document** (UIViewController) section of the **View Controller** screen that we added previously.
6. From the **Attributes Inspector** section, change the **Identifier** property to **Save**.
7. Then, change the **Style** property to **Bordered**.
8. Next, from **Object Library**, select-and-drag a (UIBarButtonItem) **Bar Button Item** control to the top right-hand corner of the navigation bar.
9. From the **Attributes Inspector** section, change the Identifier property to **Cancel**.
10. Change the **Style** property to **Bordered**.

Our next step is to start building the screen that will allow us to create a new document and enter in the relevant information, so that it can be saved to the **Scratch Pad** list:

1. Select the **Add New Document** view controller from within our Storyboard.
2. Next, drag a (UITextView) **TextView** field control onto the canvas.
3. Resize the field so that it fills the whole canvas area.
4. Select **Attributes Inspector** for the **TextView** control.
5. Set the **Alignment field** property to **Left Justify**.

The next step is to create the outlets for each of our controls that we previously added to our **Add New Document** form:

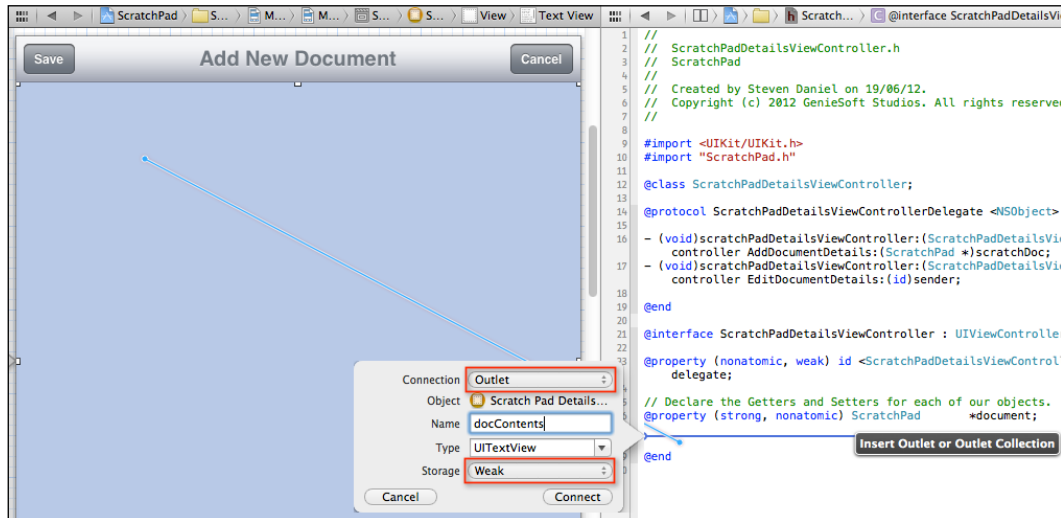
1. Open **Assistant Editor** by choosing **Navigate | Open In Assistant Editor** or press *Option + Command + ,*.
2. Ensure that the `ScratchPadDetailsViewController.h` file gets displayed.
3. Select **TextView** (UITextView), hold down the *Control* key, and drag it into the `ScratchPadDetailsViewController.h` interface file.



In order to create the `IBOutlet` properties, these will need to be created inside the curly braces `{ }` under the `@interface` directive; as these are not created by default, you will need to add them.

4. Choose **Outlet** from the **Connection** drop-down list for the connection to be created.
5. Enter in `docContents` for the name of the property to be created.

6. Choose **Weak** from the **Storage** drop-down list:



Now that we have created our outlets and properties for our control that will hold the document content, we need to start modifying our `ScratchPadDetailsViewController.h` interface file:

1. Open the `ScratchPadDetailsViewController.h` interface file, located within the `ScratchPad` folder, and enter the following highlighted code snippets:

```
// ScratchPadDetailsViewController.h
// ScratchPad
// Created by Steven Daniel on 19/06/12.
// Copyright (c) 2012 GenieSoft Studios. All rights reserved.

#import <UIKit/UIKit.h>
#import "ScratchPad.h"

@class ScratchPadDetailsViewController;

@protocol ScratchPadDetailsViewControllerDelegate
<NSObject>

- (void)scratchPadDetailsViewController: (ScratchPadDetails
ViewController *)controller
AddDocumentDetails: (ScratchPad *)scratchDoc;
- (void)scratchPadDetailsViewController: (ScratchPadDetails
ViewController *)controller EditDocumentDetails: (id) sender;

@end
```

```
@end

@interface ScratchPadDetailsViewController :
    UIViewController

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

// Declare the Getters and Setters for each of our objects.
@property (strong, nonatomic) ScratchPad *document;
@property (weak, nonatomic)
    IBOutlet UITextView *docContents;
@property (strong, nonatomic) NSString *currFile;

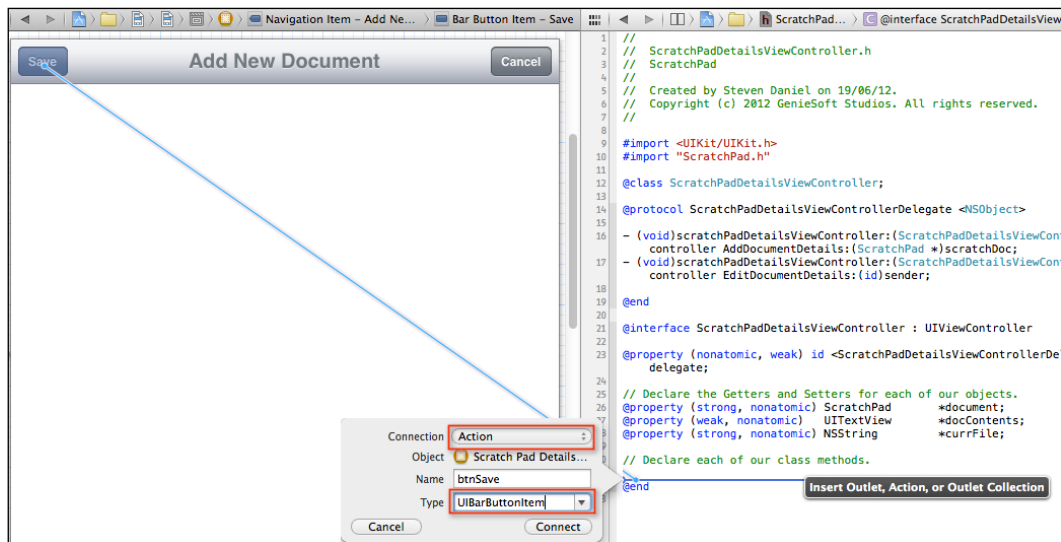
@end
```

In the preceding code snippet, we declare a new delegate object that points to the methods that are responsible when a document is added or edited, and is used to communicate back to our table view screen when the user either cancels or saves the **Add New Document** screen.

We have also declared an object variable called `document` that is an instance of our `ScratchPad` class, and contains all of the properties and methods that are associated with the `UIDocument` class. Next, we need to create the associated Action events for those Outlets. To create an Action, follow the next steps.

2. With the `ScratchPadDetailsViewController.h` interface file still displayed to the right of the **Add New Document** screen, select the **Save** (`UIBarButtonItem`) control, then hold down the *Control* key, and drag it into the `ScratchPadDetailsViewController.h` interface file.
3. Choose **Action** from the **Connection** drop-down list for the connection to be created.
4. Enter in `btnSave` for the name of the method to be created.
5. Choose `UIBarButtonItem` from the **Type** drop-down list for the type to be created.

- Repeat steps 2 to 4 and hook up the **Cancel** button, as well as creating the action event `btnCancel` for the method name:



In the next section, we will take a look at building the functionality for our Scratch Pad application, so that it has the ability to add new and edit existing documents from our application's iCloud repository from the **Scratch Pad** table view:

- Now that we have created the Action events for our View controller, it would be a good time to build and run our application to ensure that no program errors exist.
- Choose **Product | Run** from the **Product** menu, or alternatively press *Command + R*.

Functionality

Well done! You have made it this far; we have successfully finished building the user interface for both the **Scratch Pad** and **Add New Document** screens. Our next step is to start implementing the methods that will be used by our **Cancel** and **Save** buttons; these will be responsible for returning us back to the **Scratch Pad** screen:

- Open the `ScratchPadDetailsViewController.m` implementation file, located within the `ScratchPad` folder, and add the following `synthesize` methods:

```
#import "ScratchPadDetailsViewController.h"
```

```

@interface ScratchPadDetailsViewController ()

@end

@implementation ScratchPadDetailsViewController

    @synthesize document;
    @synthesize delegate;
    @synthesize docContents = m_docContents;
    @synthesize currFile;

```

2. Next, modify the `viewDidLoad` method, as shown in the following code snippet:

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Do any additional setup after loading the view.
    // Set the background color and font attributes
    // for our document.
    UIFont *font = [UIFont fontWithName:@"Helvetica-Bold"
        size:[UIFont systemFontOfSize]];
    [self.docContents setFont:font];
    [self.docContents setBackgroundColor:[UIColor
        colorWithRed:1.0f green:1.0f blue:0.6f alpha:1.0f]];

    // Populate our TextView Control with the data for
    // the selected record.
    [self.docContents setText: self.document.docContent];

```

In the preceding code snippet, we start by declaring a `UIFont` object, and then use the `setFont` method to set up and initialize our font to be bold, and to use the system font size. Next, we use the `setBackgroundColor` method to initialize our document canvas background to render as yellow, similar to that of the `Notes` application that comes with iOS. Finally, we populate our `UITextView` control's `text` property, with the text associated with our selected document.

Implementing the btnSave method

We are now ready to start implementing a method that will be responsible for saving the record when the user presses the **Save** button:

1. Open the `ScratchPadDetailsViewController.m` implementation file, located within the `ScratchPad` folder, and enter the following code snippet:

```
-(IBAction)btnSave:(id)sender
{
    NSString *fileName = @"";
    NSDateFormatter *formatter = [[NSDateFormatter alloc]
        init];
    [formatter setDateFormat:@"%ddMMyyyy_hhmmss"];

    // If we are not editing a current document, generate a
    //new filename with the dateTime stamp, appended to it.
    if(!self.currFile)
    {
        fileName = [NSString
            stringWithFormat:@"Journal_%.doc", [formatter
                stringFromDate:[NSDate date]]];
    }
    else
    {
        fileName = self.currFile;
    }
}
```

In the preceding code snippet, we start by declaring an `NSDateFormatter` object, and then use the `setDateFormat` method to set up and initialize the correct date format that we would like our filename to be generated with. We then perform a check to see if we are not currently editing an existing document. If we are creating a new document, we append the date and time to the end of the filename; otherwise, we just initialize our filename to use the name associated by our `currFile` variable:

```
// Point to our iCloud Documents container.
NSURL *ubiq = [[NSFileManager defaultManager]
    URLForUbiquityContainerIdentifier:nil];
NSURL *ubiquitousPackage = [[ubiq
    URLByAppendingPathComponent:@"Documents"]
    URLByAppendingPathComponent:fileName];
ScratchPad *doc = [[ScratchPad alloc]
    initWithFileURL:ubiquitousPackage];
doc.docContent = docContents.text;
```

In our preceding snippet, we set up our `ubiq` variable to point to our current document's container within our iCloud account, then use the `ubiquitousPackage` class, and append our filename at the location of the iCloud document container. We then initialize our `UIDocument` document with the contents of our `UITextView` object.

```
// Check to see if we are editing a currently opened note
if(!self.currFile) {
    [doc saveToURL:[doc fileURL]
     forSaveOperation:UIDocumentSaveForCreating
     completionHandler:^(BOOL success) {
        if (success) {
            [self.delegate
             scratchPadDetailsViewController:self
             AddDocumentDetails:doc];
        }
    }];
}
else
{
    // Overwrite the file that is currently being edited.
    [doc saveToURL:[doc fileURL]
     forSaveOperation:UIDocumentSaveForOverwriting
     completionHandler:^(BOOL success) {
        if (success) {
            [self.delegate
             scratchPadDetailsViewController:self
             EditDocumentDetails:sender];
        }
    }];
}
```

In the preceding code snippet, we notify our delegate object that we have added a new document item, so that it can update `ScratchPad` for iCloud table view. If we are in the process of editing a document, we use the `UIDocumentSaveForOverwriting` property of our `forSaveOperation` method, to denote that we are overwriting the document contents, which are stored within our document container. Otherwise, we just use the `UIDocumentSaveForCreating` property of our `forSaveOperation` method, to create a brand new document.



For more information about the `UIDocument` class, please refer to the following link: http://developer.apple.com/library/ios/#documentation/uikit/reference/UIDocument_Class/UIDocument/UIDocument.html

2. Now that we have created our `UIDocument` class, it would be a good time to build and run our application to ensure that no program errors exist.
3. Choose **Product** | **Run** from the **Product** menu, or alternatively press *Command + R*.

Implementing the `btnCancel:` method

Next, we need to implement the **Cancel** button. This will be responsible for closing the screen, and returning you back to the **Scratch Pad** table view when pressed.

Open the `ScratchPadDetailsViewController.m` implementation file, located within the `ScratchPad` folder, and enter the following code snippet:

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

In the preceding code snippet, we use the `dismissViewControllerAnimated` method, which is only made available in iOS 5 and later. This method is used to close the current modal screen that was sent by our **Scratch Pad** table view screen.

Implementing the `AddDocumentDetails:` method

In the previous section, we added some code to our `Save` method that created a new `ScratchPad` document instance, and then sent this information to the delegate object, located within `scratchPadDetailsViewController`. Next, we need to create the `AddDocumentDetails` method that will be responsible for adding the document to our document array.

Open the `ScratchPadViewController.m` implementation file, located within the `ScratchPad` folder, and enter the following code snippet:

```
-(void)scratchPadDetailsViewController:(
    ScratchPadDetailsViewController *)controller
AddDocumentDetails:(ScratchPad *)scratchDoc
{
    [self.document addObject:scratchDoc];
    [self.tableView reloadData];
}
```

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

In the preceding code snippet, we add the new `scratchDoc` object to our existing list of documents. We then refresh the table view, using the `reloadData` method to show that the new item was added, and then we close the **Add New Document** screen.

Implementing the `EditDocumentDetails:` method

In the previous section, we added some code to our `Save` method that created a new `ScratchPad` instance, and sent this information to the `delegate` object, located within `ScratchPadDetailsViewController`. In our next step, we need to create the `EditDocumentDetails` method that will be responsible for disabling our **Edit** button, once we have returned back from the **Add New Document** form.

Open the `ScratchPadViewController.m` implementation file, located within the `ScratchPad` folder, and enter the following code snippet:

```
- (void)scratchPadDetailsViewController:(  
    ScratchPadDetailsViewController *)controller  
    editDocumentDetails:(id)sender  
{  
    self.btnEdit.enabled = NO;  
    [self getScratchPadDetails];  
    [self dismissViewControllerAnimated:YES completion:nil];  
}
```

In the preceding code snippet, we disable our **Edit** button and repopulate our table view controller with the list of documents obtained from our iCloud repository, by calling the `getScratchPadDetails` method, and then we close the **Add New Document** screen.

Finishing up

We just have a few more things to implement before we have a complete working application. We will need to implement a couple more methods that will handle the transition between our **Scratch Pad** and **Add New Document** screens when the **+** and **Edit** buttons have been pressed. Firstly, let's handle the transition between the **Scratch Pad** screen and the Navigation controller, to determine whenever a transition is made on a segue within the storyboard.

Open the `ScratchPadViewController.m` implementation file, located within the `ScratchPad` folder, and enter the following code snippet:

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender
{
    // Find our Scratch Pad Details View Controller
    // within our Storyboard.
    UINavigationController *navigationController =
        segue.destinationViewController;
    ScratchPadDetailsViewController
        *scratchPadDetailsViewController =
        [[navigationController viewControllers]
         objectAtIndex:0];

    scratchPadDetailsViewController.delegate = self;
    scratchPadDetailsViewController.title = @"New
        Document";
    scratchPadDetailsViewController.navigationController.
        navigationBar.tintColor = [UIColor blueColor];
    scratchPadDetailsViewController.currFile = nil;

    // If we are editing the currently selected document
    if ([segue.identifier isEqualToString:@"EditDocment"])
    {
        NSInteger selectedRow = [[self.tableView
            indexPathForSelectedRow] row];
        ScratchPad *scratchDoc = [self.document
            objectAtIndex:selectedRow];

        // Set the title for our form to show we are
        // editing, pass the document contents.
        scratchPadDetailsViewController.title = [NSString
            stringWithFormat:@"Editing: %@",
            scratchDoc.fileURL.lastPathComponent];

        scratchPadDetailsViewController.navigationController.
            navigationBar.tintColor = [UIColor orangeColor];
        scratchPadDetailsViewController.currFile =
            scratchDoc.fileURL.lastPathComponent;
        scratchPadDetailsViewController.document =
            scratchDoc;
    }
}
```

In the preceding code snippet, we use the `prepareForSegue:` method to determine whenever a transition to a segue takes place. A check is required to be made on the identifier of the segue to determine if we are making a call to the **Add New Document** screen, pass the selected document to the `ScratchPadDetailsViewController` class, and update the header for the form to show we are currently in the edit mode.

If the determination has been made that editing has not taken place, a new blank document will be displayed. Next, we set `navigationController` of the segue to be the navigation controller of the destination screen, and then cycle through each of the view controller within the navigation controller properties to get the `ScratchPadDetailsViewController` instance.

Requesting entitlements for iCloud storage

In order to protect the data your application creates, a number of specific entitlements need to be created at build-time, in order to use iCloud storage. You will need to ensure that you have selected the option to enable iCloud for your application's App ID.

You will need to create a new App ID from within the iOS Provisioning Portal, located at <https://developer.apple.com/ios/manage/bundles/index.action>. If you are using an existing ID, this must not be a wild card one; that is, `com.yourcompany.*`.

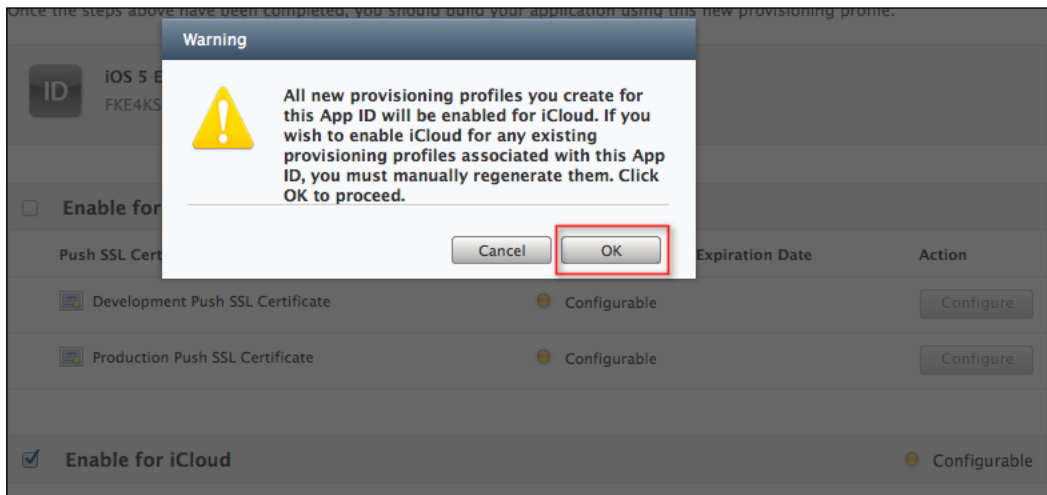
To enable iCloud services for your App ID, follow these simple steps:

1. Firstly, you will need to create a new App ID or edit the one that you have created previously.
2. Then, set up your provisioning profile for use with iCloud, by simply checking the **Enable for iCloud** checkbox from the **Configure App ID** screen:

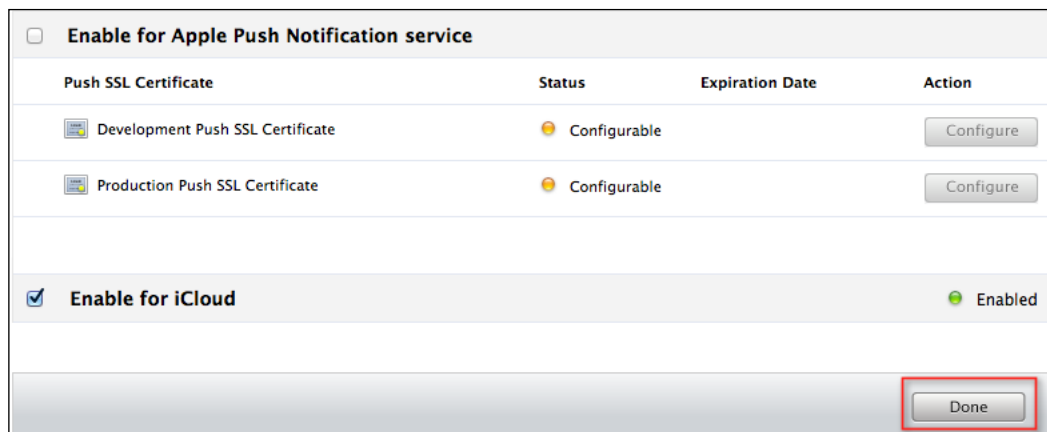
Push SSL Certificate	Status	Expiration Date	Action
Development Push SSL Certificate	Configurable		<button>Configure</button>
Production Push SSL Certificate	Configurable		<button>Configure</button>

☐ **Enable for iCloud** Configurable

- Next, you will be presented with a pop-up dialog box, explaining that any new provisioning profiles that you create using the chosen App ID will be enabled for iCloud services:



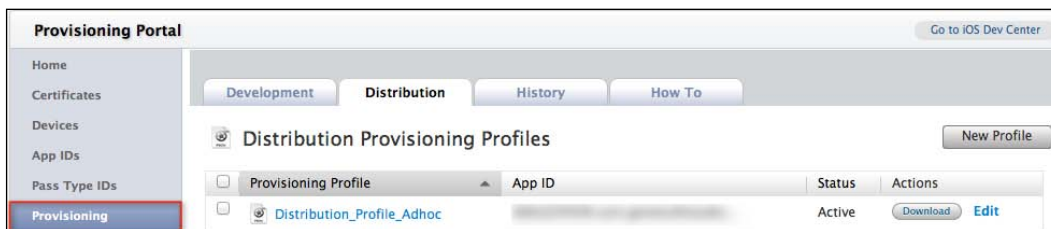
- Once you have clicked on the **OK** button, the pop-up dialog box will disappear, and you will be returned back to the **Configure App ID** screen, and the **Enable for iCloud** button will be set to green, as shown in the following screenshot:



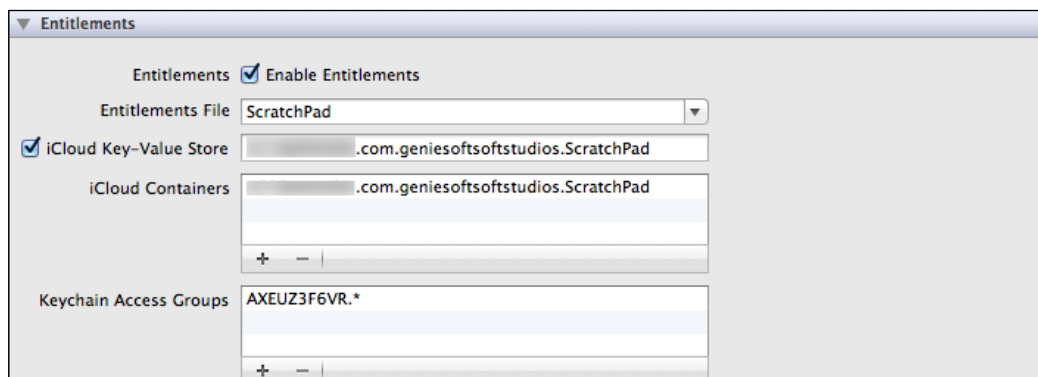
- Click on the **Done** button to close this screen.
- Next, click on the **Provisioning** tab, and then click on the **Development** tab to download your **Development Provisioning Profiles**, as shown in the following screenshot:



7. Next, from the **Provisioning** tab, click on the **Distribution** tab to download your **Distribution Provisioning Profiles**, as shown in the following screenshot.



8. Next, from the **Project Navigator** window, click on your project, on the **Targets** section, and then on the **Summary** page.
9. Scroll down till you get to the **Entitlements** section.
10. Check the **Enable Entitlements** and the **iCloud Key-Value Store** checkboxes. This will add a file called `ScratchPad.entitlements` to your project.



When you add entitlements to your project, they are bound directly to your application's provisioning profiles that are used to separate your application's documents and data repositories from those of other applications that you create. There are two entitlements that an application can request, depending on which iCloud features it is required to use. These are explained in the following table:

Entitlement	Description
<code>com.apple.developer.ubiquity-container-identifiers</code>	Use this to request the iCloud document storage entitlement. The value of this key is an array of container-identifier strings. (The first string in the array must not contain any wildcard characters.)
<code>com.apple.developer.ubiquity-kvstore-identifier</code>	Use this to request the iCloud key-value data store entitlement. The value of this key is a single container identifier string.

When you specify the container identifier string, it must be in the form of `<TEAMID>.<CUSTOM_STRING>`, where `<TEAMID>` is the unique 10-character identifier associated with your development team. The `<CUSTOM_STRING>` identifier is a reverse-DNS string that identifies the container for storing your application's documents.



To locate your unique identifier associated with your development team, log in to the Apple Developer Connection website, and then go to the **Member Center** page (<http://developer.apple.com/membercenter>). Select the **Your Account** tab, and then select **Organization Profile** (if you have set up your profile to be used as an organization) from the column on the left of that tab. Your team's identifier is in the **Company/Organization ID** field.

Applications using iCloud document storage can specify multiple containers for storing documents and data. The `com.apple.developer.ubiquity-container-identifiers` key is an array of strings. The following XML from the `ScratchPad` entitlements file shows the keys for an iOS application that can read/write its own documents, which are stored in the container directory, identified as shown in the following highlighted code sections:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPEplist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>application-identifier</key>
    <string>AXEUZ3F6VR.com.geniesoftstudios</string>
    <key>com.apple.developer.ubiquity-container-identifiers</key>
    <array>
      <string>TEAMID.com.yourcompany.ScratchPad</string>
    </array>
    <key>com.apple.developer.ubiquity-kvstore-identifier</key>
    <string>TEAMID.com.yourcompany.ScratchPad</string>
    <key>get-task-allow</key>
    <true/>
  </dict>
</plist>

```

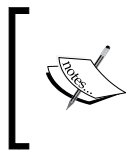
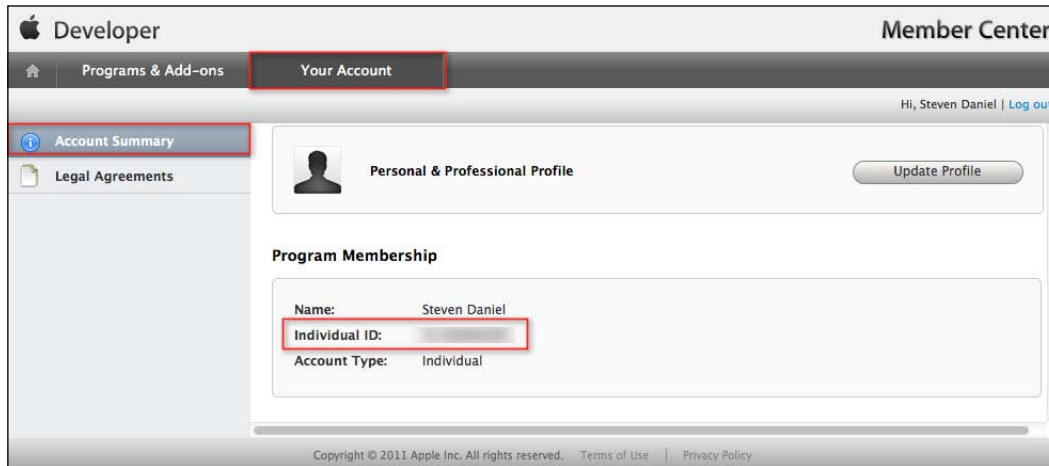


The `application-identifier` field is your application bundle ID that will need to be added manually into this file. The `get-task-allow` key will also need to be added manually into this file also.

The following screenshot displays the property list view within the project navigator of the `ScratchPad.Entitlements` entitlements file:

Key	Type	Value
application-identifier	String	AXEUZ3F6VR.com.geniesoftstudios
▼ com.apple.developer.ubiquity-container-	Array	(1 item)
Item 0	String	com.geniesoftsoftstudios.ScratchPad
com.apple.developer.ubiquity-kvstore-	String	.com.geniesoftsoftstudios.ScratchPad
get-task-allow	Boolean	YES
► keychain-access-groups	Array	(1 item)

The `TEAMID` value (as shown in the previous screenshot), can be obtained from the Account Summary page of your developer account, and using the **Individual ID**, as shown in the following screenshot:



The strings you specify in your entitlement's `property-list` file are also the strings you pass to the `URLForUbiquityContainerIdentifier:` method, when requesting the location of a directory in the user's iCloud storage.

Configuring your iOS device to use iCloud

Before our application can start to store data within our iCloud application repository, we will need to properly configure and set up our application to use iCloud, and store documents onto an iOS device; the device must first be running iOS 5 or later.

The following steps show you how easy it is to set up an iCloud account:

1. From the **Settings** pane within your device, select **iCloud**. This is shown in the following screenshot:



2. Next, sign in with your Apple ID and password, and then click on the **Sign In** button, as shown in this screenshot.
3. You will need to agree to the iCloud terms and conditions, and then click on the **Agree** button to close the pop-up dialog box.

- Next, click on the **Storage & Backup** option to proceed with the next screen:



- Next, set the **Backup to iCloud** option to **ON**, from under the **Backup sections** pane. This will automatically start synching your **Mail, Contacts, Calendars, Reminders, Bookmarks, or Notes**, and start pushing or pulling your account information to iCloud.



If you prefer, you can also log in to your iCloud account by using any web browser at <http://www.iCloud.com/>, using the same information you entered into your iOS device. Once you are successfully logged in, you can choose **Contacts** or **Calendar** to see your data already pulled into the cloud. Making edits via the web interface will push them directly back to your iOS device.

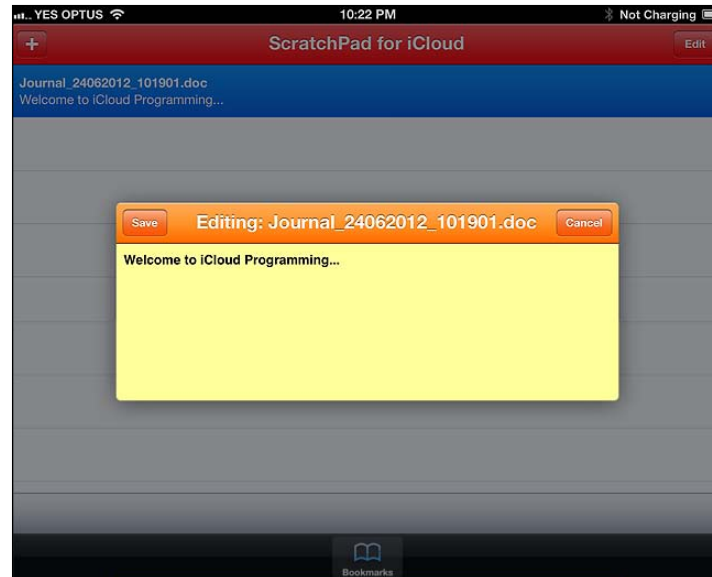
iCloud storage space

iCloud is a free service that comes with an initial 5 Gigabytes of free storage space, upon successful signup. This then allows you to synchronize all of your contact information, e-mails, and documents. Should you require additional storage space, Apple provides this to you through the **iCloud Settings** menu under **Storage & Backup**. For \$20 a year you can purchase 10GB of space, \$40 a year offers 20GB of space, and \$100 a year gets you 50GB of space; these prices are similar to what Google offers with Gmail.

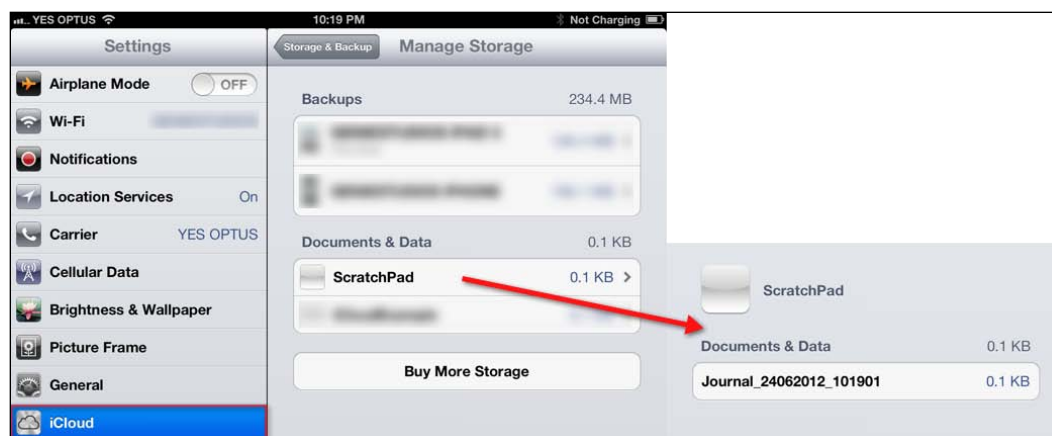


When using the iCloud storage APIs from within your applications, any documents that your application stores explicitly in iCloud are not backed up with your application, as these will already be stored within your iCloud account, and therefore do not need to be backed up separately.

Congratulations, you have finally implemented the methods for your `ScratchPad` application. Next, build and run the application by pressing *Command + R*. The following screenshot shows the application running on the iOS device with the first item in the list being selected:



From the preceding screenshot, you can see that when the item has been selected and the **Edit** button has been pressed, the details for the selected row are passed to the **Add New Document** form, and all of the document details are read from the iCloud repository, and then populated to the form. The following screenshot displays the document existing within our application's container within iCloud:



Summary

In this chapter, we learned about the benefits of using iCloud, and how to access them through their storage APIs. We learned how to create a *ScratchPad* application, making use of and incorporating iCloud functionality to store and retrieve documents within our applications iCloud repository. We also learned how we can search and locate a document within our iCloud repository, through the use of query predicates, as well as learning the process of how to handle and avoid file-version conflicts when multiple copies of the same file is being updated on more than one device, and then being submitted to the iCloud repository.

This was the final chapter; refer to the *Appendix* to learn more about the iOS Human Interface Guidelines, and what you can and can't include within your applications. We will also learn about the use of Xcode Instruments, and how we can use these to track and improve our applications performance. Finally, we will look at the steps required to submit and distribute our application to the Apple App Store.

Index

A

Accelerometer/gyroscope 18

Action button

- adding, to AddressBook app 117, 118
- adding, to Facebook app 282-285

ActionSheet 320

ActionUIBarButtonItem 317

Add button

- adding, to AddressBook app 116
- adding, to ScratchPad app 356
- adding, to TaskPriorities App 40
- adding, to VeterinaryClinic app 234

AddDocumentDetails: method

- implementing, on ScratchPad app 380

additional permissions, Facebook app

- requesting 296, 298

Address book 15

AddressBook application

- about 95
- Action button, adding 117-124
- Action button method, implementing 140-142
- Add button, adding 116
- Add New Contact screen, creating 125-131
- building 98-100
- cancel method, implementing 133
- connect method, implementing 137-140
- contact details, transferring using Bluetooth 135
- Core Data framework 96
- Core Data model, building 102, 103
- Core Data model files, creating 104-108
- delete row method, implementing 133, 134
- didSelectRowAtIndexPath method, implementing 134, 135

Game Kit framework 96

Game Kit framework, adding 100, 101

main application screen, creating 112

running 143

save record method, implementing 132, 133

search functionality, implementing 144-150

Storyboard screen, adding 108-111

table control, adding 112-116

AddTaskDetails method

- implementing 61

addWayPoint:method 211

AirPlay functionality

- considerations 332

enabling 329

features 329

used, for presenting app content to Apple

TV 330-332

Alerts 18

Apple Video Graphics Adaptor (VGA) 312

Application Delegate class

- implementing, in Facebook app 287-291

Audio mixing 17

audioRecorder object 85

Audio recording 17

authorize method 270, 290

AVAudioRecorder class 68

AVFoundation framework

- about 68

adding, to VoiceRecorder app 70

B

basic screen orientations

landscape left 32

landscape right 33

portrait 32

- portrait upside-down 32
 - Battery Gauge class**
 - about 151
 - implementing, on BatteryMonitor app 176-183
 - BatteryMonitor application**
 - about 151
 - Battery Monitor functionality, building 164
 - building 153, 154
 - Enable Monitoring UISwitch control, adding 156, 157
 - Fill Gauge Levels UISwitch control, adding 158
 - Increment Bars UIStepper control, adding 158, 159
 - main application screen, creating 155
 - MessageUI framework, adding 154, 155
 - running 184, 185
 - Send E-mail Alert UISwitch control, adding 157
 - System Information (UITextView) control, adding 160-164
 - technologies, used 152
 - Battery Monitor functionality**
 - Battery Gauge class, implementing 176-183
 - determineBatteryStatus: method, implementing 167, 170
 - enableMonitoring: method, implementing 170, 171
 - fillGauge: method, implementing 174, 175
 - sendEmailAlert: method, implementing 172-174
 - totalNoBars: method, implementing 175, 176
 - View Controller class, implementing 165, 166, 167
 - Berkeley Standard Distribution (BSD) 14**
 - Binary Data data-type 220**
 - Bonjour**
 - about 15
 - Browse button**
 - adding, to ExternalDisplays app 316
 - btnAddPhoto: method**
 - implementing 260, 261
 - btnBrowse: method**
 - implementing, on ExternalDisplays app 323, 324
 - btnCamera: method**
 - implementing, on ExternalDisplays app 324-326
 - btnCameraPhoto: method**
 - implementing 261, 262
 - btnCancel: method**
 - implementing 260
 - implementing, on ScratchPad app 380
 - btnPlayVideo: method**
 - implementing, on ExternalDisplays app 327, 328
 - btnSave: method**
 - implementing, on ScratchPad app 378, 379
 - btnSavePet: method**
 - implementing 258, 260
 - btnTransitions: method**
 - implementing, on ExternalDisplays app 332, 333
- ## C
- Camera button**
 - adding, to ExternalDisplays app 316
 - Cancel method**
 - implementing 62
 - Certificates 15**
 - CGContextAddLineToPoint 92**
 - CGContextMoveToPoint 92**
 - CGContextSetRGBStrokeColor function 92**
 - CGContextStrokePath function 92**
 - Change Map Type button**
 - adding, to RouteTracker app 195-200
 - changeMapType: method**
 - implementing, on RouteTracker app 206, 207
 - CIColor 335**
 - CIContent 335**
 - CIFilter class 311, 335**
 - CIFilterEffects application 337**
 - CIImage class**
 - used, for applying image filter effects 335-339
 - CIImageParameters**
 - attribute class 336
 - Display name 336
 - Filter category 336
 - Filter name 336

- Input parameters 336
- CIVector** 335
- classes** 96
- clearOSCLevels method** 83-85
- clickedButtonAtIndex method** 306
- CLLocationManagerDelegate protocol** 202
- Cocoa-Touch layer**
 - about 18
 - components 18
- Collections** 15
- components, Cocoa-Touch layer**
 - Accelerometer/gyroscope 18
 - Alerts 18
 - Controllers 18
 - Image picker 18
 - Localization/geographical 18
 - Multi-touch controls 18
 - Multi-touch events 18
 - People picker 18
 - View hierarchy 18
 - Web views 18
- components, Core OS layer**
 - Berkeley Standard Distribution (BSD) 14
 - Bonjour 15
 - Certificates 15
 - File system 15
 - Keychain 15
 - Mach 3.0 14
 - OS X Kernel 14
 - Power management 15
 - Security 15
 - Sockets 14
- components, Core Services layer**
 - address book 15
 - collections 15
 - Core data 16
 - Core location 16
 - File access 15
 - Net services 16
 - networking 15
 - Preferences 16
 - SQLite 16
 - Threading 16
 - URL utilities 16
- components, iOS SDK**
 - DashCode 11
 - Instruments 11
 - iOS Simulator 11
 - Xcode 11
- components, Media layer**
 - Audio mixing 17
 - Audio recording 17
 - Core animations 17
 - Core audio 17
 - Image formats 17
 - OpenGL 17
 - OpenGL ES 17
 - PDF 17
 - Quartz 17
 - Video playback 17
- connectionTypesMask property** 138
- contact details**
 - transferring, Bluetooth used 135, 137
- contactsArray array object** 123
- ContactsViewController interface file** 118, 137
- contentsForType: method** 369
- Controllers** 18
- Core animations** 17
- Core audio** 17
- Core data** 16
- Core Data framework**
 - about 96, 220
 - Managed Object 97
 - Managed Object Context 97
 - Managed Object Model 97
 - management object 97
- Core Data model**
 - about 102
 - building 102, 103
- Core Data model files**
 - creating 104-108
- Core Data model files, VeterinaryClinic app**
 - creating 226, 227
- Core Data model, VeterinaryClinic app**
 - Add button, adding 234
 - attributes, adding 223, 224
 - building 222-225
 - Edit button, adding 235-243
 - entity, creating 223
 - files, creating 226-229
 - main application screen, creating 231
 - screens, navigating between 244-255
 - Storyboard screen, adding 230, 231

- table control, adding 232, 233
- Core Data technologies**
 - overview 96
- Core Graphics framework** 29, 30
- Core Image Application Programming Interface (API)** 312
- Core Image attribute values**
 - colors 336
 - floating-point numbers 336
 - images 336
 - strings 336
 - transforms 336
 - vectors 336
- Core Image class** 335
- Core Image filters**
 - URL 341
- Core Image framework**
 - about 333
 - diagrammatic representation 334
 - features 334
 - uses 334
- CoreImage framework** 312
- Core location** 16
- Core Location framework**
 - adding, to RouteTracker application 191
- Core OS layer**
 - about 14
 - components 14
- Core Services layer**
 - about 15
 - components 15
- CVImageBufferRef** 335

D

- daemon service** 349
- dataWithData:UIImagePNGRepresentation method** 220
- Delete row method**
 - implementing 63, 262, 263
- determineBatteryStatus: method**
 - implementing, on BatteryMonitor app 167, 170
- dialogDidComplete** 303
- didFailWithError method** 301, 304

- didFinishPickingMediaWithInfo method** 262
- didLoad method** 301
- disconnectFromAllPeers method** 138
- dismissViewControllerAnimated method** 380
- drawRect: method** 92

E

- Edit button**
 - adding, to ScratchPad app 356-368
 - adding, to VeterinaryClinic app 234-243
- EditDocumentDetails: method**
 - implementing, on ScratchPad app 381
- E-mail button**
 - adding, to VoiceRecorder application 76-80
- e-mailRecording method**
 - implementing, in VoiceRecorder app 86-89
- enableMonitoring: method**
 - implementing, on BatteryMonitor app 170, 171
- entitlements**
 - requesting, for iCloud Storage 383
- External Displays application**
 - AirPlay, used 329-332
 - building 312
 - content, presenting to external monitor device 342-344
 - functionality 320
 - main application screen, creating 315
 - Media Player framework, adding 314
 - running 345
 - technologies, used 312
- ExternalDisplays Functionality**
 - about 320
 - btnBrowse: method, implementing 323, 324
 - btnCamera: method, implementing 324-327
 - btnPlayVideo: method, implementing 327-329
 - btnTransitions: method, implementing 332, 333
 - shouldAutorotateToInterfaceOrientation: method, implementing 344
 - View Controller class, implementing 320-323

F

Facebook app functionality

- additional permissions, requesting 296
- Application Delegate class, implementing 287-292
- building 286
- errors, handling 304
- Graph API, using 298-301
- loginButton: method, implementing 306
- Log Out functionality, adding to app 295
- postMessageButton: method, implementing 305, 306
- Social channels, integrating with 302, 303
- SSO, implementing within app 286
- View Controller class, implementing 292-294

Facebook application

- about 269
- Action button, adding 282-285
- building 276, 277
- Facebook app functionality, building 286
- Facebook iOS SDK, adding 277-279
- Facebook iOS SDK, downloading 271
- main application screen, creating 280
- running 307, 308
- sign-in button, adding 280, 281
- sign-out button, adding 281, 282
- technologies, used 270

Facebook iOS SDK

- about 269
- adding, to Facebook app 277-279
- downloading 271
- iOS app, registering with Facebook 272-275
- types 270

Facebook iOS SDK types

- authentication and authorization 270
- Display dialog 270
- Make API calls 270

Facebook Query Language (FQL) 298

Facebook SSO process

- running 286

FBAccessTokenKey 291

FBDialogDelegate 288

fbDidLogin method 291

fbDidLogout method 291-295

FBExpirationDateKey 291

FBRequestDelegate 293

FBSessionDelegate 288, 292

fbSessionInvalidated method 292

fetchRequest object 122, 241

fetchResultsController object 122

File access 15

file coordinator 349

file presenter 349

File system 15

fillGauge: method

- implementing, on BatteryMonitor app 174, 175

filter attributes 336

forSaveOperation method 379

frameworks 13

functionality, VeterinaryClinic app

- about 255, 257
- btnAddPhoto
 - method, implementing 260, 261
- btnCameraPhoto
 - method, implementing 261, 262
- btnSavePet
 - method, implementing 258, 260
- Delete row method, implementing 262, 263

G

Game Kit framework

- about 96
- adding, to AddressBook app 100, 101

getContactDetails method 121

getPetDetails method 239

getScratchPadData:query method 365

getScratchPadDetails method 363, 381

GKPeerPickerConnectionTypeNearby 138

GKPeerPickerConnectionTypeOnline 138

GKPeerPickerControllerDelegate 137

Graph API, Facebook app

- using 298-301

H

handleOpenURL method 290

HelloWorld iOS application

- building 19-22
- creating 7

objects, placing within View 22-24
Xcode Developer Tools, removing 24

I

iCloud

about 348
daemon service, using 348
documents, storing 348
documents, using 348
file coordinator 349
file presenter 349
file-version conflicts, handling 352
storage APIs, using 350

iCloud document storage 351

iCloud file-version conflicts

handling 352

iCloud key-value data storage 351

iCloud services

enabling 383, 384

iCloud Storage

entitlements, requesting for 383-388

iCloud storage APIs

about 350
iCloud document storage 351
iCloud key-value data storage 351
using 350, 351

iCloud storage space 391, 392

image filter effects

applying, UIImage class used 335-339

Image formats 17

Image picker 18

ImagePickerController 320

imagePickerControllerDidCancel 262

imgPhoto control 301

initWithCoder: method 91

iOS app

registering, with Facebook 272-275

iOS Developer Program

registering 8, 9

iOS device

configuring, for using iCloud 388-390

iOS SDK

components 11
downloading 10
installing 10, 11
system requirements 9

iOS Simulator

about 12
architecture layers 13, 14
default settings 13
features 12

iOS Simulator application

reference link 13

iPad

about 7
BatteryMonitor application 151
RouteTracker application 187
ScratchPad application 347
VeterinaryClinic application 219

isEqualToString method 137

isTracking variable 205

K

kCLErrorDenied error 209

kCLErrorHeadingFailure error 209

kCLErrorLocationUnknown error 209

kCLErrorNetwork error 209

kCLErrorRegionMonitoringDenied error 209

kCLErrorRegionMonitoringFailure error 209

kCLErrorRegionMonitoringSetupDelayed error 209

key 336

Keychain 15

L

layers, of iOS architecture

about 13, 14
Cocoa-Touch layer 18
Core OS layer 14
Core Services layer 15
Media layer 16

Localization/geographical 18

locationManager

didFailWithError: method
implementing, on RouteTracker app 208

locationManager class 203

locationManager: method

implementing, on RouteTracker app 207, 208

- loginButton: method**
 - implementing 306
- logoutFacebook method** 295
- Log Out functionality**
 - adding, to Facebook app 295

M

- Mac App Store**
 - link 10
- Mach 3.0** 14
- Mac OS X Lion** 10
- main application screen, ExternalDisplays app**
 - about 315
 - Browse button, adding 316
 - Camera button, adding 316
 - Play Video button, adding 317
 - Transitions button, adding 317
 - VGA Out button, adding 317-320
- main application screen, Facebook app**
 - creating 280
- main application screen, RouteTracker app**
 - creating 193
- main application screen, ScratchPad app**
 - creating 354
- main application screen, VeterinaryClinic app**
 - creating 231
- Managed Object** 97
- Managed Object Context** 97
- managedObjectContext method** 239
- Managed Object Model** 97
- MapKit framework**
 - about 188
 - adding, to RouteTracker application 192
- Media layer**
 - about 16
 - components 16
- MediaPlayer framework**
 - about 312
 - adding, to ExternalDisplays app 314
- MessageUI framework**
 - about 68, 152
 - adding to BatteryMonitor app 154, 155
 - adding, to VoiceRecorder app 71

- MFMailComposeViewController class** 151, 152
- MFMailComposeViewController class object** 68
- MFMailComposeViewControllerDelegate class** 81
- MKMapTypeHybrid** 188
- MKMapTypeSatellite** 188
- MKMapTypeStandard** 188
- MKMapViewDelegate protocol** 188, 202
- modal segue** 125, 244
- MPMoviePlayerController** 328
- MPMoviePlayerPlaybackDidFinish Notification** 328
- multiple screen orientations**
 - handling 32
- Multi-touch controls** 18
- Multi-touch events** 18
- mutableFetchResults method** 241

N

- Navigational controller** 31
- NavigationController** 320
- Net services** 16
- Networking** 15
- NSArray object** 301
- NSDictionary object** 301
- NSFetchedResultsController** 118
- NSFetchedResultsController object** 237
- NSFileCoordinator class** 349
- NSFilePresenter protocol** 349
- NSManagedObject class** 227
- NSManagedObjectContext** 118
- NSManagedObjectContext object** 237
- NSManagedObjectContexttthat** 253
- NSMutableArray array** 118, 237
- NSSearchPathForDirectoriesInDomains class** 68, 83
- numberOfRowsInSection method** 123, 242, 361

O

- OpenGL** 17
- OpenGL ES** 17
- OSCLevel array** 90

OSCLLevel NSMutableArray object 90
OS X Kernel 14

P

PDF 17
peerPickerControllerDidCancel: method 139
People picker 18
performFetch method 122
PetDetails class files 228
PetDetails entity
 creating 223
PetDetails.h interface 228
PetDetailsViewController.h interface 254
petListArray property 241
PetsViewController interface file 237
Play button
 adding, to VoiceRecorder application 74
Play Video button
 adding, to ExternalDisplays app 317
PopoverController 320
postMessageButton: method
 implementing 305, 306
Power management 15
Preferences 16
push notifications 302

Q

Quartz 17
QuartzCore framework 312
Quartz Core frameworks
 URL 341

R

receiveData:fromPeer:inSession
 context: method 142
Refresh button
 adding, to TaskPriorities App 41-43
Refresh button method
 implementing 62
Refresh Map button
 adding, to RouteTracker app 194, 195
refreshMap: method
 implementing, on RouteTracker app 205

reloadData method 122, 241
removeAllObjects method 91
requested permission, Facebook application
 basic information (no permissions) 296
 extended permissions 296
 open graph permissions 296
 page permissions 296
 user and friend permissions 296
requestWithMethodName method 301
resetWayPoints method 206, 211
rippleEffect transition effect 340
RouteTracker application
 about 187
 building 189, 190
 Change Map Type button, adding 195-200
 Core Location framework, adding 191
 main application screen, creating 193
 MapKit framework, adding 192
 Refresh Map button, adding 194, 195
 RouteTracker functionality, building 200
 running 215, 216
 Start Tracking button, adding 193, 194
 technologies, used 188

RouteTracker functionality

building 200
changeMapType: method, implementing 206, 207
locationManager:didFailWithError: method, implementing 208
locationManager: method, implementing 207, 208
refreshMap: method, implementing 205, 206
shouldAutorotateToInterfaceOrientation: method, implementing 210
startTracking: method, implementing 204, 205
TrackMapView class, implementing 210-214
View Controller class, implementing 201-203

S

Satellite or Hybrid views 188
Save record method
 implementing 61

ScratchPad application

- about 347
- Add button, adding 356
- building 352-354
- Edit button, adding 356-368
- main application screen, creating 354
- running 381, 383
- screens, navigating between 369-376
- table control, adding 354, 355

ScratchPadDetailsViewController class protocol 359

ScratchPad Functionality

- about 376, 377
- AddDocumentDetails: method, implementing 380
- btnCancel: method, implementing 380
- btnSave: method, implementing 378-380
- EditDocumentDetails: method, implementing 381

Security 15

segue 31, 125

segway 27

sendEmailAlert: method

- implementing, on BatteryMonitor app 172-174

session:didChangeState: method 139

setBackgroundColor method 377

shouldAutorotateToInterfaceOrientation 32

shouldAutorotateToInterfaceOrientation: method

- implementing, on External Displays app 344
- implementing, on RouteTracker app 210

showInView:self:view method 306

Sign-in button

- adding, to Facebook app 280, 281

Sign-out button

- adding, to Facebook app 281, 282

Single Sign-On (SSO) feature

- about 269
- implementing, in Facebook app 286

social channel dialogs

- Feed dialog 302
- Requests dialog 302

social channels

- integrating, with Facebook app 302, 303

Sockets 14

Software Development Kit (SDK) 7

SQLite 16

stack 96

startPlayback method 84

Start Recording button

- adding, to VoiceRecorder application 73, 74

startRecord method 84

startTime variable 205

Start Tracking button

- adding, to RouteTracker app 193, 194

startTracking: method

- implementing, on RouteTracker app 204, 205

Stop button

- adding, to VoiceRecorder application 75

stopPlayback method 84

stopUpdatingHeading 205

stopUpdatingLocation 205

Storyboard screen

- adding, to AddressBook app 108-111
- adding, to VeterinaryClinic app 230, 231

T

Tab Bar controller 32

table control

- adding, to ScratchPad app 354, 355
- adding, to TaskPriorities App 33

TaskPriorities application

- about 27
- Add a record, implementing 61, 62
- Add button, adding 40
- building 28, 29
- Cancel method, implementing 62
- Delete row method, implementing 63
- main application screen, creating 31, 32
- multiple screen orientations, handling 32
- Refresh button, adding 41-51
- Refresh button method, implementing 62
- required frameworks, adding 30
- running 64, 65
- Save record method, implementing 61
- screens, navigating between using Storyboards 51-60
- table control, adding 33-39
- Xcode, creating 28

- technologies, ExternalDisplays app
 - CoreImage 312
 - MediaPlayer 312
 - QuartzCore 312, 313
- technologies, ScratchPad app
 - iCloud 348
- Threading 16
- totalNoBars: method
 - implementing, on BatteryMonitor app 175, 176
- TouchedEnded 18
- TouchesBegan 18
- TouchesMoved 18
- TrackingOverlay class 187
- TrackMapView class
 - implementing, on RouteTracker app 210-214
- trackMapView custom class 203
- transitions
 - about 340
 - applying, to images 340, 341
- Transitions button
 - adding, to ExternalDisplays app 317

U

- ubiquitousPackage class 379
- UIActionSheet class 306
- UIActionSheetDelegate 293, 322
- UIActionSheetDelegate protocol 202
- UIAlertView class 18
- UIBarButtonItem control 317
- UIDevice class 152
- UIDocumentSaveForCreating property 379
- UIGraphicsGetCurrentContext function 92, 212
- UIImagePickerControllerDelegate 253, 322
- UIImagePickerControllerMediaType
 - property 326
- UIImagePickerControllerMediaURL 326
- UINavigationController screen 114
- UINavigationControllerDelegate 253, 322
- UIPopoverControllerDelegate 253
- UIPopoverControllerDelegate 322
- UIScreen class 311
- UISearchBarDelegate object 137
- UITableView control 28, 98

- UITableViewController class 236
- UITableViewController control 33, 112
 - implementing 112
- UIViewAnimationCurveEaseInOut 340
- uninstall-devtools script 24
- URLForUbiquityContainerIdentifier
 - method 363
- URL utilities 16

V

- VeterinaryClinic application
 - about 219
 - building 220-222
 - Core Data model, building 222, 223
 - functionality 255
 - running 264-266
 - technologies, using 220
- VeterinaryClinic.xcdatamodeld file 223
- VGA Out button
 - adding, to ExternalDisplays app 317
- Video playback 17
- ViewController 24
- View Controller class
 - adding, to VoiceRecorder application 80-83
 - implementing, on BatteryMonitor app 165-167
 - implementing, on ExternalDisplays app 320-323
 - implementing, on Facebook app 292-294
 - implementing, on RouteTracker app 201-203
- ViewController.h interface 319
- viewDidAppear method 122, 241
- viewDidLoad method 121, 202, 239, 294, 362, 377
- View hierarchy 18
- Visualizer class 84
- voicePlayback method
 - implementing, in VoiceRecorder app 85
- voicePlaybackStop method
 - implementing, in VoiceRecorder app 86
- VoiceRecorder application
 - about 67
 - AVFoundation framework, adding 70
 - building 68-70
 - E-mail button, adding 76-80

- e-mailRecording method, implementing 86, 87, 89
- main application screen 72, 73
- MessageUI framework, adding 71, 72
- overview 68
- Play button, adding 74
- running 92-94
- Start Recording button, adding 73, 74
- Stop button, adding 75
- View Controller class, implementing 80-83
- voicePlaybackStop method, implementing 86
- voiceRecord method, implementing 83-85
- VoiceVisualizer class, implementing 89-92
- voiceRecord method**
 - implementing, in VoiceRecorder app 83, 84
- VoiceVisualizer class**
 - implementing, in VoiceRecorder app 89-92

W

- wayPoints array** 211
- wayPointsNSMutableArray object** 211
- Web views** 18

X

- Xcode**
 - about 10
 - installing 10
- Xcode Developer Tools** 7



Thank you for buying iPad Enterprise Application Development BluePrints

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

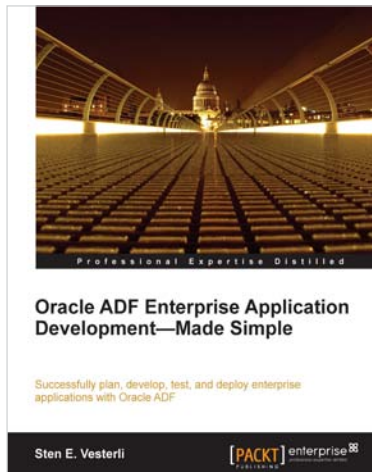
Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



This material is copyright and is licensed for the sole use by on 7th October 2012



Oracle ADF Enterprise Application Development—Made Simple

ISBN: 978-1-849681-88-9

Paperback: 396 pages

Successfully plan, develop, test and deploy enterprise applications with Oracle ADF

1. Best practices for real-life enterprise application development
2. Proven project methodology to ensure success with your ADF project from an Oracle ACE Director
3. Understand the effort involved in building an ADF application from scratch, or converting an existing application



Microsoft SQL Azure: Enterprise Application Development

ISBN: 978-1-849680-80-6

Paperback: 420 pages

Build enterprise-ready applications and projects with SQL Azure

1. Develop large scale enterprise applications using Microsoft SQL Azure
2. Understand how to use the various third party programs such as DB Artisan, RedGate, ToadSoft etc developed for SQL Azure
3. Master the exhaustive Data migration and Data Synchronization aspects of SQL Azure.

Please check www.PacktPub.com for information on our titles



Microsoft SharePoint 2010 Enterprise Applications on Windows Phone 7

ISBN: 978-1-849682-58-9

Paperback: 252 pages

Create enterprise-ready websites and applications that access Microsoft SharePoint on Windows Phone 7

1. Provides step-by-step instructions for integrating Windows Phone 7-capable web pages into SharePoint websites
2. Provides an overview of creating Windows Phone 7 applications that integrate with SharePoint services
3. Examines Windows Phone 7's enterprise capabilities
4. Highlights SharePoint communities and their use in a Windows Phone 7-connected enterprise



Amazon Web Services: Migrating your .NET Enterprise Application

ISBN: 978-1-849681-94-0

Paperback: 336 pages

Evaluate your Cloud requirements and successfully migrate your .NET Enterprise application to the Amazon Web Services Platform

1. Get to grips with Amazon Web Services from a Microsoft Enterprise .NET viewpoint
2. Fully understand all of the AWS products including EC2, EBS, and S3
3. Quickly set up your account and manage application security

Please check www.PacktPub.com for information on our titles

