

Join the discussion @ [p2p.wrox.com](http://p2p.wrox.com)



Wrox Programmer to Programmer™



# Professional iOS Programming

Foreword by Pertti Karlainen, *Product Manager, Northern Lights Software*

Peter van de Put



# iOS PROGRAMMING

---

FOREWORD .....	xix
INTRODUCTION .....	xxi
▶ PART I DEVELOPING A PROFESSIONAL UI	
CHAPTER 1 Creating a Personal Library .....	3
CHAPTER 2 Advancing with Tableviews .....	29
CHAPTER 3 Advancing with Map Kit.....	79
CHAPTER 4 Understanding Action Views and Alerts .....	119
CHAPTER 5 Internationalization: Building Apps for the World.....	141
CHAPTER 6 Using Multimedia .....	165
▶ PART II NETWORKING–DATA PROCESSING	
CHAPTER 7 Using Web Services and Parsing .....	213
CHAPTER 8 Using FTP .....	271
CHAPTER 9 Implementing Core Data .....	299
▶ PART III INTEGRATING YOUR APP	
CHAPTER 10 Notifications .....	335
CHAPTER 11 Sending E-Mail, SMS, and Dialing a Phone .....	355
CHAPTER 12 Understanding the Address Book .....	363
CHAPTER 13 Event Programming .....	385
CHAPTER 14 Integrating with Social Media .....	403
▶ PART IV TAKING YOUR APPLICATION TO PRODUCTION	
CHAPTER 15 Analyzing Your Application .....	435
CHAPTER 16 Monetize Your App .....	447
CHAPTER 17 Understanding iTunes Connect.....	481
CHAPTER 18 Building and Distribution .....	505
APPENDIX A Audio Codes.....	523
APPENDIX B Artwork Dimensions.....	527
INDEX .....	531



PROFESSIONAL

# iOS Programming

Peter van de Put



## **Professional iOS Programming**

Published by

John Wiley & Sons, Inc.  
10475 Crosspoint Boulevard  
Indianapolis, IN 46256  
[www.wiley.com](http://www.wiley.com)

Copyright © 2014 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-66113-0

ISBN: 978-1-118-66110-9 (ebk)

ISBN: 978-1-118-84425-0 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

**Limit of Liability/Disclaimer of Warranty:** The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit [www.wiley.com](http://www.wiley.com).

**Library of Congress Control Number:** 2013949552

**Trademarks:** Wiley, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

*This book is being dedicated to my wife, Miranda and  
daughter, Anique, whose continuous support and  
encouragement made it possible to write this book.*

# CREDITS

**ACQUISITIONS EDITOR**

Mary James

**PROJECT EDITOR**

Ed Connor

**TECHNICAL EDITOR**

Abhishek Mishra

**PRODUCTION EDITOR**

Christine Mugnolo

**COPY EDITOR**

Kim Cofer

**EDITORIAL MANAGER**

Mary Beth Wakefield

**FREELANCER EDITORIAL MANAGER**

Rosemarie Graham

**ASSOCIATE DIRECTOR OF MARKETING**

David Mayhew

**MARKETING MANAGER**

Ashley Zurcher

**BUSINESS MANAGER**

Amy Knies

**VICE PRESIDENT AND  
EXECUTIVE GROUP PUBLISHER**

Richard Swadley

**ASSOCIATE PUBLISHER**

Jim Minatel

**PROJECT COORDINATOR, COVER**

Katie Crocker

**COMPOSITOR**

Cody Gates, Happenstance Type-O-Rama

**PROOFREADERS**

Daniel Aull, Word One New York

Sarah Kaikini, Word One New York

**INDEXER**

John Sleeva

**COVER DESIGNER**

Ryan Sneed

**COVER IMAGE**

iStockphoto.com/bostb

## ABOUT THE AUTHOR

**PETER VAN DE PUT** is CEO and lead developer of YourDeveloper, a global software development company based in France. He began developing software in 1980 and delivered high-end software solutions for companies like Fuji Photo Film, Shell, Unilever, Bridgestone, Alcatel, Ricoh and many others. In 2006, he started a software company that focuses on developing iOS applications and backend software solutions and has developed applications for global clients such as banks, government agencies, telecommunications and utilities. In tandem with his software development career, he also trained hundreds of developers and co-founded several service companies and worked as a project manager and business consultant. By owning and directing a consultancy firm, he has seen all aspects of projects from planning to design to deployment to maintenance.

As an experienced trainer, he is also available to provide training classes for your iOS developers in countries all over the world.

## ABOUT THE TECHNICAL EDITOR

**ABHISHEK MISHRA** has been developing software for over 13 years and has experience with a diverse set of programming languages and platforms. He has worked on iOS projects for EURO RSCG, MusicQubed and is currently working as a lead iOS consultant for British Gas, part of Centrica PLC. Abhishek is the author of *iPhone and iPad App 24 Hour Trainer* (Wiley, 2010) and holds a master's degree in Computer Science from the University of London. He lives in London, and in his spare time works on developing a cross-platform game engine and animated short films with his wife.



# ACKNOWLEDGMENTS

**I WOULD LIKE TO START BY THANKING** my wife Miranda and my daughter Anique for their continuous support and encouragement during the writing process.

To Andre Smits, my closest friend, who's supported and encouraged me and who has done all the proofreading and provided very useful feedback.

To all my clients, who made it possible to obtain the experience level I've reached during the realization of their project.

To all my friends and followers who were interested during the writing of this book.



# CONTENTS

<b>FOREWORD</b>	<b>xix</b>
<b>INTRODUCTION</b>	<b>xxi</b>

## PART I: DEVELOPING A PROFESSIONAL UI

<b>CHAPTER 1: CREATING A PERSONAL LIBRARY</b>	<b>3</b>
Creating Your Personal Library	4
Understanding Project Basics	4
Starting a New Project	5
Configuring Your Project	6
Defining Constants	8
Using the Configuration	8
Importing the Header File	10
Registration—Login	11
Creating Registration Logic	12
Initializing Data	15
Initializing Application Defaults	15
Creating Login Logic	16
Securing Passwords	18
Storing the Password in a Keychain	20
Crash Management	20
Understanding Crashes	21
Implementing a Crash Handler	21
Summary	28
<b>CHAPTER 2: ADVANCING WITH TABLEVIEWS</b>	<b>29</b>
Understanding the UITableView	29
datasource and delegate	30
Scrolling	34
Building a Chat View Controller	38
Building a datasource	39
Building a Chat Data Object	39
Building a Custom UITableView	42
Flexible Cell Height	45
Developing Custom Cells	46

Creating the Chat User Object	49
Putting It All Together	50
Drilling Down with UITableView	56
Implementing a UISearchBar	66
Adding an Alphabet Index	72
Summary	77
<b>CHAPTER 3: ADVANCING WITH MAP KIT</b>	<b>79</b>
Simulating iOS Device Movement	80
Why You Need a GPS Simulator	80
Creating the Simulator	80
Creating a GPS Route File with Google Maps	84
Implementing the YDLocationSimulator	87
Working with Annotations	90
Creating Custom Annotations	90
Responding to Annotation Call-Outs	95
Clustering Annotations	100
Summary	118
<b>CHAPTER 4: UNDERSTANDING ACTION VIEWS AND ALERTS</b>	<b>119</b>
Asking for User Input	119
Creating a UIActionSheet with Multiple Options	120
Presenting the UIActionSheet	125
Responding to User Input	133
Processing the User Selection	133
Extending the UIAlertView	136
Adding a UITextField to a UIAlertView	136
Summary	140
<b>CHAPTER 5: INTERNATIONALIZATION: BUILDING APPS FOR THE WORLD</b>	<b>141</b>
Localizing Your Application	141
Setting Up Localization	143
Localizing Interface Builder Files	144
Localizing Strings	145
Localizing Images	148
Localize the Name of Your Application	150
Working with Date Formats	151
What Is a Locale?	151
Understanding Calendars	155
Storing Dates in a Generic Way	159

Working with Numbers	160
Introducing Number Formatters	160
Summary	164
<b>CHAPTER 6: USING MULTIMEDIA</b>	<b>165</b>
Portable Document Format	165
Displaying a PDF Document with a UIWebView	166
Displaying a PDF Document using QuickLook	170
Creating a Thumbnail from a PDF Document	173
Creating a PDF Document	177
Playing and Recording Audio	181
Introduction to the Frameworks	182
Playing an Audio File from the Bundle	182
Playing Audio from Your iTunes Library	187
Playing Streaming Audio	191
Recording Audio	193
Playing and Recording Video	198
Playing a Video File from the Bundle	199
Playing a Video from Your iTunes Library	202
Playing a YouTube Video	205
Recording Video	207
Summary	210

**PART II: NETWORKING–DATA PROCESSING**

<b>CHAPTER 7: USING WEB SERVICES AND PARSING</b>	<b>213</b>
Why Would You Need to Use a Web Service?	213
Understanding Basic Networking	214
Understanding Protocols	214
Understanding Operations	215
Understanding Response Codes	215
Introduction to Web Services	216
Calling an HTTP Service	216
Requesting a Website	216
Downloading an Image from an HTTP URL	219
Requesting a Secure Website Using HTTPS	225
Using Blocks	228
Calling a REST Service	232
Constructing Your Request	232
Processing the Response	236
Posting to a RESTful Service	242

Making SOAP Requests	248
Preparing Your Request	250
Passing Values to an Operation	252
Understanding Secure SOAP Requests	258
More Parsing	260
What about Comma-Separated Value Files?	260
Transforming XML to an NSDictionary	266
Summary	270
<b>CHAPTER 8: USING FTP</b>	<b>271</b>
Developing an FTP Client	272
Writing a Simple FTP Client	272
Downloading a Remote File	277
Creating a Remote Directory	279
Listing a Remote Directory	280
Uploading a File	283
Reading from an NSSet	284
Writing to an NSSet	285
Writing a Complex FTP Client	288
Working with an FTP Client	297
Summary	297
<b>CHAPTER 9: IMPLEMENTING CORE DATA</b>	<b>299</b>
Introduction to Core Data	299
Why Should You Use Core Data?	300
Introducing Managed Object Context	300
Introducing the Managed Object Model	300
Introducing Managed Objects	301
Introducing Persistent Stores	301
Introducing Fetch Requests	301
Using Core Data in Your Application	302
Creating a Managed Object Model	302
Creating Managed Objects	305
Creating Persistent Stores	306
Setting Up Your AppDelegate	307
Using Core Data in Your Application	309
Using Managed Objects	309
Fetching Managed Objects	311
Using Relationships	316
Understanding Model Changes	320
Tuning for Performance	324

Concurrency with Core Data	331
Summary	332
<b>PART III: INTEGRATING YOUR APP</b>	
<b>CHAPTER 10: NOTIFICATIONS</b>	<b>335</b>
Implementing Local Notifications	336
Understanding Local Notifications	336
Creating a Notification	337
Receiving a Notification	340
Understanding Push Notifications	341
Configuring the Developer Portal	343
Obtaining Certificates	346
Implementation with Urban Airship	349
External Notifications	352
Defining a Custom URL Scheme	352
Responding to the URL Request	353
Summary	354
<b>CHAPTER 11: SENDING E-MAIL, SMS, AND DIALING A PHONE</b>	<b>355</b>
Sending E-Mail	355
Composing an E-Mail	356
Working with Attachments	358
Sending SMS (Text Message)	359
Verifying if SMS Is Available	359
Composing a Text Message	359
Dialing a Phone Number	360
Verifying Dialing Capability	360
Summary	361
<b>CHAPTER 12: UNDERSTANDING THE ADDRESS BOOK</b>	<b>363</b>
Introduction to the Address Book Framework	363
Accessing the Address Book	364
Selecting a Contact	364
Requesting Access Permission	367
Displaying and Editing a Contact	370
Creating a Contact	373
Deleting a Contact	375
Programmatically Accessing the Address Book	375
Understanding Address Books	376
Understanding Records	379

Understanding Properties	380
Creating a Contact Programmatically	381
Deleting a Contact Programmatically	384
Summary	384
<b>CHAPTER 13: EVENT PROGRAMMING</b>	<b>385</b>
Introduction to the Event Kit Framework	385
Using the EventKitUI Framework	386
Requesting Access Permission	386
Accessing a Calendar	388
Creating and Editing a Calendar Event	390
Programmatically Accessing the Calendar Database	391
Creating an Event	392
Editing an Event	396
Deleting an Event	397
Stay Synchronized	397
Working with Reminders	397
Creating a Reminder	398
Editing a Reminder	399
Deleting a Reminder	399
Working with Alarms	399
Summary	401
<b>CHAPTER 14: INTEGRATING WITH SOCIAL MEDIA</b>	<b>403</b>
Introduction to Social Media Integration	403
Understanding the Accounts Framework	404
Understanding the Social Framework	408
Making a Post	409
Retrieving Tweets	418
Integrating with Facebook	419
Creating a Single Sign-In Application	426
Summary	431
<b>PART IV: TAKING YOUR APPLICATION TO PRODUCTION</b>	
<b>CHAPTER 15: ANALYZING YOUR APPLICATION</b>	<b>435</b>
Performing a Technical Analysis	435
Application Crashes	436
Blocking the Main Thread	436
Memory Leaks	437

Using Synchronized HTTP Requests	438
Extensive Bandwidth Usage	438
Battery Drainage	442
Bad User Interface	444
<b>Performing a Commercial Analysis</b>	<b>444</b>
Introducing Flurry Analytics	445
Summary	446
<b>CHAPTER 16: MONETIZE YOUR APP</b>	<b>447</b>
Introduction to Monetizing	447
Paid Application	448
Advertising	448
In-App Purchases	448
Subscriptions	448
Lead Generation	449
Affiliate Sales	449
Developing In-App Purchases	449
Introduction to In-App Purchase	449
Registering Products	450
Choosing the Product Type	450
Understanding the In-App Purchase Process	451
Implementing an In-App Purchase	452
Monetizing with Advertisements	473
Introducing the iAd Framework	473
Implementing the AdMob Network	476
Summary	479
<b>CHAPTER 17: UNDERSTANDING ITUNES CONNECT</b>	<b>481</b>
iOS Developer Member Center	482
Obtaining a Developer Certificate	482
Managing Devices	486
Managing Apps	489
Creating a Development Provisioning Profile	496
Creating a Distribution Provisioning Profile	499
Summary	503
<b>CHAPTER 18: BUILDING AND DISTRIBUTION</b>	<b>505</b>
App Store Review	505
Understanding the Review Guidelines	506
Understanding the Review Process	506

## CONTENTS

---

Understanding Rejections	509
Avoiding Common Pitfalls	509
Building for Ad Hoc Distribution	510
Building Your Application	510
Distribute for Testing	512
Building for App Store Distribution	514
Summary	521
<b>APPENDIX A: AUDIO CODES</b>	<b>523</b>
<hr/>	<hr/>
<b>APPENDIX B: ARTWORK DIMENSIONS</b>	<b>527</b>
Device Dimensions	527
iTunes Connect Artwork Dimensions	529
<b>INDEX</b>	<b>531</b>

# FOREWORD

Since the introduction of the App Store in 2008, more than 900,000 applications have been available with a total of 50 billion downloads (as of June 2013).

The effect of this phenomenon is simple: there is money to be made in the emerging market of mobile applications.

There was global media coverage when Steve Jobs unveiled the first iPhone with iOS in January 2007 at the Macworld Conference and Expo, and the release of the first iOS version in June 2007 generated a huge interest from traditional Apple fans. Developers were also interested in this new operating system with this amazing-looking smartphone built around the concept of user interaction using touches and gestures.

As more and more programming “how-to” videos became available on the Internet, people started copying and pasting bits and pieces of them to make their own applications without really knowing what they were doing. When their applications required functionality beyond the obvious, they started to run into problems.

Professional iOS Programming has been written with those developers in mind. It explains in great detail how to build professional iOS applications, step by step, by making use of available iOS frameworks using modern Objective-C programming principles. The 70 real-world sample programs included in the book (and available for download) are used to explain the functionalities, and you are free to use them as a starting point for your applications.

Peter van de Put, the author of this book, started his programming career in 1980 on the Sinclair ZX-81, a machine with a total memory of 16 KB that could be programmed in Assembler. In the years following, Peter learned many other programming languages, and moved from register-based programming in Assembler to object-oriented programming in Delphi, Java, C#, and C++.

In 2009, Peter began to study the technology behind iOS and the iPhone device, and mastered the Objective-C language.

Since then, he and his professional team have developed iPhone and iPad applications for clients in Australia, Europe, and the United States, where they deliver end-to-end solutions, from back-end management systems to iOS applications, using the latest modern programming practices and SDKs.

If you are an iOS developer with some experience and want to improve and extend your skills to be able to develop real-life iOS applications, or if you are a less experienced developer and frequently need to Google for answers on “How to ...” in relation to Objective-C or one of the iOS frameworks, this book is certainly worth adding to your library. Reading it will save you precious time.

“Professional iOS Programming will be mandatory literature for all our developers and we think it should also be for yours.”

—PERTTI KARJAINEN  
PRODUCT MANAGER, NORTHERN LIGHTS SOFTWARE  
[www.northernlightssoftware.com](http://www.northernlightssoftware.com)



# INTRODUCTION

**MY FIRST EXPERIENCE** with a computer was at the age of 15 at a technical high school in the Netherlands where I was introduced to the Sinclair ZX-81. It was a little computer with only 16 KB of memory, but it was amazing and I started programming on it from day one.

In 1981 when the hardware and software revolution started, I moved on to an Exidy Sourcer and the Commodore 64, soon followed by my first XT personal computer. I was intrigued by the fact that you could program these machines to do exactly what you wanted them to do, and I wanted to be able to develop programs for these systems.

I started programming in Assembler, and in time I learned to program in C, Basic, QuickBasic, Delphi, Pascal, Turbo Pascal, C++, Java, Microsoft VB.NET, Microsoft C#, and Objective-C. Programming became my profession, and I liked it.

However, because of the skills and knowledge I had gained, like many programmers I moved on to work as an analyst, software engineer, and consultant, and eventually ended up in a management position. I was doing less and less programming, which was the thing I liked the most.

In my professional career I've managed and co-founded several IT service companies and worked as a project manager, business consultant, and director, but always was able to find the time to do some software development during these projects.

During these years, I've delivered high-end software solutions for companies like Fuji Photo Film, Shell, Unilever, Bridgestone, Alcatel, Ricoh, and many others on all continents. I'm not a graphical designer, and designing an application's look is not my strongest point. My focus is always on optimization of code and exploring new technologies. Developing a high level of object-oriented code design with the lowest memory footprint is still a challenge.

In 2006, I decided to leave the hectic life of the Netherlands behind, and moved to France where I started a software company that focused on developing back-end software solutions and, a little later, on iOS application development.

Our professional team now develops iPhone and iPad applications for clients in Australia, Europe, and the United States, where we deliver end-to-end solutions, from back-end management systems to iOS applications, using the latest modern programming practices and SDKs.

## WHO THIS BOOK IS FOR

This book is written for both experienced and novice iOS developers who want to improve and extend their Objective-C programming skills.

A basic knowledge of the Xcode environment and the basics of Objective-C programming are required to understand the detailed, in-depth explanations and 70 programming samples.

This book is valuable to each and every iOS developer who wants to get a deeper understanding of the technologies involved in developing professional iOS applications. It can be used as a reference book to understand the details of the iOS SDK.

The detailed explanations of the programming techniques and the 70 sample applications also make this an ideal textbook for college professors and trainers.

## WHAT THIS BOOK COVERS

This book covers all the subjects required to develop professional iOS applications. All the explanations and code samples have been tested with and applied for the latest iOS 7 SDK.

The following subject areas are covered in this book:

- Creating a Personal Library
- Advancing UITableView
- Advancing with the Map Kit framework
- Understanding Actions and Alerts
- Internationalization
- Using Multimedia in your applications
- Using web services and XML-JSON parsing
- Developing an FTP client
- Implementing the Core Data framework
- Using notifications
- Implementing e-mail, SMS, and dialing functionality
- Understanding and integrating with the Address Book framework
- Programming the Event Kit framework
- Integrating the Social and Accounts framework
- Analyzing your application

- Monetizing your application and implementing the Store Kit framework
- Understanding certificates and profiles
- Building and distributing your application

## THE USE OF INTERFACE BUILDER, STORYBOARDS, AND USER INTERFACE DESIGN EXAMPLES

Each iOS programmer has a personal preference on how to create the user interface elements in their applications—one is not necessarily better than the other. In my own work, I create all user interface elements in code, as I appreciate the control and scalability it provides. Therefore, in most lessons in this book you will find Interface Builder files. Some developers prefer using Storyboards, for instance, and they can be used for any project in place of Interface Builder, should you so choose. The book should be of use to developers in either case.

## HOW THIS BOOK IS STRUCTURED

This book is divided into four parts:

- **Part I:** Developing a Professional User Interface
- **Part II:** Networking and Data Processing
- **Part III:** Integrating Your App
- **Part IV:** Taking Your App into Production

**Part I: Developing a Professional User Interface** consists of the following chapters:

- Chapter 1: Creating a Personal Library
- Chapter 2: Advancing with Table Views
- Chapter 3: Advancing with Map Kit
- Chapter 4: Understanding Actions Views and Alerts
- Chapter 5: Internationalization, Building Apps for the World
- Chapter 6: Using Multimedia

Chapter 1 starts with the development of a Personal Library class containing reusable code that you can use as the basis for each of your applications. During the chapters that follow, you extend the Personal Library class with more functionality.

Chapter 2 will teach you how to build astonishing table views. You will learn how to build a sectioned drill-down table view, implementing search.

Chapter 3 is all about the Map Kit framework. You will learn the concepts of the location manager and develop a GPS simulator for development purposes. You will develop clustered map views with custom annotations.

Action views and Alerts are the subject in Chapter 4. You will learn how to implement action views and alert views in your application to interact with the user of your application.

Chapter 5 will teach you how to internationalize your applications for the world by explaining localization techniques and explaining how to work with international dates and number formats.

Chapter 6 is all about multimedia. You will learn efficient techniques to display and create PDF documents, how to play and record audio and video using the different frameworks available.

**Part II: Networking and Data Processing** consists of the following chapters:

- Chapter 7: Using Web Services and Parsing
- Chapter 8: Using FTP
- Chapter 9: Implementing Core Data

Chapter 7 will teach you how to consume web services by REST or SOAP and how to GET and POST data to these web services. Parsing the response in XML and JSON is explained in detail to complete this chapter.

Chapter 8 will teach you how the File Transfer Protocol (FTP) fits in an iOS application architecture and how you can write a simple FTP client in Objective-C. For most advanced requirements this chapter also teaches you how to write an FTP command based on Objective-C class.

Chapter 9 will teach you all about the Core Data framework. It explains the Core Data concept, storage techniques, entities, relationships and the techniques to fetch data.

**Part III: Integrating Your App** consists of the following chapters:

- Chapter 10: Notifications
- Chapter 11: Sending E-Mail, SMS and Dialing a Phone
- Chapter 12: Understanding the Address Book
- Chapter 13: Event Programming
- Chapter 14: Integrating Social Media

Chapter 10 will teach you how to implement internal notification and external push notifications.

Chapter 11 will teach you how to send e-mail and text messages from within your application and how to dial phone numbers.

Chapter 12 will teach you how to read from and write to the contacts database by using the AddressBook framework. You will learn how to request access permissions and present a user interface for working with contacts.

Chapter 13 will teach you how to create and manage events and reminders from within your application.

Chapter 14 will teach you how to natively integrate your application with Facebook and Twitter.

Learn how to present a user's Tweets and Facebook wall posts and how to post to a Facebook wall and send a Tweet.

**Part IV: Taking Your App into Production** consists of the following chapters:

- Chapter 15: Analyzing your Application
- Chapter 16: Monetize your Application
- Chapter 17: Understanding iTunes Connect
- Chapter 18: Building and Distribution

Chapter 15 will teach you how to implement usage analysis in your application.

Chapter 16 will teach you how to monetize your application. This chapter contains an in-depth explanation and a helper class for In-App Purchase. Also the implementation of Advertisement frameworks like iAd and AdMob are covered in this chapter.

Chapter 17 will teach you how to work with iTunes Connect to prepare for your application's submission. Understanding provisioning profiles, certificates and devices is essential to be able to publish your application.

Chapter 18, the final chapter, will teach you how to build and distribute your application for either Ad-Hoc distribution or App Store distribution.

## WHAT YOU NEED TO USE THIS BOOK

To program iOS applications you need to download the latest version of Xcode with the latest iOS SDK included. You can download this at <http://developer.apple.com>.

## CONVENTIONS

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

**WARNING** *Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.*

**NOTE** *Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.*

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

We use a monofont type with no highlighting for most code examples.

We use bold to emphasize code that's particularly important in the present context.

## SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at <http://www.wrox.com>.

Specifically for this book, the code download is on the Download Code tab at:

[www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog)

You can also search for the book at [www.wrox.com](http://www.wrox.com) by ISBN (the ISBN for this book is 978-1-118-66113-0 to find the code. And a complete list of code downloads for all current Wrox books is available at [www.wrox.com/dynamic/books/download.aspx](http://www.wrox.com/dynamic/books/download.aspx).

Throughout each chapter, you'll find references to the names of code files as needed in listing titles and text.

Most of the code on [www.wrox.com](http://www.wrox.com) is compressed in a .ZIP, .RAR archive or similar archive format appropriate to the platform. Once you download the code, just decompress it with an appropriate compression tool.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at [www.wrox.com/dynamic/books/download.aspx](http://www.wrox.com/dynamic/books/download.aspx) to see the code available for this book and all other Wrox books.

## ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to <http://www.wrox.com> and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at [www.wrox.com/misc-pages/booklist.shtml](http://www.wrox.com/misc-pages/booklist.shtml).

If you don't spot "your" error on the Book Errata page, go to [www.wrox.com/contact/techsupport.shtml](http://www.wrox.com/contact/techsupport.shtml) and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

## P2P.WROX.COM

For author and peer discussion, join the P2P forums at [p2p.wrox.com](http://p2p.wrox.com). The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to [p2p.wrox.com](http://p2p.wrox.com) and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

**NOTE** *You can read messages in the forums without joining P2P but in order to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.



# PART I

## Developing a Professional UI

---

- ▶ **CHAPTER 1:** Creating a Personal Library
- ▶ **CHAPTER 2:** Advancing with Tableviews
- ▶ **CHAPTER 3:** Advancing with Map Kit
- ▶ **CHAPTER 4:** Understanding Action Views and Alerts
- ▶ **CHAPTER 5:** Internationalization: Building Apps for the World
- ▶ **CHAPTER 6:** Using Multimedia



# 1

## Creating a Personal Library

### WHAT'S IN THIS CHAPTER?

---

- Creating registering and login logic
- Configuring application settings
- Storing settings and securing password
- Handling crashes in your application

### WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 1 download and individually named according to the names throughout the chapter.

In this chapter you learn how to develop a Personal Library, which is a series of classes and techniques you can use to give you a head start on your projects. Creating a Personal Library will save you time during future projects. The Personal Library you will create during this chapter will teach you how to implement the logic for user registration, user login, and securing password storage.

If you've developed software before, you've likely noticed a lot of repetitive tasks in each application you've worked on.

This is no different when developing applications for iOS, and for that reason Xcode, the Apple Developer Platform, comes with different so-called project templates. The problem with these project templates is that they provide you only with some main object containers and don't provide out-of-the-box configurable features that you require in each application. In this chapter you develop your own configurable Personal Library that you can use for all your subsequent applications. Throughout the different chapters in this book, you extend the Personal Library with additional functionalities.

## CREATING YOUR PERSONAL LIBRARY

The Personal Library you are about to create is basically a project skeleton containing different classes and functionalities to implement repetitive tasks in a flexible and organized way. Functionalities in the Personal Library are:

- Managing configuration settings
- Creating a user registration process with a `UIViewController`
- Implementing a user login process with a `UIViewController`
- Storing user values and secured password storage in a `Keychain`

In your future development projects, you can just drag and drop the files of the Personal Library you require.

## Understanding Project Basics

When you develop iOS applications, you have to make some basic choices when you create a new project. Two important choices are:

- Whether to use Automatic Reference Counting (ARC)
- Whether to use Interface Builder for the UI composition, or to code the UI components in your controller classes

### Automatic Reference Counting

ARC is a technology introduced in iOS 5.x and higher that automatically manages your memory allocations. As the name suggests, it counts the references to objects and retains or releases objects automatically when required. The main advantages of using ARC are:

- No need to send `release` or `autorelease` messages to your objects
- Less chance for memory leaks in case you forget to `release` objects
- Less code to write because you can skip the `release` for your objects and also skip the `dealloc` method in most cases

The only reason for keeping a `dealloc` method around is when you need to free resources that don't fall under the ARC umbrella, such as:

- Calling `CFRelease` on Core Foundation objects
- Calling `free()` on memory you allocated with `malloc()`
- Invalidating a timer

When you are using source code and/or libraries from other parties, you must keep in mind that not all of them have been updated to support ARC. If you need to work with a class that doesn't support ARC and you still want to use ARC in your project, you can set a compiler flag for the file in question and assign it the value `-fno-objc-arc`, as shown in Figure 1-1.

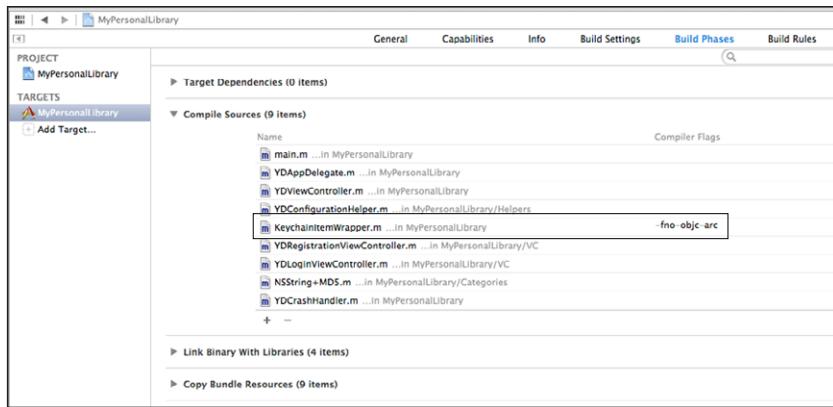


FIGURE 1-1

## Interface Builder

Because you are an experienced programmer, you know that Interface Builder is a part of Xcode that enables you to create and configure the UI elements of your project. Personally, I don't like working with Interface Builder because I like to have complete control over all aspects of the UI by means of code. The side effect, however, is that a large amount of my code is UI-related code. To keep the source code listings focused on the subject in most of the examples, Interface Builder is used to create the user interface and the Assistant Editor is used to create the `IBAction` and `IBOutlet`s.

## Starting a New Project

Start Xcode, create a new project using the Single View Application Project template, and name it `MyPersonalLibrary` using the configuration settings as shown in Figure 1-2.

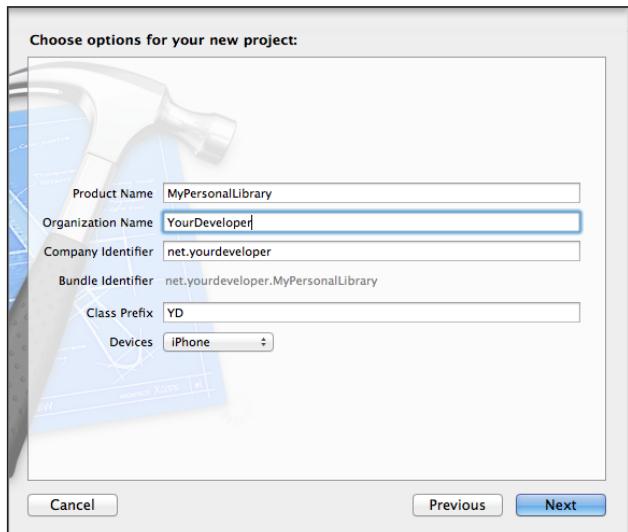


FIGURE 1-2

The Class Prefix field will help you to follow Apple’s coding guidelines because each class within your project needs to have a unique name. When you create the project you can use this field to define your default class prefix, which will be used when you are creating a new class. Although Apple’s guidelines advise using a three-character prefix, I always use a two-character prefix—the abbreviation of my company name, YourDeveloper—so I choose YD as a default prefix for my classes.

When you click the Next button, you will be asked to select a file location to store the project, and at the bottom of the screen you will see a checkbox that says Create Local git Repository for This Project, as shown in Figure 1-3. Check the option to create the local git repository and click the Create button to create the project with the chosen configuration.

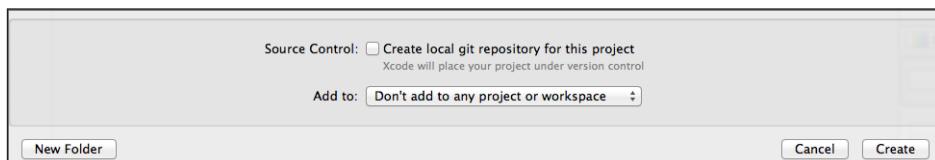


FIGURE 1-3

### **USING A LOCAL GIT REPOSITORY**

When you select the option to create a local git repository, Xcode creates this repository for you automatically, which enables you to use version control on your project. With version control, Xcode keeps track of changes in your source code, enabling you to revert to a version, commit changes, compare versions, and so on. If you are using a server based subversion system (SVN) to manage your source control you don’t require a local git repository since you will be using an SVN client or SVN commands to manage your sources. If you don’t use a server based subversion system I strongly recommend to check the Create Local git Repository option when creating a new project.

After the project is created, you’ll see two classes in the project: the `YDAppDelegate` class and the `YDViewController` class. Both classes are created by default by Xcode. You also see a `YDViewController.xib` file, which is an Interface Builder file.

## **Configuring Your Project**

It’s good practice to use groups in your project structure to organize different elements like images, sounds, classes, views, helpers, and so on, so start by creating some groups under the `MyPersonalLibrary` node. Create the following groups:

- Externals
- Categories
- CrashHandler
- Helpers

- Definitions
- Images
- VC

Because a group is only a container and not a physical folder on your filesystem, you should also create these folders on the filesystem under your project root folder.

Because there will be a lot of repetitive tasks in all the applications you develop, and you still need the flexibility to switch functionalities on and off without recoding and copying and pasting parts of code, create a configuration file as follows:

In the Project Explorer, navigate to the Definitions group and select New File from the context menu. Create a C header file called `YDConstants.h` as shown in Figures 1-4 and 1-5.

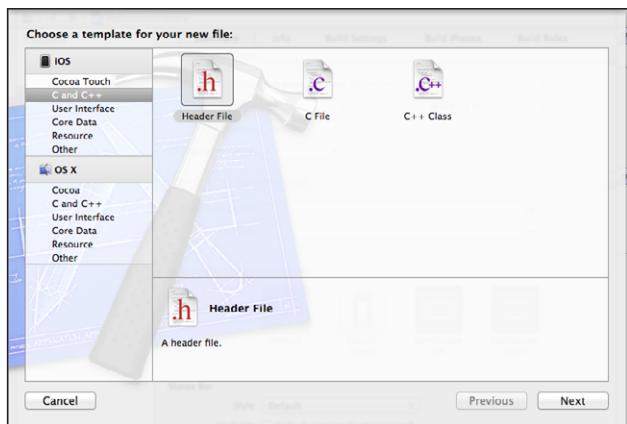


FIGURE 1-4

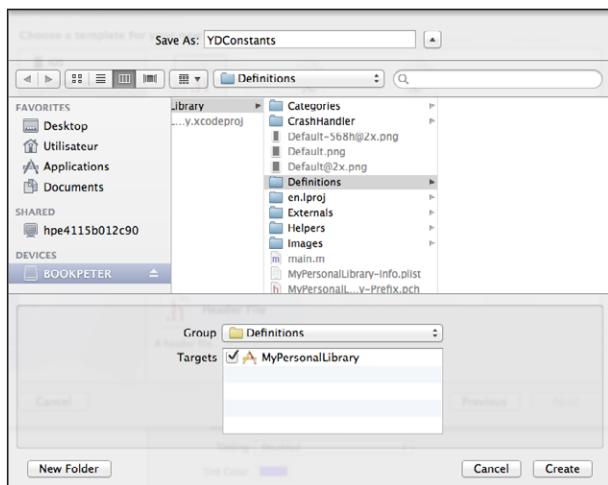


FIGURE 1-5

## Defining Constants

The Personal Library you are developing contains functionality for registration of a user, login of a user, and a settings controller. In addition to that you'll be implementing a crash handler.

Implement the code as shown in Listing 1-1 in your `YDConstants.h` file.

**LISTING 1-1:** Chapter1/MyPersonalLibrary/YDConstants.h

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

// My application switches
#define bYDActivateGPSOnStartUp YES
#define bYDRegistrationRequired YES
#define bYDLoginRequired NO
#define bYDShowLoginAfterRegistration YES
#define bYDInstallCrashHandler YES
//keys that are used to store data
#define bYDRegistered @"bYDRegistered"
#define bYDAuthenticated @"bYDAuthenticated"
#define bYDFirstLaunch @"bYDFirstLaunch"
#define bYDVibrate @"bYDVibrate"
```

## Using the Configuration

You can store settings using the `NSUserDefaults` class. Because you can store or change individual settings in many different places in your application, it's important to first synchronize the object to make sure modifications are presented with the correct value.

Next, create a helper class that will give you some static methods to read and write values using the `NSUserDefaults` class.

In the Project Explorer, navigate to the `Helpers` group and select `File ➔ New` from the context menu. Create a new Objective-C class named `YDConfigurationHelper` that inherits from `NSObject`, as shown in Listing 1-2.

**LISTING 1-2:** Chapter1/MyPersonalLibrary/YDConfigurationHelper.h

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

@interface YDConfigurationHelper : NSObject

+(void)setApplicationStartupDefaults;

+(BOOL)getValueForKey:(NSString *)_objectkey;

+(NSString *)getStringValueForKey:(NSString *)_objectkey;

+(void)setBoolValueForKey:(NSString *)_objectkey withValue:(BOOL)_boolvalue;
```

```
+ (void)setStringValueForConfigurationKey:(NSString *)  
    _objectkey withValue:(NSString *)_value;  
  
@end
```

The YDConfigurationHelper class in Listing 1-3 contains a few static methods to help you access the `NSUserDefaults` object. It contains get and set methods for `NSString` values and `BOOL` values to make life a lot easier.

**LISTING 1-3: Chapter1/MyPersonalLibrary/YDConfigurationHelper.m**

```
#import "YDConfigurationHelper.h"  
  
@implementation YDConfigurationHelper  
+ (void)setApplicationStartupDefaults  
{  
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];  
    [defaults synchronize];  
    [defaults setBool:NO forKey:kYDFirstLaunch];  
    [defaults setBool:NO forKey:kYDAuthenticated];  
    [defaults synchronize];  
}  
  
+ (BOOL)getBoolValueForConfigurationKey:(NSString *)_objectkey  
{  
    //create an instance of NSUserDefaults  
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];  
    [defaults synchronize]; //let's make sure the object is synchronized  
    return [defaults boolForKey:_objectkey];  
}  
  
+ (NSString *)getStringValueForConfigurationKey:(NSString *)_objectkey  
{  
    //create an instance of NSUserDefaults  
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];  
    [defaults synchronize]; //let's make sure the object is synchronized  
    if ([defaults stringForKey:_objectkey] == nil )  
    {  
        //I don't want a (null) returned  
        return @"";  
    }  
    else  
    {  
  
        return [defaults stringForKey:_objectkey];  
    }  
}  
+ (void)setBoolValueForConfigurationKey:(NSString *)  
    _objectkey withValue:(BOOL)_boolvalue  
{  
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];  
    [defaults synchronize]; //let's make sure the object is synchronized
```

*continues*

**LISTING 1-3 (continued)**

```

[defaults setBool:_boolvalue forKey:_objectkey];
[defaults synchronize];//make sure you're synchronized again
}

+(void)setStringValueForConfigurationKey:(NSString *)
    _objectkeyWithValue:(NSString *)_value
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults synchronize]; //let's make sure the object is synchronized
    [defaults setValue:_value forKey:_objectkey];
    [defaults synchronize];//make sure you're synchronized again
}

@end

```

The `getStringValueForConfigurationKey:` method is testing for a null value and returns an empty `NSString` instead of the null value. The reason is that in case you want to retrieve a value and assign it to the text property of a `UILabel`, you don't want to display (null).

## Importing the Header File

As you know, when you want to use a header definition file in your application you need to import it using the `#import "YDConstants.h"` statement.

This is not really convenient to repeat in each of your subsequent classes and ViewControllers. Each application, however, also has a precompiled header file that is applicable for the complete application. You can find this under the Supporting Files group in the Project Explorer; it is called `MyPersonalLibrary-Prefix.pch`.

If you import a header file here, it's globally available. Add the import statements here for the `YDConstants` and `YDConfigurationHelper` header files, as shown in Listing 1-4.

**LISTING 1-4: Chapter1/MyPersonalLibrary/MyPersonalLibrary-Prefix.pch**

```

#import <Availability.h>
#ifndef __OBJC__
    #import "YDConstants.h"
    #import "YDConfigurationHelper.h"
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
#endif

```

In the next section, you create a login, a registration, and a setting ViewController.

## REGISTRATION—LOGIN

Many applications require the user to register and log in, or log in only. If the user management is handled externally, you can ask a user to register in different ways. You can either ask for credentials like an e-mail address or a username and a password, or—what you see a lot nowadays—you can ask the user to register via Facebook.

Facebook registration is covered in depth in Chapter 14, “Social Media Integration,” and in that chapter, you will extend your Personal Library.

### Introducing the iOS Keychain Services

The iOS Keychain Services provide you with a secure storage solution for passwords, keys, certificates, notes, and custom data on the user’s device.

In your Personal Library you’ll be using the iOS Keychain Services to store the password the user enters during the registration process you’ll develop later.

To make interaction with the iOS Keychain Services more accessible, Apple has released a `KeyChainItemWrapper` class that will provide you with a higher-level API to work with the keychain.

You can download a sample project including the `KeyChainItemWrapper` class that you need in your Personal Library from [http://developer.apple.com/library/ios/#samplecode/GenericKeychain/Listings/Classes\\_KeychainItemWrapper\\_h.html](http://developer.apple.com/library/ios/#samplecode/GenericKeychain/Listings/Classes_KeychainItemWrapper_h.html).

Download the sample project from the URL and navigate to its files using Finder. In your Xcode project use the Project Explorer to navigate to the Externals group, select New Group from the context menu, and name it KeyChain. Create a folder named keychain under the `externals` folder on your filesystem. Copy the `KeychainItemWrapper.h` and `KeychainItemWrapper.m` files from the download folder to your projects folder by using drag-and-drop. Xcode will prompt you to choose the options for adding these files. Make the same choices as shown in Figure 1-6.

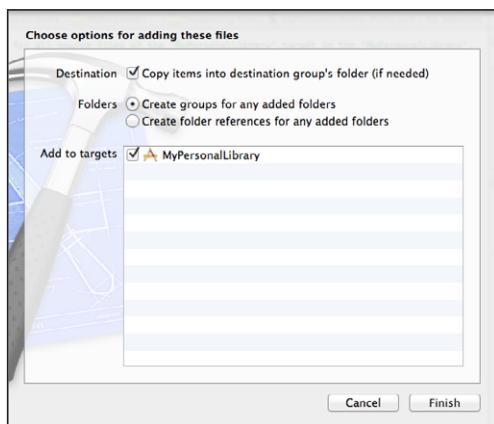


FIGURE 1-6

This `KeychainItemWrapper` class has not been developed for ARC, and for that reason you must set the `-fno -objc -arc` compiler flag for the `KeyChainItemWrapper.m` file as already shown in Figure 1-1.

For more information about KeyChain Services programming, please visit <https://developer.apple.com/library/ios/#documentation/security/Conceptual/keychainServConcepts/01introduction/introduction.html>.

To be able to work with the Keychain Services using the `KeychainItemWrapper` class, you also need to add a reference to the `Security.framework`.

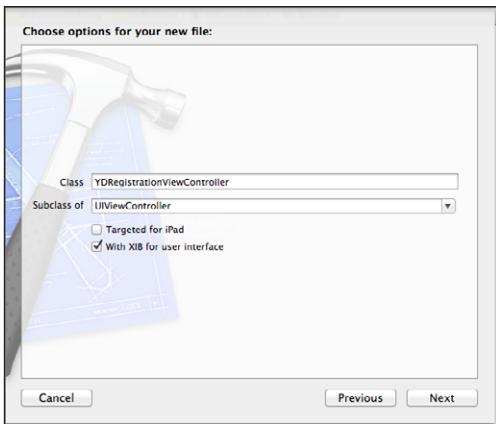
## Creating Registration Logic

Your user registration screen may look different in each application because the design of each application will be different; nevertheless, you can standardize a part of the registration logic because you'll always have to follow certain steps:

- Verify if the user has registered before so you don't need to present the registration view again.
- If the user has registered before, define what subsequent processes are required; for example, present a login view or load data from a web service.

You already have defined a constant called `bYDRegistrationRequired` in the `YDConstants` header file, which you can use in your application delegate to check if you need to present a `ViewController` to capture the registration credentials.

Using the Project Explorer, navigate to the VC group and select New File from the context menu to create a `UIViewController` subclass called `YDRegistrationViewController` as shown in Figure 1-7.



**FIGURE 1-7**

In this `YDRegistrationViewController` class, you need to set up a delegate that will notify the delegate ancestor with the result of the registration process. Use Interface Builder and the Assistant Editor to create a user interface with a `UILabel`, two `UITextField`s, and two `UIButton` objects. The `YDRegistrationViewController.xib` file will look like Figure 1-8.

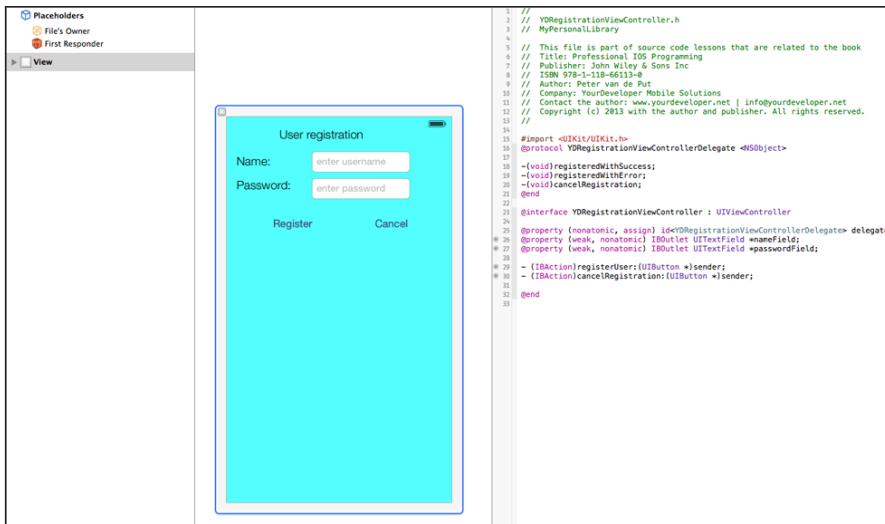


FIGURE 1-8

The YDRegistrationViewController.h code is shown in Listing 1-5.

#### LISTING 1-5: Chapter1/MyPersonalLibrary/YDRegistrationViewController.h

```

#import <UIKit/UIKit.h>
@protocol YDRegistrationViewControllerDelegate <NSObject>

- (void)registeredWithSuccess;
- (void)registeredWithError;
- (void)cancelRegistration;
@end

@interface YDRegistrationViewController : UIViewController

@property (nonatomic, assign) id<YDRegistrationViewControllerDelegate> delegate;
@property (weak, nonatomic) IBOutlet UITextField *nameField;
@property (weak, nonatomic) IBOutlet UITextField *passwordField;

- (IBAction)registerUser:(UIButton *)sender;
- (IBAction)cancelRegistration:(UIButton *)sender;

@end

```

In YDRegistrationViewController.m, implement the registerUser: and cancelRegistration: methods as shown in Listing 1-6.

The registerUser: method first hashes the entered password into an MD5 string and then stores it in the keychain.

**LISTING 1-6:** Chapter1/MyPersonalLibrary/YDRegistrationViewController.m

```

#import "YDRegistrationViewController.h"
#import "NSString+MD5.h"
#import "KeychainItemWrapper.h"
@interface YDRegistrationViewController : UIViewController

@end

@implementation YDRegistrationViewController
@synthesize delegate;
- (id)initWithNibName:(NSString *)NibNameOrNil bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)registerUser:(UIButton *)sender
{
    if (([self.nameField.text length]== 0 ) ||
        ([self.passwordField.text length] == 0))
    {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
            @"Error" message:@"Both fields are mandatory"
            delegate:self cancelButtonTitle:@"Ok"
            otherButtonTitles:nil, nil];
        [alert show];
    }
    else
    {

        KeychainItemWrapper* keychain = [[KeychainItemWrapper alloc]
            initWithIdentifier:@"YDAPPNAME" accessGroup:nil];
        [keychain setObject:self.nameField.text
            forKey:(__bridge id)kSecAttrAccount];
        [keychain setObject:[self.passwordField.text MD5]
            forKey:(__bridge id)kSecValueData];
        //reading back a value from the keychain for comparison
        //get username [keychain objectForKey:(__bridge id)kSecAttrAccount];
        //get password [keychain objectForKey:(__bridge id)kSecValueData];
    }
}

```

```

        [YDConfigurationHelper setBoolValueForConfigurationKey:
                     bYDRegisteredWithValue:YES];
        [self.delegate registeredWithSuccess];
        //or
        // [self.delegate registeredWithError];
    }
}

- (IBAction)cancelRegistration:(UIButton *)sender
{
    [self.delegate cancelRegistration];
}
@end

```

## Initializing Data

Each application requires some default settings that are applicable during the launch of the application. For that reason, in this section you learn how to initialize application defaults.

## Initializing Application Defaults

So far you've been working on creating a Personal Library that contains reusable and configurable code so you don't have to write the same code in each of your applications. You've been using the `NSUserDefaults` class via the `YDConfigurationHelper` class you defined to store settings and results of operations.

The first time your application launches, none of these values are set. To support the initialization of your defaults, the `YDConfigurationHelper` class has a static method called `setApplicationStartupDefaults`. In the implementation, you set the initial value of each property to control your application's logic at the first launch of an application, as in this example:

```

+(void)setApplicationStartupDefaults
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults synchronize];
    [defaults setBool:NO forKey:bYDFirstLaunch];
    [defaults setBool:NO forKey:bYDAuthenticated];
    [defaults synchronize];
}

```

You also may want to delete keychain items if they are left over from a previous install. This is also an ideal place if you need to create, for example, some special directories outside the bundle, so during further processing in your application you don't have to test each time if a directory exists.

The purpose of the `bYDFirstLaunch` key is to support the fact that this initialization code is being called only once.

The following code snippet explains how you should call this method in the `YDAppDelegate` `application: didFinishWithLaunchingOptions:` method. It tests if the stored value returns YES, and in that case it calls `setApplicationStartupDefaults`, which sets the value to NO.

```

if (! [YDConfigurationHelper getBoolValueForConfigurationKey:bYDFirstLaunch])
    [YDConfigurationHelper setApplicationStartupDefaults];

```

## Creating Login Logic

It is nice that a user can now register, but you also need to support the fact that a registered user can log in.

As a first step, create a new ViewController that serves as the ViewController for entering a user-name and a password. Create a new `YDLoginViewController` that inherits from `UIViewController` and code it like in Listing 1-7. Like in the `YDRegistrationViewController`, you define a protocol that can be used by your delegate so your delegate object will be notified with the result of the login process. Three methods are called depending on the result of the login process.

**LISTING 1-7:** Chapter1/MyPersonalLibrary/YDLoginViewController.h

```
#import <UIKit/UIKit.h>
@protocol YDLoginViewControllerDelegate <NSObject>

- (void)loginWithSuccess;
- (void)loginWithError;
- (void)loginCancelled;
@end

@interface YDLoginViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITextField *nameField;
@property (weak, nonatomic) IBOutlet UITextField *passwordField;

@property (nonatomic, assign) id<YDLoginViewControllerDelegate> delegate;

- (IBAction)loginUser:(UIButton *)sender;
- (IBAction)cancelLogin:(UIButton *)sender;

@end
```

The implementation of the `YDLoginViewController` is shown in Listing 1-8.

**LISTING 1-8:** Chapter1/MyPersonalLibrary/YDLoginViewController.m

```
#import "YDLoginViewController.h"
#import "YDLoginViewController.h"
#import "NSString+MD5.h"
#import "KeychainItemWrapper.h"
@interface YDLoginViewController : UIViewController

@implementation YDLoginViewController
@synthesize delegate;
- (id)initWithNibName:(NSString *)NibNameOrNil bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self)
        self.delegate = delegate;
    return self;
}
```

```

        if (self) {
            // Custom initialization
        }
        return self;
    }

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)loginUser:(UIButton *)sender
{
    if ([[self.nameField.text length]== 0 ) ||
        ([self.passwordField.text length] == 0))
    {
        [self showErrorWithMessage:@"Both fields are mandatory!"];
    }
    else
    {
        KeychainItemWrapper* keychain = [[KeychainItemWrapper alloc]
                                         initWithIdentifier:@"YDAPPNAME" accessGroup:nil];
        if ([self.nameField.text isEqualToString:
             [keychain objectForKey:(__bridge id)kSecAttrAccount]])
        {
            if ([[self.passwordField.text MD5]
                 isEqualToString:[keychain objectForKey:
                               (__bridge id)kSecValueData]])
            {
                [self.delegate loginWithSuccess];
            }
            else
                [self showErrorWithMessage:@"Password not correct."];
        }
        else
            [self showErrorWithMessage:@"Name not correct."];
    }
}

- (IBAction)cancelLogin:(UIButton *)sender
{
    [self.delegate loginCancelled];
}

-(void)showErrorWithMessage:(NSString *)msg
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error"

```

*continues*

**LISTING 1-8 (continued)**

```

        message:msg delegate:self
        cancelButtonTitle:@"Ok"
        otherButtonTitles:nil, nil];
    [alert show];
}
@end

```

In the `loginUser:` method you see that the first check performed is if both fields have been entered with data. Next, the logic checks if the username is stored and, if so, if the password stored in the keychain is equal to the one that has been entered.

The `YDLoginViewController.xib` file should look something like Figure 1-9.

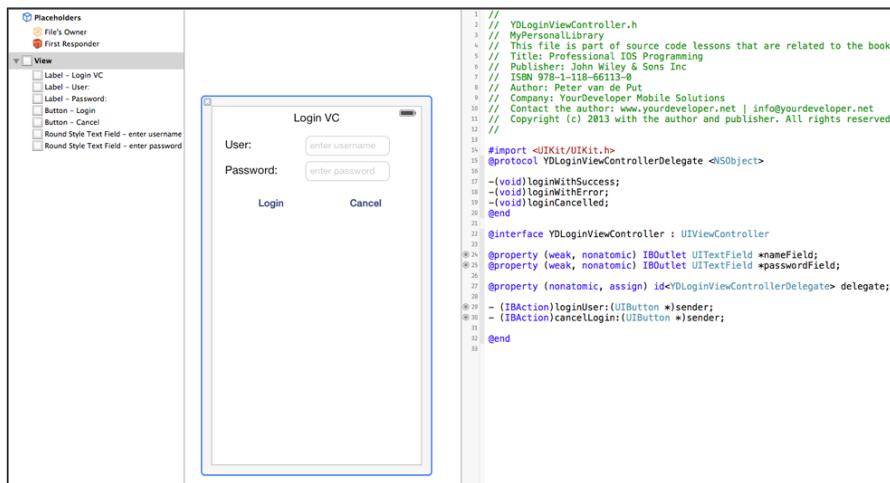


FIGURE 1-9

## Securing Passwords

In your `YDRegistrationViewController`, you accept a value for the username and for the password and store it in the keychain of the device. Many developers still store the username and password using the `NSUserDefaults` class, which is not a secure solution.

This is not a very secure solution because the password is stored in plaintext, and the `NSUserDefaults` object is not 100 percent safe and can be hacked.

To make your Personal Library more secure, implement a category of `NSString` that creates an MD5 hashed value of the password and store it in the keychain; the only safe place on the device to store critical data.

In the Project Explorer, navigate to the Categories group and select New File from the context menu. Select Objective-C Category from the template list as shown in Figure 1-10.

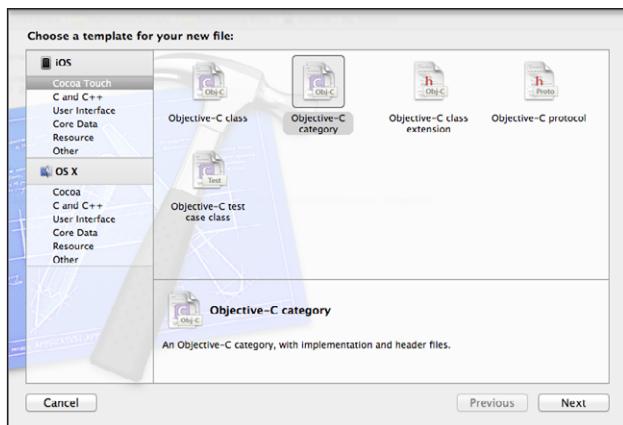


FIGURE 1-10

Click the Next button, and in the next screen enter **MD5** as the name of the category you are creating. From the Category On drop-down list, select **NSString** as shown in Figure 1-11.

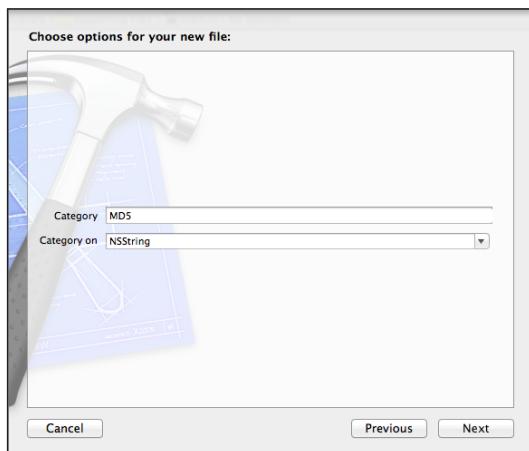


FIGURE 1-11

The header file is shown in Listing 1-9.

#### LISTING 1-9: Chapter1/MyPersonalLibrary/NSString+MD5 header

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

@interface NSString (MD5)
- (NSString *)MD5;
@end
```

The implementation of your category is shown in Listing 1-10 and the explanation is in the code.

**LISTING 1-10: Chapter1/MyPersonalLibrary /NSString+MD5 implementation**

```
#import "NSString+MD5.h"
#import <CommonCrypto/CommonDigest.h>
@implementation NSString (MD5)
- (NSString*)MD5
{
    // Create pointer to the string as UTF8
    const char *ptr = [self UTF8String];

    // Create byte array of unsigned chars
    unsigned char md5Buffer[CC_MD5_DIGEST_LENGTH];

    // Create 16 bytes MD5 hash value, store in buffer
    CC_MD5(ptr, strlen(ptr), md5Buffer);

    // Convert unsigned char buffer to NSString of hex values
    NSMutableString *output = [NSMutableString
        stringWithCapacity:CC_MD5_DIGEST_LENGTH * 2];
    for(int i = 0; i < CC_MD5_DIGEST_LENGTH; i++)
        [output appendFormat:@"%@", md5Buffer[i]];

    return output;
}
@end
```

In this category you import the `<CommonCrypto/CommonDigest.h>` header file. To do this, you must add the `Security.framework` to your project if you haven't already done that when adding the `KeychainItemWrapper` class.

## Storing the Password in a Keychain

Apple has developed a keychain wrapper class that makes it easy for you to work with the keychain.

The `KeychainItemWrapper` classes are part of the download of this book and are taken directly from Apple's developer portal. This `KeychainItemWrapper` class has not been developed for ARC, and for that reason you must set the `-fno -objc -arc` compiler flag for the `KeyChainItemWrapper.m` file as shown earlier in Figure 1-1.

## CRASH MANAGEMENT

Despite all your programming efforts and extensive test processes, your application still might crash. There can be many different reasons for an application to crash, and sometimes it may not even relate to your application at all. In this section you learn about the kinds of crashes that happen and how to implement a crash handler that shows a `UIAlertView` with a standard message just before the app crashes, so your user knows the application is crashing and might be less irritated.

The two main reasons for a crash are:

- Memory leaks
- Over-releasing objects

Memory leaks are most often caused by not releasing your objects. `UIImageView`, `UIImage`, and `UIWebView` are consuming a lot of memory; they definitely need to be released at the right time.

Over-releasing happens when you try to access an object that is already released.

### **USING INSTRUMENTS**

Instruments is a performance, analysis, and testing tool for dynamically tracing and profiling iOS and OS X code. It's important to have a good understanding of Instruments and how it will help you in improving the quality of your code, by identifying memory leaks, giving you insight into memory consumption, and many other performance-relevant measures. Please visit <http://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html> for the complete documentation on Instruments.

You can use Instruments and profile your application to identify memory leaks. You should use the Zombie instrument to identify access to an object that is already released (also known as a zombie).

## **Understanding Crashes**

At run time, the application's signal handling can, by default, launch six different standard signals in case an application crashes. These signals are:

- **SIGABRT:** An abnormal termination
- **SIGFPE:** A floating-point exception
- **SIGILL:** An invalid instruction
- **SIGINT:** An interactive attention request sent to the application
- **SIGSEGV:** Access to an invalid memory address
- **SIGTERM:** Termination request sent to the application

In the next section, you build a global exception handler that captures the signal and presents a `UIAlertView` to notify the user.

## **Implementing a Crash Handler**

Create a new class called `YDCrashHandler` as shown in Listing 1-11.

**LISTING 1-11:** Chapter1/MyPersonalLibrary/YDCrashHandler.h

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

@interface YDCrashHandler : NSObject
{
    BOOL dismissed;
}
void InstallCrashExceptionHandler();
@end
```

The implementation of the class is shown in Listing 1-12.

The class installs an `NSSetUncaughtExceptionHandler` for each of the available signals. If an exception occurs, the handler is invoked and accesses all modes in the current RunLoop. The process is killed and the back trace is read into an `NSDictionary` and passed to the `handleException` method that presents a `UIAlertView`.

**LISTING 1-12:** Chapter1/MyPersonalLibrary/YDCrashHandler.m

```
#import "YDCrashHandler.h"
#include <libkern/OSAtomic.h>
#include <execinfo.h>

NSString * const YDCrashHandlerSignalExceptionName =
    @"YDCrashHandlerSignalExceptionName";
NSString * const YDCrashHandlerSignalKey = @"YDCrashHandlerSignalKey";
NSString * const YDCrashHandlerAddressesKey = @"YDCrashHandlerAddressesKey";

volatile int32_t UncaughtExceptionCount = 0;
const int32_t UncaughtExceptionMaximum = 10;

const NSInteger UncaughtExceptionHandlerSkipAddressCount = 4;
const NSInteger UncaughtExceptionHandlerReportAddressCount = 5;
@implementation YDCrashHandler
+ (NSArray *)backtrace
{
    void* callstack[128];
    int frames = backtrace(callstack, 128);
    char **strs = backtrace_symbols(callstack, frames);

    int i;
    NSMutableArray *backtrace = [NSMutableArray arrayWithCapacity:frames];
    for (
        i = UncaughtExceptionHandlerSkipAddressCount;
        i < UncaughtExceptionHandlerSkipAddressCount +
            UncaughtExceptionHandlerReportAddressCount;
        i++)
    {
        [backtrace addObject:[NSString stringWithUTF8String:strs[i]]];
    }
    free(strs);
}
```

```

        return backtrace;
    }

- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)anIndex
{
    //if (anIndex == 0)
    //{
    dismissed = YES;
    //}
}

- (void)handleException:(NSError * )exception
{
    UIAlertView *thisAlert = [[UIAlertView alloc] initWithTitle:@"Sorry"
message:@"An unexpected event happened causing the application to
shutdown." delegate:nil cancelButtonTitle:@"Ok"
otherButtonTitles:nil, nil];

[thisAlert show];

CFRunLoopRef runLoop = CFRunLoopGetCurrent();
CFArrayRef allModes = CFRunLoopCopyAllModes(runLoop);

while (!dismissed)
{
    for (NSString *mode in (NSArray *)CFBridgingRelease(allModes))
    {
        CFRunLoopRunInMode((CFStringRef)CFBridgingRetain(mode), 0.001, false);
    }
}

CFRelease(allModes);

NSSetUncaughtExceptionHandler(NULL);
signal(SIGABRT, SIG_DFL);
signal(SIGILL, SIG_DFL);
signal(SIGSEGV, SIG_DFL);
signal(SIGFPE, SIG_DFL);
signal(SIGBUS, SIG_DFL);
signal(SIGPIPE, SIG_DFL);

if ([[exception name] isEqualToString:YDCrashHandlerSignalExceptionName])
{
    kill(getpid(), [[[exception userInfo]
objectForKey:YDCrashHandlerSignalKey] intValue]);
}
else
{
    [exception raise];
}
}

```

*continues*

**LISTING 1-12 (continued)**

```
        }
    }

@end

void HandleException(NSException *exception)
{
    int32_t exceptionCount = OSAtomicIncrement32(&UncaughtExceptionCount);
    if (exceptionCount > UncaughtExceptionMaximum)
    {
        return;
    }

    NSArray *callStack = [YDCrashHandler backtrace];
    NSMutableDictionary *userInfo =
    [NSMutableDictionary dictionaryWithDictionary:[NSDictionary dictionaryWithObject:exception forKey:@"exception"]];
    [userInfo
     setObject:callStack
     forKey:YDCrashHandlerAddressesKey];

    [[[YDCrashHandler alloc] init]
     performSelectorOnMainThread:@selector(handleException:)
     withObject:
     [NSEXception
      exceptionWithName:[exception name]
      reason:[exception reason]
      userInfo:userInfo]
     waitUntilDone:YES];
}

void SignalHandler(int signal)
{
    int32_t exceptionCount = OSAtomicIncrement32(&UncaughtExceptionCount);
    if (exceptionCount > UncaughtExceptionMaximum)
    {
        return;
    }

    NSMutableDictionary *userInfo =
    [NSMutableDictionary
     dictionaryWithObject:[NSNumber numberWithInt:signal]
     forKey:YDCrashHandlerSignalKey];

    NSArray *callStack = [YDCrashHandler backtrace];
    [userInfo
     setObject:callStack
     forKey:YDCrashHandlerAddressesKey];

    [[[YDCrashHandler alloc] init]
     performSelectorOnMainThread:@selector(handleException:)
     withObject:
     [NSEXception
```

```

exceptionWithName:YDCrashHandlerSignalExceptionName
reason:
[NSString stringWithFormat:@"Signal %d was raised.", signal]
userInfo:
[NSDictionary
 dictionaryWithObject:[NSNumber numberWithInt:signal]
 forKey:YDCrashHandlerSignalKey]]
waitUntilDone:YES];
}

void InstallCrashExceptionHandler()
{
    NSSetUncaughtExceptionHandler(&HandleException);
    signal(SIGABRT, SignalHandler);
    signal(SIGILL, SignalHandler);
    signal(SIGSEGV, SignalHandler);
    signal(SIGFPE, SignalHandler);
    signal(SIGBUS, SignalHandler);
    signal(SIGPIPE, SignalHandler);
}

```

Your Personal Library is now ready to be used. Open your `YDAppDelegate.h` file and implement the code as shown in Listing 1-13.

#### LISTING 1-13: Chapter1/MyPersonalLibrary/YDAppDelegate.h

```

#import <UIKit/UIKit.h>
#import "YDRegistrationViewController.h"
#import "YDLoginViewController.h"

@class YDViewController;
@interface YDAppDelegate : UIResponder <UIApplicationDelegate,
    YDLoginViewControllerDelegate,
    YDRegistrationViewControllerDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) YDViewController *viewController;
@property (strong, nonatomic) YDLoginViewController *loginVC;
@property (strong, nonatomic) YDRegistrationViewController *registrationVC;

@end

```

Open your `YDAppDelegate.m` implementation file and write the code as shown in Listing 1-14.

The application: `didFinishLaunchingWithOptions:` method first checks if the `YDCrashHandler` needs to be installed. Next, it checks if the application runs for the first time, and if so, it sets the application startup defaults using the `YDConfigurationHelper` class.

Next, it follows a logic that tests whether or not registration is required. If the previous test returns true, a second test is performed to see if a user registration has been executed before. If registration

needs to be done, the `YDRegistrationViewController` is presented and the delegate implementation checks if the `YDLoginViewController` needs to be presented. After the login process has completed with success, the `YDMainViewController` is presented. The complete implementation is shown in Listing 1-14.

**LISTING 1-14:** Chapter1/MyPersonalLibrary/YAppDelegate.m

```
#import "YAppDelegate.h"

#import "YDViewController.h"
#import "YDCrashHandler.h"
@implementation YAppDelegate

- (void)installYDCrashHandler
{
    InstallCrashExceptionHandler();
}
- (BOOL)application:(UIApplication *)application
 didFinishLaunchingWithOptions:
 (NSDictionary *)launchOptions
{
    if (bYDInstallCrashHandler)
    {
        [self performSelector:@selector(installYDCrashHandler)
 withObject:nil afterDelay:0];
    }

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

    if (!+[YDConfigurationHelper getBoolValueForConfigurationKey:bYDFirstLaunch])
        [YDConfigurationHelper setApplicationStartupDefaults];

    if (bYDActivateGPSOnStartUp)
    {
        //Start your CLLocationManager here if you're application needs the GPS
    }

    if (bYDRegistrationRequired && !+[YDConfigurationHelper
        getBoolValueForConfigurationKey:bYDRegistered])
    {
        //Create an instance of your RegistrationViewController
        self.registrationVC =[[YDRegistrationViewController alloc] init];
        //Set the delegate
        self.registrationVC.delegate=self;
        self.window = [[UIWindow alloc] initWithFrame:
                      [[UIScreen mainScreen] bounds]];
        self.window.rootViewController = _registrationVC;
        self.window.backgroundColor = [UIColor clearColor];
        [self.window makeKeyAndVisible];
    }
}
```

```

        else
        {
            // you arrive here if either the registration is not required
            // or yet achieved
            if (bYDLoginRequired)
            {
                self.loginVC= [[YDLoginViewController alloc] init];
                self.loginVC.delegate=self;
                self.window = [[UIWindow alloc] initWithFrame:
                               [[UIScreen mainScreen] bounds]];
                self.window.rootViewController = _loginVC;
                self.window.backgroundColor = [UIColor clearColor];
                [self.window makeKeyAndVisible];
            }
            else
            {
                self.viewController= [[YDViewController alloc] init];
                self.window.rootViewController =self.viewController;
                [self.window makeKeyAndVisible];
            }
        }

    }

#pragma Registration Delegates
-(void)registeredWithError
{
    //called from RegistrationViewcontroller if registration failed
}
-(void)registeredWithSuccess
{
    //called from RegistrationViewcontroller if the registration with success
    //
    if (bYDShowLoginAfterRegistration)
    {
        self.loginVC = [[YDLoginViewController alloc] init];
        self.loginVC.delegate=self;
        self.window = [[UIWindow alloc] initWithFrame:
                      [[UIScreen mainScreen] bounds]];
        self.window.rootViewController = self.loginVC;
        self.window.backgroundColor = [UIColor clearColor];
        [self.window makeKeyAndVisible];
    }
    else
    {
        self.viewController= [[YDViewController alloc] init];
        self.window.rootViewController =self.viewController;
        [self.window makeKeyAndVisible];
    }
}
-(void)cancelRegistration
{
    //called from RegistrationViewcontroller if cancel is pressed
}

```

*continues*

**LISTING 1-14** (*continued*)

```
#pragma Login delegates
- (void)loginWithSuccess
{
    //called when login with success
    self.viewController= [[YDViewController alloc] init];
    self.window.rootViewController =self.viewController;
    [self.window makeKeyAndVisible];
}
- (void)loginWithError
{
    //called when login with error
}
- (void)loginCancelled
{
    //called when login is cancelled
}

@end
```

## SUMMARY

In this chapter you created a configurable Personal Library that you can use as a starting point for your applications. This Personal Library supports the following features:

- A single location for configuration settings
- A registration process with a ViewController
- A login process with a ViewController
- Storing values in user defaults
- Securing password storage by MD5 encryption and keychain storage
- A Settings ViewController that enables you to use checkboxes to retrieve and store application settings
- A crash handler to avoid unexpected crashes of your application

In the next chapter, you learn how to advance your applications using UITableView objects and customize them to your needs. After you understand the basics for working with UITableView objects, you develop a custom UITableView solution that results in a chat view controller, like the one used in iMessage. Finally, you develop a custom UITableView that supports drill-down functionality to improve the user's experience.

# 2

## Advancing with Tableviews

### WHAT'S IN THIS CHAPTER?

---

- Learning how to advance with the `UITableView` to give your applications a professional look and feel.
- Learning how to develop your own custom `UITableView` class that mimics the look and feel of the iMessage application.
- Implementing a drill-down logic for a grouped `UITableView`.

### WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 2 download and individually named according to the names throughout the chapter.

The `UITableView` is probably the most-used user interface object for presenting data in an iOS application. In this chapter you learn to use the `UITableView` beyond the standard implementation and understand how the `UITableView` works. You create a chat view controller that supports custom cells and flexible row height, a drill-down implementation that groups categories of objects to make a professional user interface, and finally, you add search capabilities to your table view implementation.

### UNDERSTANDING THE UITABLEVIEW

`UITableView` inherits directly from `UIScrollView`, giving it direct scrolling capabilities. When you want to use a `UITableView` you first have to create an instance of `UITableView`, position it on your `UIView` to make it visible, and set up a datasource and a delegate that takes care of the interaction with the `UITableView`.

## datasource and delegate

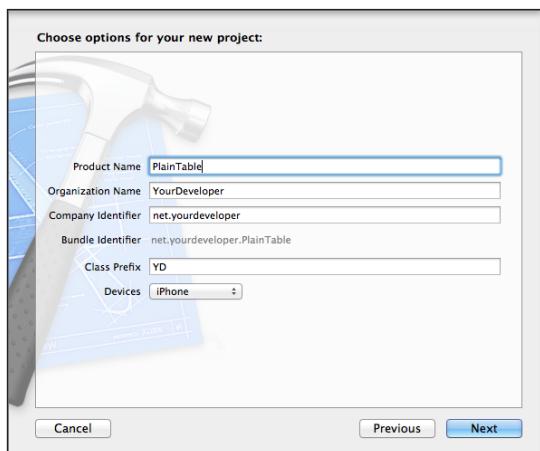
Each UITableView needs both a datasource and a delegate. The datasource supplies data to the UITableView. Typically the datasource uses an NSArray or an NSDictionary to store data internally and present it to the table view as needed. The delegate must implement the UITableViewDelegate and UITableViewDataSource protocol.

The UITableViewDelegate protocol defines several methods of which a minimum of three must be implemented by the delegate object.

The mandatory delegate methods to implement are:

- `tableview:numberOfRowsInSection:`
- `numberOfSectionsInTableView:`
- `tableview:cellForRowAtIndexPath:`

Start Xcode and create a new project using the Single View Application Project template, and name it PlainTable using the configuration shown in Figure 2-1.



**FIGURE 2-1**

Open the YDViewController.xib file using Interface Builder and add a UITableView to the window. Use the Assistant Editor to create a property for the UITableView. You also need to set the Referencing Outlets, datasource, and delegate to the UITableView object. Make sure your YDViewController.xib file looks like Figure 2-2.

Open the YDViewController.h file and create an NSMutableArray named `rowData` to act as a datasource as shown in Listing 2-1.

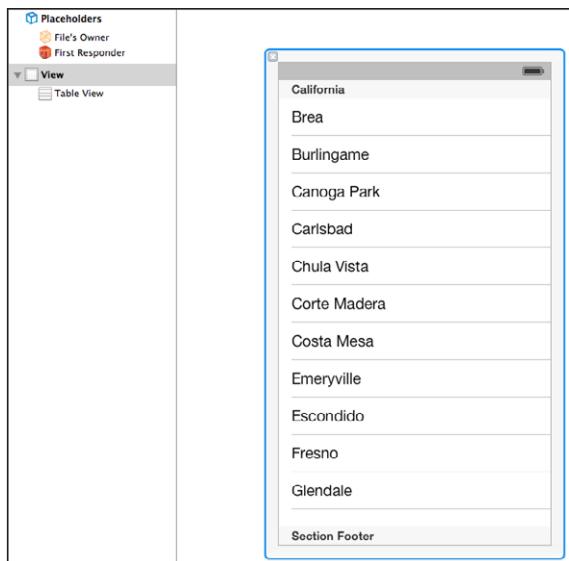


FIGURE 2-2

**LISTING 2-1:** Chapter2/PlainTable/YDViewController.h

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

@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;
@property(nonatomic,strong) NSMutableArray* rowData;

@end
```

Open the YDViewController and implement the code as shown in Listing 2-2, which is explained in detail after the code listing.

**LISTING 2-2:** Chapter2/PlainTable/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()
```

@end

```
@implementation YDViewController
```

- (void)viewDidLoad

{

[super viewDidLoad];  
// Do any additional setup after loading the view, typically from a nib.

*continues*

**LISTING 2-2 (continued)**

```

        [self loadData];
    }
    -(void)loadData
    {
        if (self.rowData!=nil)
        {
            [self.rowData removeAllObjects];
            self.rowData=nil;
        }
        self.rowData = [[NSMutableArray alloc] init];
        for (int i=0 ; i<100;i++)
        {
            [self.rowData addObject:[NSString stringWithFormat:@"Row: %i",i]];
        }
        //now my datasource is populated let's reload the tableview
        [self.mTableView reloadData];
    }
    #pragma mark UITableView delegate
    - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
        return 1;
    }

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

        return [self.rowData count];
    }
    - (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
        static NSString *CellIdentifier = @"Cell";
        UITableViewCell *cell = (UITableViewCell *)[tableView
            dequeueReusableCellWithIdentifier:CellIdentifier];
        if (cell == nil) {
            cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier];
        }
        cell.selectionStyle = UITableViewCellSelectionStyleNone;
        cell.textLabel.text = [self.rowData objectAtIndex:indexPath.row];
        return cell;
    }
    -(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
        (NSIndexPath *)indexPath
    {
        [tableView deselectRowAtIndexPath:indexPath animated:YES];
    }
    -(void)didReceiveMemoryWarning
    {
        [super didReceiveMemoryWarning];
        // Dispose of any resources that can be recreated.
    }
}

@end

```

Here's the breakdown of this code so you understand what's being done.

In the `viewDidLoad` method, the local method `loadData` is called, which creates an `NSMutableArray` with 100 entries and sends the `reloadData` message to `self.tableView`.

This `reloadData` method forces the `mTableView` object to reload its data and update its user interface by calling the `delegate` methods.

After the `#pragma mark UITableViewDelegate` marker, you implement the minimum set of delegates that are required for the table view to operate.

`numberOfSectionsInTableView:` is the delegate method that is called to determine the number of sections in a `UITableView`. If you are using a `UITableView` with the `UITableViewStylePlain`, the number of sections is always 1. If you are using sections, as you will learn in the drill-down example, you return the number of sections.

`tableView:cellForRowAtIndexPath:` is the delegate method that is called when the cell needs to be rendered. This is exactly the location where you lay out your `UITableViewCell` (a row in a `UITableView`). For now you are simply creating a `UITableViewCell` and trying to reuse it from memory if it's still available.

To display the correct row from your `rowData` array, you assign to `cell.textLabel.text` the value of `[rowData objectAtIndex :indexPath.row];`.

`tableView:didSelectRowAtIndexPath:` is the delegate method that is called when a user selects a row by tapping on the row. The `deselectRowAtIndexPath:animated:` delegate method deselects the row so it doesn't remain highlighted.

If you want to keep the selection visible, you omit this code.

When you run the application, it looks like Figure 2-3.

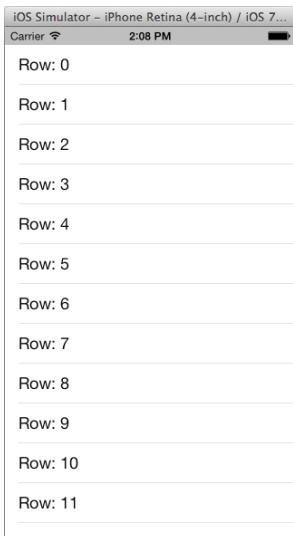


FIGURE 2-3

## Scrolling

Because the `UITableView` object inherits from `UIScrollView`, it has full scrolling capabilities in itself. However, in some cases, for example when you are adding a new row in the `UITableView` or deleting a row, you might want to scroll directly to a certain position in the `UITableView`.

You can achieve code-based scrolling in a `UITableView` by calling the `scrollToRowAtIndexPath:atScrollPosition:animated:` method of the `UITableView`. The first argument this method takes is an object of type `NSIndexPath`. An `NSIndexPath` object represents a path to a specific node in a tree of nested array collection. This path is known as an index path. In iOS the `NSIndexPath` object is used to identify the path to a row and section within a table view. You can create an instance of `NSIndexPath` by calling the `indexPathForRow:inSection:` method of the `NSIndexPath` class and pass the row and section index numbers.

Start Xcode and create a new project using the Single View Application Project template, and name it `ScrollingTable` using the configuration shown in Figure 2-4.

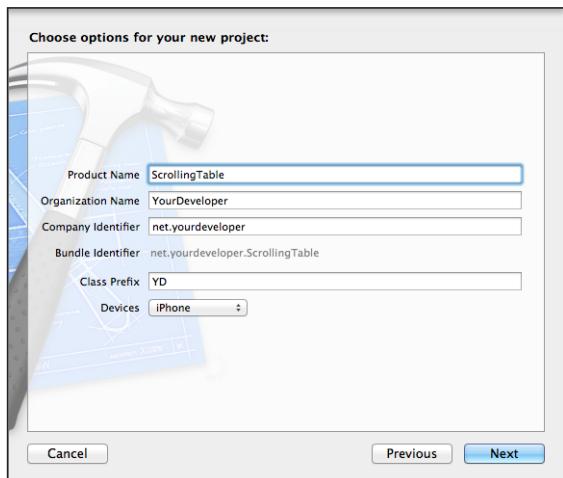


FIGURE 2-4

Open the `YDViewController.xib` file with Interface Builder and create a user interface as shown in Figure 2-5.

Set up the `YDViewController.h` file as shown in Listing 2-3. In addition to the previous sample, two actions have been added for the two introduced `UIButtons`.

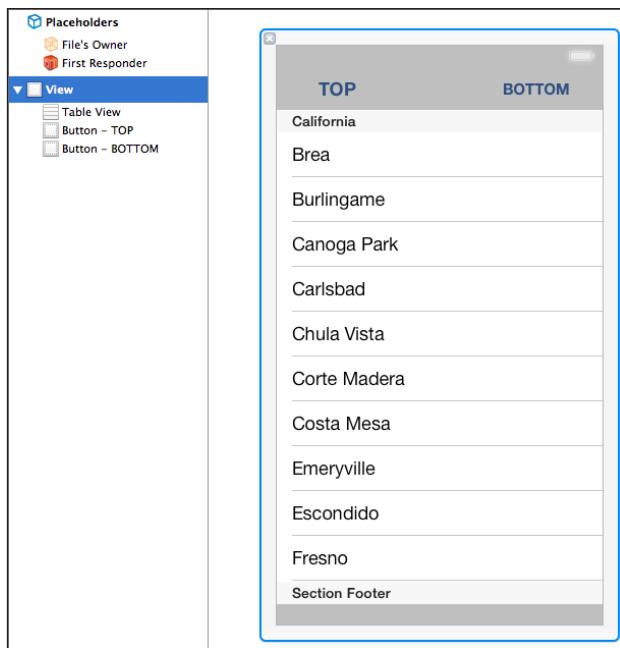


FIGURE 2-5

**LISTING 2-3: Chapter2/ScrollingTable/YDViewController.h**

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

@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;
@property (nonatomic, strong) NSMutableArray* rowData;
- (IBAction)scrollToTop:(UIButton *)sender;
- (IBAction)scrollToBottom:(UIButton *)sender;

@end
```

The implementation of `YDViewController.m` is similar to the one in the previous listing, where the only difference is that now the `scrollToTop:` and `scrollToBottom:` methods are implemented as shown in Listing 2-4.

**LISTING 2-4: Chapter2/ScrollingTable/YDViewController.m**

```
#import "YDViewController.h"

@interface YDViewController ()
```

*continues*

**LISTING 2-4** (*continued*)

```

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    [self loadData];
}

-(void)loadData
{
    if (self.rowData!=nil)
    {
        [self.rowData removeAllObjects];
        self.rowData=nil;
    }

    self.rowData = [[NSMutableArray alloc] init];
    for (int i=0 ; i<100;i++)
    {
        [self.rowData addObject:[NSString stringWithFormat:@"Row: %i",i]];
    }
    //now my datasource is populated let's reload the tableview
    [self.mTableView reloadData];
}

#pragma mark UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

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

    return [self.rowData count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = (UITableViewCell *)[tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    cell.selectionStyle = UITableViewCellSelectionStyleNone;
    cell.textLabel.text = [self.rowData objectAtIndex:indexPath.row];
    return cell;
}

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

- (void)didReceiveMemoryWarning

```

```

{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)scrollToTop:(UIButton *)sender
{
    NSIndexPath *topRow = [NSIndexPath indexPathForRow:0 inSection:0];
    [self.mTableView scrollToRowAtIndexPath:topRow
        atScrollPosition:UITableViewScrollPositionTop animated:YES];
}

- (IBAction)scrollToBottom:(UIButton *)sender
{
    NSIndexPath *bottomRow = [NSIndexPath indexPathForRow:
        [self.rowData count]-1 inSection:0];
    [self.mTableView scrollToRowAtIndexPath:bottomRow
        atScrollPosition:UITableViewScrollPositionBottom animated:YES];
}
@end

```

In your `scrollToTop:` method, you create an instance of an `NSIndexPath` object with an `indexPathForRow` value set to 0 to scroll to the top. In the `scrollToBottom:` method, you create the `NSIndexPath` instance with a value of `[self.rowData count]-1` to scroll to the bottom.

When you call the `scrollToRowAtIndexPath:atScrollPosition:animated:` method with the created `NSIndexPath` object, the `mTableView` scrolls either to the top of the table or to the bottom of the table.

The result of this implementation is shown in Figures 2-6 and 2-7.

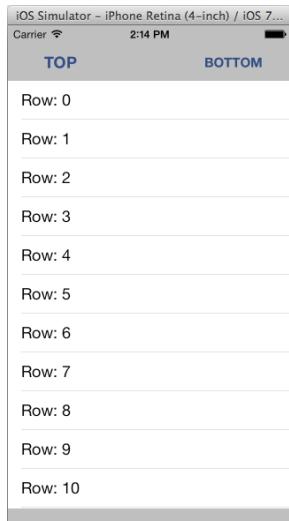


FIGURE 2-6

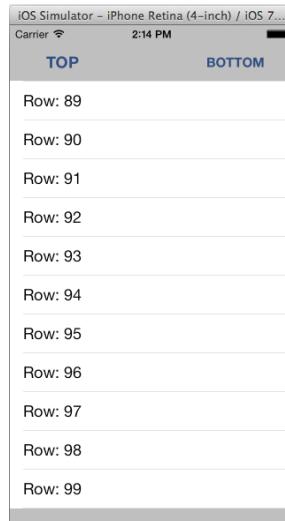


FIGURE 2-7

## BUILDING A CHAT VIEW CONTROLLER

In this section you develop a chat view controller that mimics the behavior of iMessage and other instant messaging applications. To achieve this, you learn how to create a custom instance of a UITableView, working with flexible cell heights and custom cells.

The final application will look like Figure 2-8.



FIGURE 2-8

Start Xcode and create a new project using the Single View Application Project template, and name it YDChatApp using the configuration shown in Figure 2-9.

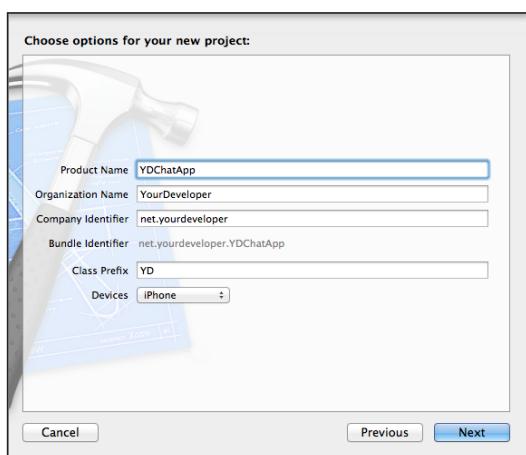


FIGURE 2-9

The images used for this example are available in the download for this chapter.

The `YDViewController` that presents the custom `UITableView` you're about to develop was developed without using Interface Builder. All the UI code is part of the `YDViewController.m` file.

## Building a datasource

You won't be using a standard `UITableView`, but you'll create your custom `UITableView` object with specific behavior and use custom cells to support the different chat bubbles and sections. For that reason, you start with coding a custom datasource that will be hooked up to the custom `UITableView`. Create a new protocol that inherits from `NSObject` and name it `YDChatTableViewDataSource`. The source for this protocol is shown in Listing 2-5.

**LISTING 2-5:** Chapter2/YDChatApp/YDChatTableViewDataSource.h

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

@class YDChatData;
@class YDChatTableView;
@protocol YDChatTableViewDataSource <NSObject>

- (NSInteger)rowsForChatTable:(YDChatTableView *)tableView;
- (YDChatData *)chatTableView:(YDChatTableView *)tableView
    dataForRow:(NSInteger)row;

@end
```

This protocol inherits directly from `NSObject` and has two methods defined in the protocol that you must implement in your `YDViewController` where you define your custom `UITableView`.

## Building a Chat Data Object

To make life easier, define an object called `YDChatData`, which is used to store the information about a chat message. You can initialize the object with the user of this chat, a date stamp, and either a text or an image. The enumerator `YDChatType` has two possible values, `ChatTypeMine` and `ChatTypeSomeone`, taking care of the positioning of the chat message in the `UITableView`. Create a new Objective-C class that inherits from `NSObject` and name it `YDChatData`.

The source code for the `YDChatData.h` file is shown in Listing 2-6.

**LISTING 2-6:** Chapter2/YDChatApp/YDChatData.h

```
#import <Foundation/Foundation.h>
@class YDChatUser;
//enumerator to identify the chattype
typedef enum _YDChatType
{
    ChatTypeMine = 0,
    ChatTypeSomeone = 1
} YDChatType;

@interface YDChatData : NSObject
```

*continues*

**LISTING 2-6** (*continued*)

```

@property (readonly, nonatomic) YDChatType type;
@property (readonly, nonatomic, strong) NSDate *date;
@property (readonly, nonatomic, strong) UIView *view;
@property (readonly, nonatomic) UIEdgeInsets insets;
@property (nonatomic,strong) YDChatUser *chatUser;

//custom initializers

+ (id)dataWithText:(NSString *)text date:(NSDate *)date
    type:(YDChatType)type andUser:(YDChatUser *)_user;

+ (id)dataWithImage:(UIImage *)image date:(NSDate *)date
    type:(YDChatType)type andUser:(YDChatUser *)_user;

+ (id)dataWithView:(UIView *)view date:(NSDate *)date
    type:(YDChatType)type andUser:(YDChatUser *)_user
    insets:(UIEdgeInsets)insets;

@end

```

In the implementation, you implement the different initialization methods as shown in Listing 2-7.

**LISTING 2-7:** Chapter2/YDChatApp/YDChatData.m

```

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

@implementation YDChatData
//create some constant UIEdgeInsets to property align text and images
const UIEdgeInsets textInsetsMine = {5, 10, 11, 17};
const UIEdgeInsets textInsetsSomeone = {5, 15, 11, 10};
const UIEdgeInsets imageInsetsMine = {11, 13, 16, 22};
const UIEdgeInsets imageInsetsSomeone = {11, 18, 16, 14};
#pragma initializers
+ (id)dataWithText:(NSString *)text date:(NSDate *)date type:(YDChatType)type
    andUser:(YDChatUser *)_user
{
    return [[YDChatData alloc] initWithText:text date:date
        type:type andUser:_user];
}

•(id)initWithText:(NSString *)text date:(NSDate *)date type:(YDChatType)type
andUser:(YDChatUser *)_user
{
    UIFont* font = [UIFont boldSystemFontOfSize:12];
    int width = 225, height = 10000.0;
    NSMutableDictionary *atts = [[NSMutableDictionary alloc] init];
    [atts setObject:font forKey:NSFontAttributeName];
    CGRect size = [text boundingRectWithSize:CGSizeMake(width, height)
        options:NSStringDrawingUsesLineFragmentOrigin
        attributes:atts]

```

```
        context:nil];
    UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, size.size.
width, size.size.height)];
    label.numberOfLines = 0;
    label.lineBreakMode = NSLineBreakByWordWrapping;
    label.text = (text ? text : @"");
    label.font = font;
    label.backgroundColor = [UIColor clearColor];
    UIEdgeInsets insets = (type == ChatTypeMine ? textInsetsMine :
textInsetsSomeone);
    return [self initWithFrame:label date:date type:type andUser:_user
insets:insets];
}

•(id)initWithImage:(UIImage *)image date:(NSDate *)date type:(YDChatType)type
    andUser:(YDChatUser *)_user
{
    CGSize size = image.size;
    if (size.width > 220)
    {
        size.height /= (size.width / 220);
        size.width = 220;
    }
    UIImageView *imageView = [[UIImageView alloc] initWithFrame:
        CGRectMake(0, 0, size.width, size.height)];
    imageView.image = image;
    imageView.layer.cornerRadius = 5.0;
    imageView.layer.masksToBounds = YES;
    UIEdgeInsets insets =
    (type == ChatTypeMine ? imageInsetsMine : imageInsetsSomeone);
    return [self initWithFrame:imageView date:date type:type andUser:_user
insets:insets];
}
+ (id)dataWithView:(UIView *)view date:(NSDate *)date type:(YDChatType)type
    andUser:(YDChatUser *)_user insets:(UIEdgeInsets)insets
{
    return [[YDChatData alloc] initWithFrame:view date:date type:type
        andUser:_user insets:insets];
}

•(id)initWithView:(UIView *)view date:(NSDate *)date type:(YDChatType)type
    andUser:(YDChatUser *)_user insets:(UIEdgeInsets)insets
{
    self = [super init];
    if (self)
    {
        _chatUser = _user;
        _view = view;
        _date = date;
        _type = type;
        _insets = insets;
    }
    return self;
}
@end
```

## Building a Custom UITableView

Create a new Objective-C class called `YDChatTableView` that inherits from `UITableView`, and implement an enum named `ChatBubbleTypingType` and the required properties as shown in Listing 2-8.

**LISTING 2-8:** Chapter2/YDChatApp/YDChatTableView.h

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

#import "YDChatTableViewDataSource.h"
#import "YDChatTableViewCell.h"
//enumerator to identify the bubble type
typedef enum _ChatBubbleTypingType
{
    ChatBubbleTypingTypeNobody = 0,
    ChatBubbleTypingTypeMe = 1,
    ChatBubbleTypingTypeSomebody = 2
} ChatBubbleTypingType;

@interface YDChatTableView : UITableView
@property (nonatomic, assign) id<YDChatTableViewDataSource> chatDataSource;
@property (nonatomic) NSTimeInterval snapInterval;
@property (nonatomic) ChatBubbleTypingType typingBubble;

@end
```

The `YDchatTableView` implementation private interface is subscribing to the `UITableViewDelegate` and `UITableViewDataSource` protocol, and a `bubbleSection` property is defined here.

The initializer method sets the default properties for the `UITableView`, such as the background color and the delegate and datasource properties. You override the `reloadData` method to write your own code to load the data in your `YDChatTableView`.

Also, you must override the `numberOfSectionsInTableView`, `tableView:numberOfRowsInSection:`, `tableView:heightForRowAtIndexPath:`, and `tableView:cellForRowAtIndexPath:` methods. The `tableView:cellForRowAtIndexPath:` method is creating and returning a `YDChatHeaderTableViewCell` or a `YDChatTableViewCell`.

The `tableView:heightForRowAtIndexPath:` is either returning the height of the `YDChatHeaderTableViewCell` if it's the first row, or the calculated height based on the content of the `YDChatData` related to this specific data row.

The complete implementation is shown in Listing 2-9.

**LISTING 2-9:** Chapter2/YDChatApp/YDChatTableView.m

```
#import "YDChatTableView.h"
#import "YDChatData.h"
#import "YDChatHeaderTableViewCell.h"
```

```

@interface YDChatTableView ()<UITableViewDelegate, UITableViewDataSource>

@property (nonatomic, retain) NSMutableArray *bubbleSection;

@end

@implementation YDChatTableView
- (void)initializer
{
    self.backgroundColor = [UIColor clearColor];
    self.separatorStyle = UITableViewCellStyleNone;
    self.delegate = self;
    self.dataSource = self;
    //the snap interval in seconds implements a headerview to seperate chats
    self.snapInterval = 60 * 60 * 24; //one day
    self.typingBubble = ChatBubbleTypingTypeNobody;
}

- (id)init
{
    self = [super init];
    if (self) [self initializer];
    return self;
}

- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) [self initializer];
    return self;
}

- (id)initWithFrame:(CGRect)frame style:(UITableViewStyle)style
{
    self = [super initWithFrame:frame style:UITableViewStylePlain];
    if (self) [self initializer];
    return self;
}

#pragma mark - Override

- (void)reloadData
{
    self.showsVerticalScrollIndicator = NO;
    self.showsHorizontalScrollIndicator = NO;
    self.bubbleSection = nil;
    int count = 0;
    self.bubbleSection = [[NSMutableArray alloc] init];
    if (self.chatDataSource && (count = [self.chatDataSource
        rowsForChatTable:self]) > 0)
    {
        NSMutableArray *bubbleData = [[NSMutableArray alloc]
            initWithCapacity:count];
        for (int i = 0; i < count; i++)
}

```

*continues*

**LISTING 2-9** (continued)

```

{
    NSObject *object = [self.chatDataSource
        chatTableView:self dataForRow:i];
    assert([object isKindOfClass:[YDChatData class]]);
    [bubbleData addObject:object];
}

[bubbleData sortUsingComparator:^NSComparisonResult(id obj1, id obj2)
{
    YDChatData *bubbleData1 = (YDChatData *)obj1;
    YDChatData *bubbleData2 = (YDChatData *)obj2;

    return [bubbleData1.date compare:bubbleData2.date];
}];

NSDate *last = [NSDate dateWithTimeIntervalSince1970:0];
NSMutableArray *currentSection = nil;
for (int i = 0; i < count; i++)
{
    YDChatData *data = (YDChatData *)[bubbleData objectAtIndex:i];
    if ([data.date timeIntervalSinceDate:last] > self.snapInterval)
    {
        currentSection = [[NSMutableArray alloc] init];
        [self.bubbleSection addObject:currentSection];
    }
    [currentSection addObject:data];
    last = data.date;
}
}
[super reloadData];
}

-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    int result = [self.bubbleSection count];
    if (self.typingBubble != ChatBubbleTypingTypeNobody) result++;
    return result;
}

•(NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    if (section >= [self.bubbleSection count]) return 1;
    return [[self.bubbleSection objectAtIndex:section] count] + 1;
}

•(float)tableView:(UITableView *)tableView heightForRowAtIndexPath:
    (NSIndexPath *)indexPath
{
    // Header
    if (indexPath.row == 0)
    {
        return [YDChatHeaderTableViewCell height];
    }
}

```

```

    }
    YDChatData *data = [[self.bubbleSection objectAtIndex:indexPath.section]
        objectAtIndex:indexPath.row - 1];

    return MAX(data.insets.top + data.view.frame.size.height +
        data.insets.bottom, 52);
}

•(UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Header based on snapInterval
    if (indexPath.row == 0)
    {
        static NSString *cellId = @"HeaderCell";
        YDChatHeaderTableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:cellId];
        YDChatData *data = [[self.bubbleSection objectAtIndex:indexPath.section]
            objectAtIndex:0];
        if (cell == nil) cell = [[YDChatHeaderTableViewCell alloc] init];
        cell.date = data.date;
        return cell;
    }
    // Standard
    static NSString *cellId = @"ChatCell";
    YDChatTableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:cellId];
    YDChatData *data = [[self.bubbleSection objectAtIndex:indexPath.section]
        objectAtIndex:indexPath.row - 1];
    if (cell == nil) cell = [[YDChatTableViewCell alloc] init];
    cell.data = data;
    return cell;
}

@end

```

## Flexible Cell Height

Because chat messages can have an image and text, which can vary in length, the height for each cell table can be different and needs to be calculated based on the content of the cell. In your override for the `tableView:heightForRowAtIndexPath:` method, there is a condition that returns the height for the cell if the cell is a header cell, and if the cell is a normal chat cell, it retrieves the `YDChatData` object linked to that cell and calculates the MAX value of the `UIEdgeInsets` used to display the cell. The following code snippet is implemented in the `YDChatTableView` and is responsible for returning a flexible height of a cell:

```

(float)tableView:(UITableView *)tableView
    heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Header
    if (indexPath.row == 0)
    {

```

```

        return [YDChatHeaderTableViewCell height];
    }
YDChatData *data = [[self.bubbleSection objectAtIndex:indexPath.section]
    objectAtIndex:indexPath.row - 1];
return MAX(data.insets.top + data.view.frame.size.height +
    data.insets.bottom, 52);
}

```

## Developing Custom Cells

To display the chat data correctly, you require two different custom `UITableViewCell` objects that both inherit from `UITableViewCell`.

One is to display the header; in this case that is the date and time grouping related to the `snapInterval`. The other one is to display the chat message from `YDChatData`. This object has a static method of type `CGFloat` called `height` that returns the height of this `UITableViewCell` and a `date` property so the date and time can be derived from the `snapInterval` property. Create a new Objective-C class named `YDChatTableViewCellHeaderCell`, open the `YDChatTableViewCellHeaderCell.h` file, and apply the code as shown in Listing 2-10.

**LISTING 2-10:** Chapter2/YDChatApp/YDChatTableViewCellHeaderCell.h

```

#import <UIKit/UIKit.h>

@interface YDChatHeaderTableViewCell : UITableViewCell
+ (CGFloat)height;
@property (nonatomic, strong) NSDate *date;
@end

```

The implementation of the `YDChatTableViewCellHeaderCell` simply returns a value of 30.0 for the `height` method. The  `setDate` method accepts a date and creates a `UILabel` that is added to the view to display the date-time stamp of this section. The implementation is shown in Listing 2-11.

**LISTING 2-11:** Chapter2/YDChatApp/YDChatTableViewCellHeader.m

```

#import "YDChatHeaderTableViewCell.h"
@interface YDChatHeaderTableViewCell ()
@property (nonatomic, retain) UILabel *label;
@end
@implementation YDChatHeaderTableViewCell
+ (CGFloat)height
{
    return 30.0;
}
- (void)setDate:(NSDate *)value

```

```

{
    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
    [dateFormatter setTimeStyle:NSDateFormatterShortStyle];
    NSString *text = [dateFormatter stringFromDate:value];

    if (self.label)
    {
        self.label.text = text;
        return;
    }
    self.selectionStyle = UITableViewCellSelectionStyleNone;
    self.label = [[UILabel alloc] initWithFrame:CGRectMake(0, 0,
    self.frame.size.width, [YDChatHeaderTableViewCell height])];
    self.label.text = text;
    self.label.font = [UIFont boldSystemFontOfSize:12];
    self.label.textAlignment = NSTextAlignmentCenter;
    self.label.shadowOffset = CGSizeMake(0, 1);
    self.label.shadowColor = [UIColor whiteColor];
    self.label.textColor = [UIColor darkGrayColor];
    self.label.backgroundColor = [UIColor clearColor];
    [self addSubview:self.label];
}
@end

```

Now that you've created a class for the `HeaderCell`, you also need to create a custom class for the `ChatCell`, displaying the actual chat message. Create a new Objective-C class that inherits from `UITableViewCell` and name it `YDChatTableViewCell`. Give the class a single property of type `YDChatData` to display the actual chat message and return it as a custom `UITableViewCell`.

Implement the code in the `YDChatTableViewCell.h` file as shown in Listing 2-12.

#### LISTING 2-12: Chapter2/YDChatApp/YDChatTableViewCell.h

```

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

@interface YDChatTableViewCell : UITableViewCell

@property (nonatomic, strong) YDChatData *data;
-(void)setData(YDChatData*)data;
@end

```

The `setData:` method accepts a `YDChatData` object and assigns it to the `data` property. Next it calls the `rebuildUserInterface` method that will create the `bubbleImage` if it has not been created before. If the `YDchatData` object has a user value, the avatar image of the chat user is used and added as a subview.

The implementation of the `YDChatTableViewCell.m` file is shown in Listing 2-13.

**LISTING 2-13:** Chapter2/YDChatApp/YDChatTableViewCell.m

```
#import <QuartzCore/QuartzCore.h>
#import "YDChatTableViewCell.h"
#import "YDChatData.h"
#import "YDChatUser.h"
@interface YDChatTableViewCell ()
//declare properties
@property (nonatomic, retain) UIView *customView;
@property (nonatomic, retain) UIImageView *bubbleImage;
@property (nonatomic, retain) UIImageView *avatarImage;

- (void) setupInternalData;

@end

@implementation YDChatTableViewCell
@synthesize data=_data;
- (void)setData:(YDChatData *)data
{
    _data = data;
    [self rebuildUserInterface];
}

- (void) rebuildUserInterface
{
    self.selectionStyle = UITableViewCellStyleNone;
    if (!self.bubbleImage)
    {
        self.bubbleImage = [[UIImageView alloc] init];
        [self addSubview:self.bubbleImage];
    }
    YDChatType type = self.data.type;
    CGFloat width = self.data.view.frame.size.width;
    CGFloat height = self.data.view.frame.size.height;
    CGFloat x = (type == ChatTypeSomeone) ? 0 :
        self.frame.size.width -
        width -
        self.data.insets.left -
        self.data.insets.right;
    CGFloat y = 0;
    //if we have a chatUser show the avatar of the YDChatUser property
    if (self.data.chatUser)
    {
        YDChatUser *thisUser = self.data.chatUser;
        [self.avatarImage removeFromSuperview];
        self.avatarImage = [[UIImageView alloc] initWithImage:(thisUser.avatar ?
            thisUser.avatar : [UIImage imageNamed:@"noAvatar.png"])];
        self.avatarImage.layer.cornerRadius = 9.0;
        self.avatarImage.layer.masksToBounds = YES;
        self.avatarImage.layer.borderColor =
            [UIColor colorWithWhite:0.0 alpha:0.2].CGColor;
        self.avatarImage.layer.borderWidth = 1.0;
    }
}
```

```

//calculate the x position
CGFloat avatarX = (type == ChatTypeSomeone) ? 2 :
    self.frame.size.width - 52;
CGFloat avatarY = self.frame.size.height - 50;
//set the frame correctly
self.avatarImage.frame = CGRectMake(avatarX, avatarY, 50, 50);
[self addSubview:self.avatarImage];
CGFloat delta = self.frame.size.height -
    (self.data.insets.top + self.data.insets.bottom +
     self.data.view.frame.size.height);
if (delta > 0) y = delta;
if (type == ChatTypeSomeone) x += 54;
if (type == ChatTypeMine) x -= 54;
}
[self.customView removeFromSuperview];
self.customView = self.data.view;
self.customView.frame =
    CGRectMake(x + self.data.insets.left,
               y + self.data.insets.top, width, height);
[self.contentView addSubview:self.customView];
//depending on the ChatType a bubble image on the left or right
if (type == ChatTypeSomeone)
{
    self.bubbleImage.image = [[UIImage imageNamed:@"yoububble.png"]
        stretchableImageWithLeftCapWidth:21 topCapHeight:14];
}
else {
    self.bubbleImage.image = [[UIImage imageNamed:@"mebubble.png"]
        stretchableImageWithLeftCapWidth:15 topCapHeight:14];
}
self.bubbleImage.frame =
    CGRectMake(x, y, width + self.data.insets.left +
               self.data.insets.right, height +
               self.data.insets.top + self.data.insets.bottom);
}
- (void)setFrame:(CGRect)frame
{
    [super setFrame:frame];
    [self rebuildUserInterface];
}
@end

```

## Creating the Chat User Object

Create a new class named `YDChatUser` that has two properties—a username and an avatar—which will be shown in the `YDChatTableViewCell` you’ve just created. This class is designed to set a username and an avatar image for a user object that you can relate to the `YDchatData` object.

Create a new Objective-C class that inherits from `NSObject` and name it `YDChatUser`. The `YDChatUser.h` file is shown in Listing 2-14.

**LISTING 2-14:** Chapter2/YDChatApp/YDChatUser.h

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

@interface YDChatUser : NSObject
@property (nonatomic, strong) NSString *username;
@property (nonatomic, strong) UIImage *avatar;

- (id)initWithUsername:(NSString *)user avatarImage:(UIImage *)image;
@end
```

Implement the custom initializer and assign the passed values to the properties as shown in Listing 2-15.

**LISTING 2-15:** Chapter2/YDChatApp/YDChatUser.m

```
#import "YDChatUser.h"

@implementation YDChatUser
@synthesize avatar = _avatar;
@synthesize username = _username;

- (id)initWithUsername:(NSString *)user avatarImage:(UIImage *)image
{
    self = [super init];
    if (self)
    {
        self.avatar = [image copy];
        self.username = [user copy];
    }
    return self;
}
@end
```

## Putting It All Together

Now that you've developed all the individual components, you can code your `YDViewController` to display the `YDChatTableView` with some messages.

The `YDViewController.h` file doesn't need any coding.

The `YDViewController.m` file starts with importing the required header files and subscribing to the `YDChatTableViewDataSource` and `UITextViewDelegate` protocol. The `viewDidLoad` method starts with the programmatic creation of the user interface elements and ends with the creation of two objects of type `YDChatUser`, as shown in the next code sample:

```
me = [[YDChatUser alloc] initWithUsername:@"Peter"
                           avatarImage:[UIImage imageNamed:@"me.png"]];
you = [[YDChatUser alloc] initWithUsername:@"You"
                           avatarImage:[UIImage imageNamed:@"noavatar.png"]];
```

Finally, in the `viewDidLoad` method some `YDChatData` records are created and added to the `Chats` array that is used as the datasource for your `YDChatTableView`:

```

YDChatData *first = [YDChatData dataWithText:
    @"Hey, how are you doing? I'm in Paris take a look at this picture."
    date:[NSDate dateWithTimeIntervalSinceNow:-600]
    type:ChatTypeMine andUser:me];
YDChatData *second = [YDChatData dataWithImage:
    [UIImage imageNamed:@"eiffeltower.jpg"]
    date:[NSDate dateWithTimeIntervalSinceNow:-290]
    type:ChatTypeMine andUser:me];
YDChatData *third = [YDChatData dataWithText:
    @"Wow.. Really cool picture out there. Wish I could be with you"
    date:[NSDate dateWithTimeIntervalSinceNow:-5]
    type:ChatTypeSomeone andUser:you];
YDChatData *forth = [YDChatData dataWithText:
    @"Maybe next time you can come with me."
    date:[NSDate dateWithTimeIntervalSinceNow:+0]
    type:ChatTypeMine andUser:me];
//Initialize the Chats array with the created YDChatData objects
Chats = [[NSMutableArray alloc]
    initWithObjects:first, second, third, forth, nil];

```

The `sendMessage` method creates a `YDChatData` object, initializes it with the text from the `msgText` control, adds it to the `Chats` array, and calls the `reloadData` method of the `chatTable` object.

The `textView:shouldChangeTextInRange:replacementText:`, `textViewDidBeginEditing:`, and `textViewDidChange:` methods are used to manipulate the user interface when you select the `UITextView` and start typing in it. The `shortenTableView` and `showTableView` methods are controlling the height of the `YDChatTableView`.

The complete implementation is shown in Listing 2-16.

#### LISTING 2-16: Chapter2/YDChatApp/YDViewController.m

```

#import "YDChatUser.h"
#import "YDChatTableViewDataSource.h"
#import "YDViewController.h"
#import <QuartzCore/QuartzCore.h>
#import "YDChatTableView.h"
#import "YDChatTableViewDataSource.h"
#import "YDChatData.h"
#import "YDChatUser.h"
#define lineHeight 16.0f
@interface YDViewController ()<YDChatTableViewDataSource, UITextViewDelegate>
{
    YDChatTableView *chatTable;
    UIView *textInputView;
    UITextField *textField;
    NSMutableArray *Chats;

    UIView* sendView;
    UIButton* sendButton;
    UITextView* msgText;
}

```

*continues*

**LISTING 2-16 (continued)**

```

BOOL composing;
float prevLines;
YDChatUser* me ;
YDChatUser* you ;
}
@end

@implementation YDViewController

CGRect appFrame;
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor lightGrayColor];
    //create your instance of YDChatTableView
    self.chatTable=[[YDChatTableView alloc] initWithFrame:
    CGRectMake(0,40,[UIScreen mainScreen] bounds).size.width,
    [[UIScreen mainScreen] bounds].size.height -40) style:UITableViewStylePlain];
    chatTable.backgroundColor=[UIColor whiteColor];
    [self.view addSubview:chatTable];
    appFrame= [[UIScreen mainScreen] applicationFrame];

    sendView = [[UIView alloc] initWithFrame:
    CGRectMake(0,appFrame.size.height-56,320,56)];
    sendView.backgroundColor=[UIColor blueColor];
    sendView.alpha=0.9;

    msgText = [[UITextView alloc] initWithFrame:CGRectMake(7,10,225,36)];
    msgText.backgroundColor = [UIColor whiteColor];
    msgText.textColor=[UIColor blackColor];
    msgText.font=[UIFont boldSystemFontOfSize:12];
    msgText.autoresizingMask =
        UIViewAutoresizingFlexibleHeight |
        UIViewAutoresizingFlexibleTopMargin;
    msgText.layer.cornerRadius = 10.0f;
    msgText.returnKeyType=UIReturnKeySend;
    msgText.showsHorizontalScrollIndicator=NO;
    msgText.showsVerticalScrollIndicator=NO;
    //Set the delegate so you can respond to user input
    msgText.delegate=self;
    [sendView addSubview:msgText];
    msgText.contentInset = UIEdgeInsetsMake(0,0,0,0);
    [self.view addSubview:sendView];

    sendButton = [[UIButton alloc] initWithFrame:CGRectMake(235,10,77,36)];
    sendButton.backgroundColor=[UIColor lightGrayColor];
    [sendButton addTarget:self action:@selector(sendMessage)
        forControlEvents:UIControlEventTouchUpInside];
    sendButton.autoresizingMask=UIViewAutoresizingFlexibleTopMargin;
    sendButton.layer.cornerRadius=6.0f;
    [sendButton setTitle:@"Send" forState:UIControlStateNormal];
    [sendView addSubview:sendButton];
    //create two YDChatUser object one representing me and one
    representing the other party
}

```

```

me = [[YDChatUser alloc] initWithUsername:@"Peter"
    avatarImage:[UIImage imageNamed:@"me.png"]];
you = [[YDChatUser alloc] initWithUsername:@"You"
    avatarImage:[UIImage imageNamed:@"noavatar.png"]];
//Create some YDChatData objects here
YDChatData *first = [YDChatData dataWithText:
    @"Hey, how are you doing? I'm in Paris take a look at this picture."
    date:[NSDate dateWithTimeIntervalSinceNow:-600]
    type:ChatTypeMine andUser:me];
YDChatData *second = [YDChatData dataWithImage:
    [UIImage imageNamed:@"eiffeltower.jpg"]
    date:[NSDate dateWithTimeIntervalSinceNow:-290]
    type:ChatTypeMine andUser:me];
YDChatData *third = [YDChatData dataWithText:
    @"Wow.. Really cool picture out there. Wish I could be with you"
    date:[NSDate dateWithTimeIntervalSinceNow:-5]
    type:ChatTypeSomeone andUser:you];
YDChatData *forth = [YDChatData dataWithText:
    @"Maybe next time you can come with me."
    date:[NSDate dateWithTimeIntervalSinceNow:+0]
    type:ChatTypeMine andUser:me];
//Initialize the Chats array with the created YDChatData objects
Chats = [[NSMutableArray alloc]
    initWithObjects:first, second, third, forth, nil];
//set the chatDataSource
chatTable.chatDataSource = self;
//call the reloadData, this is actually calling your override method
[chatTable reloadData];
}

-(void)sendMessage
{
    composing=NO;
    YDChatData *thisChat = [YDChatData dataWithText:msgText.text
        date:[NSDate date] type:ChatTypeMine andUser:me];
    [Chats addObject:thisChat];
    [chatTable reloadData];
    [self showTableView];
    [msgText resignFirstResponder];
    msgText.text=@";
    sendView.frame=CGRectMake(0,appFrame.size.height-56,320,56);
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];
    [chatTable scrollToRowAtIndexPath:indexPath
        atScrollPosition:UITableViewScrollIndicatorBottom
        animated:YES];
}

#pragma UITextViewDelegate
//if user presses enter consider as end of message and send it
-(BOOL)textView:(UITextView *)textView shouldChangeTextInRange:(NSRange)range
    replacementText:(NSString *)text
{
    if([text isEqualToString:@"\n"]) {
        [self sendMessage];
}

```

*continues*

**LISTING 2-16** (continued)

```

        return NO;
    }
    return YES;
}
// this function returns the height of the entered text in the msgText field
-(CGFloat )textY
{
    UIFont* systemFont = [UIFont boldSystemFontOfSize:12];
    int width = 225.0, height = 10000.0;
    NSMutableDictionary *atts = [[NSMutableDictionary alloc] init];
    [atts setObject:systemFont forKey:NSFontAttributeName];

    CGRect size = [msgText.text boundingRectWithSize:CGSizeMake(width, height)
options:NSStringDrawingUsesLineFragmentOrigin
attributes:atts
context:nil];
    float textHeight = size.size.height;

    float lines = textHeight / lineHeight;
    if (lines >=4)
        lines=4;
    if ([msgText.text length]==0)
        lines=0.9375f;
    return 190 - (lines * lineHeight) + lineHeight;
}
-(void)textViewDidChange:(UITextView *)textView
{
    UIFont* systemFont = [UIFont boldSystemFontOfSize:12];
    int width = 225.0, height = 10000.0;
    NSMutableDictionary *atts = [[NSMutableDictionary alloc] init];
    [atts setObject:systemFont forKey:NSFontAttributeName];

    CGRect size = [msgText.text boundingRectWithSize:CGSizeMake(width, height)
options:NSStringDrawingUsesLineFragmen
tOrigin
                        attributes:atts
                        context:nil];
    float textHeight = size.size.height;
    float lines = textHeight / lineHeight;
    if (lines >=4)
        lines=4;

    composing=YES;
    msgText.contentInset = UIEdgeInsetsMake(0,0,0,0);
    sendView.frame = CGRectMake(0,appFrame.size.height-270 - (lines * lineHeight) +
lineHeight ,320,56 + (lines * lineHeight)-lineHeight);

    if (prevLines!=lines)
        [self shortenTableView];

    prevLines=lines;
}

```

```

        prevLines=lines;
    }
    //let's change the frame of the chatTable so we can see the bottom

- (void)shortenTableView
{
    [UIView beginAnimations:@"moveView" context:nil];
    [UIView setAnimationDuration:0.1];
    chatTable.frame=CGRectMake(0, 0, 320, [self textY] );
    [UIView commitAnimations];
    prevLines=1;
}

// show the chatTable as it was

- (void)showTableView
{
    [UIView beginAnimations:@"moveView" context:nil];
    [UIView setAnimationDuration:0.1];
    chatTable.frame=CGRectMake(0,0,320,460 - 56);
    [UIView commitAnimations];
}
//when user starts typing change the frame position and shorten the chatTable

- (void)textViewDidBeginEditing:(UITextView *)textView
{
    [UIView beginAnimations:@"moveView" context:nil];
    [UIView setAnimationDuration:0.3];
    sendView.frame = CGRectMake(0,appFrame.size.height-270,320,56);
    [UIView commitAnimations];
    [self shortenTableView];
    [msgText becomeFirstResponder];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}

#pragma mark - YDChatTableView implementation
//here are the required implementation from your YDChatTableViewDataSource
- (NSInteger)rowsForChatTable:(YDChatTableView *)tableView
{
    return [Chats count];
}

- (YDChartData *)chatTableView:(YDChatTableView *)tableView
    dataForRow:(NSInteger)row
{
    return [Chats objectAtIndex:row];
}

@end

```

The Chat solution you've created implements only the logic for creating chat messages and displays them in a very fancy way, but it doesn't send or receive messages. For this, a real messaging implementation following the XMPP protocol ([www.xmpp.org](http://www.xmpp.org)) is required.

## DRILLING DOWN WITH UITABLEVIEW

If you are using a UITableView with data that can be grouped, it will make the user experience so much better if the data is grouped together in such a way that the user can drill down. In this sample project, you create an application that displays some cars with different makes and models. The implemented solution groups the cars into sections based on the make of the car.

In a normal UITableView, data displays as shown in Figure 2-10.

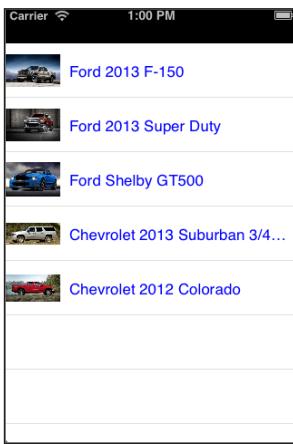


FIGURE 2-10

In this example, you learn to create an implementation that displays your data as shown in Figure 2-11.

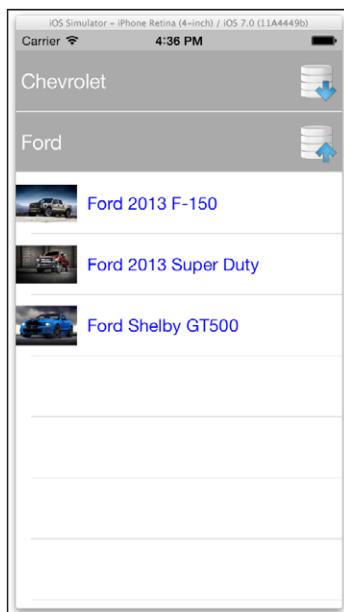


FIGURE 2-11

Start Xcode and create a new project using the Single View Application Project template, and name it YDDrillDown using the options shown in Figure 2-12.

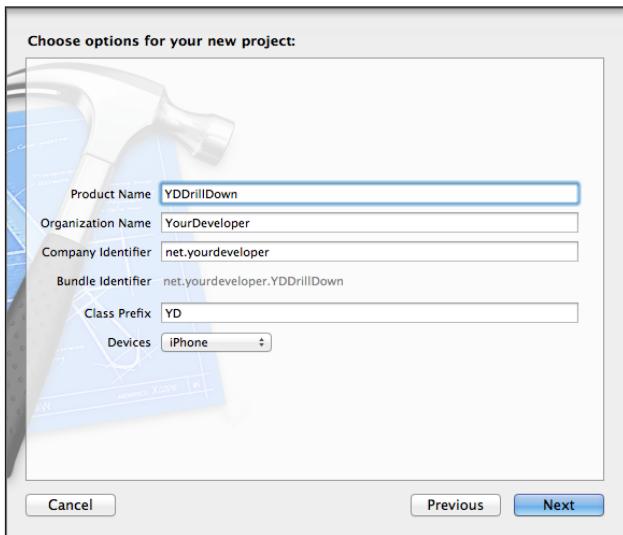


FIGURE 2-12

The images used in this sample are part of the download of this chapter.

Because you'll be showing cars in this project, start by creating a YDCar object. Create three properties: one for the make, one for the model, and a third one for the image name. Create a new Objective-C class that inherits from `NSObject` and implement the code as shown in Listing 2-17.

#### LISTING 2-17: Chapter2/YDDrillDown/YDCar.h

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

@interface YDCar : NSObject
@property (nonatomic,strong) NSString *make;
@property (nonatomic,strong) NSString *model;
@property (nonatomic,strong) NSString *imageName;

- (id)initWithMake: (NSString *)carMake model:(NSString *)carModel
              imageName:(NSString *)carImageName;
@end
```

The implementation of the custom initializer simply copies the passed parameters to the properties, as shown in Listing 2-18.

**LISTING 2-18:** Chapter2/YDDrillDown/YDCar.m

```
#import "YDCar.h"

@implementation YDCar

- (id)initWithMake: (NSString *)carMake model: (NSString *)carModel
    imageName: (NSString *)carImageName;
{
    self = [super init];
    if (self)
    {
        self.make = [carMake copy];
        self.model = [carModel copy];
        self.imageName = [carImageName copy];
    }
    return self;
}
@end
```

To implement your drill-down UITableView, create a UIView subclass called YDSectionHeaderView.

The table view will use sections to group the presented data. For this the YDSectionHeaderView will be used to display the section header in addition to the table cells themselves.

You need a UILabel to display the title, a UIButton to enable expand and collapse functionality, a BOOL to keep track of whether or not this section is expanded, and an NSInteger holding the number of rows within this section. Define a YDSectionHeaderViewDelegate protocol in the YDSectionHeaderView.h file so you can inform the delegate when a section opens and closes.

Create a new Objective-C class that inherits from UIView and name the class YDSectionHeaderView. The YDSectionHeaderView.h class is shown in Listing 2-19.

**LISTING 2-19:** Chapter2/YDDrillDown/YDSectionHeaderView.h

```
#import <UIKit/UIKit.h>
@protocol YDSectionHeaderViewDelegate;
@interface YDSectionHeaderView : UIView

@property (nonatomic,assign) BOOL expanded;
@property (nonatomic, retain) UILabel *titleLabel;
@property (nonatomic, retain) UIButton *disclosureButton;
@property (nonatomic, assign) NSInteger section;
@property (nonatomic,assign) NSInteger numberOfRowsInSection;
@property (nonatomic, retain) NSString *key;
@property (nonatomic, assign) id <YDSectionHeaderViewDelegate> delegate;
//initializer
- (id)initWithFrame: (CGRect)frame title: (NSString*)title section: (NSInteger)
    sectionNumber delegate: (id <YDSectionHeaderViewDelegate>)delegate
    numrow: (NSInteger)numofRows;
- (void)toggleOpenWithUserAction: (BOOL)userAction;
```

```

@end

@protocol YDSectionHeaderViewDelegate <NSObject>
@optional
- (void)sectionHeaderView:(YDSectionHeaderView*)sectionHeaderView
    sectionOpened:(NSInteger)section;
- (void)sectionHeaderView:(YDSectionHeaderView*)sectionHeaderView
    sectionClosed:(NSInteger)section;

@end

```

The implementation of the `YDSectionHeaderView` starts with a custom initializer that accepts the title of the section, the number of rows in this section, as well as a delegate pointer. You add a `UITapGestureRecognizer` to enable detecting a tap on the view itself. You create an instance of the `UIButton` with two different images, one for the selected state and one for the normal state. Finally, the `toggleOpenWithUserAction:` method does the magic by toggling the disclosure button and sending a message to the delegate. The complete implementation is shown in Listing 2-20.

#### LISTING 2-20: Chapter2/YDDrillDown/YDSectionHeaderView.m

```

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

@interface YDSectionHeaderView()

@end
@implementation YDSectionHeaderView

-(id)initWithFrame:(CGRect)frame title:(NSString*)title
    section:(NSInteger)sectionNumber
    delegate:(id <YDSectionHeaderViewDelegate>)delegate
    numrow:(NSInteger)numofRows{
    self = [super initWithFrame:frame];
    if (self != nil) {
        // Set up the tap gesture recognizer.
        UITapGestureRecognizer *tapGesture = [[UITapGestureRecognizer alloc]
            initWithTarget:self action:@selector(toggleOpen:)];
        [self addGestureRecognizer:tapGesture];
        _delegate = delegate;
        self.userInteractionEnabled = YES;
        self.backgroundColor = [UIColor whiteColor];
        self.numberOfRows=numofRows;
        self.key=title;
        UIView *redView = [[UIView alloc] initWithFrame:
            CGRectMake(frame.origin.x, frame.origin.y,
                      frame.size.width, frame.size.height - 1)];
        redView.backgroundColor = [UIColor redColor];
        [self addSubview:redView];
        // Create and configure the title label.
        self.section = sectionNumber;
        self.expanded=NO; //start colapsed
    }
}

```

*continues*

**LISTING 2-20** (continued)

```

CGRect titleLabelFrame = self.bounds;
titleLabelFrame.origin.x += 5.0;
titleLabelFrame.size.width -= 35.0;
CGRectInset(titleLabelFrame, 0.0, 5.0);
self.titleLabel = [[UILabel alloc] initWithFrame:titleLabelFrame];
self.titleLabel.text = title;
self.titleLabel.font = [UIFont systemFontOfSize:20];
self.titleLabel.textColor = [UIColor whiteColor];
self.titleLabel.backgroundColor = [UIColor clearColor];
[self addSubview:self.titleLabel];

// Create and configure the disclosure button.
UIButton *button = [UIButton buttonWithType:UIButtonTypeCustom];
button.frame = CGRectMake(275.0, 10.0, 40.0, 40.0);
[button setImage:[UIImage imageNamed:@"down.png"]
    forState:UIControlStateNormal];
[button setImage:[UIImage imageNamed:@"up.png"]
    forState:UIControlStateSelected];
button.backgroundColor=[UIColor clearColor];
[button addTarget:self action:@selector(toggleOpen:)
    forControlEvents:UIControlEventTouchUpInside];
[self addSubview:button];
self.disclosureButton = button;
}

return self;
}

-(IBAction)toggleOpen:(id)sender
{
    [self toggleOpenWithUserAction:YES];
}

-(void)toggleOpenWithUserAction:(BOOL)userAction
{
    // Toggle the disclosure button state.
    self.disclosureButton.selected = !self.disclosureButton.selected;
    // If this was a user action, send the delegate the appropriate message.
    if (userAction) {
        if (self.disclosureButton.selected) {
            if ([self.delegate respondsToSelector:
                @selector(sectionHeaderView:sectionOpened:)])
            {
                self.expanded=YES;
                [self.delegate sectionHeaderView:self sectionOpened:self.section];
            }
        }
    else {
        if ([self.delegate respondsToSelector:
            @selector(sectionHeaderView:sectionClosed:)])
        {
            self.expanded=NO;
            [self.delegate sectionHeaderView:self sectionClosed:self.section];
        }
    }
}

```

```

        }
    }
}
@end

```

Your YDViewController starts with importing your YDSectionHeaderView.h file. Create a property for a UITableView instance and an NSInteger to keep track of the openSectionIndex. You define an NSMutableArray named arrayOfSectionHeaders to hold instances of the created YDSectionHeaderViews. Finally, you define a property for an NSMutableDictionary named dataDict to hold the data representing your collection of cars. The complete implementation is shown in Listing 2-21.

#### LISTING 2-21: Chapter2/YDDrillDown/YDViewController.h

```

#import <UIKit/UIKit.h>
#import "YDSectionHeaderView.h"
@interface YDViewController : UIViewController

@property (nonatomic, strong) UITableView *mTableView;
@property (nonatomic, assign) NSInteger openSectionIndex;
@property (nonatomic, strong) NSMutableArray *arrayOfSectionHeaders;
@property (nonatomic, strong) NSMutableDictionary *dataDict;
@end

```

In the implementation of the YDViewController, in the viewDidLoad method, you create the table view and set its datasource and delegate properties and add it as a sub view of the ViewControllers view at the end of the viewDidLoad method, and you call the loadData method that creates and loads the data.

For example purposes, create an NSArray named ford to hold some objects of type YDCar and another NSArray named chevy to hold some Chevrolet models. Once the dataDict object has been created, create the YDSectionHeaderView objects for each key in your NSDictionary and store them in your arrayOfSectionHeaders. The code under the #pragma Headers implements the logic that is called from the delegate. It derives the correct YDSectionHeaderView from your arrayOfSectionHeaders, determines the number of rows in that section, deletes the rows from the previously opened section, and inserts the new rows to the UITableView using animations. The complete implementation is shown in Listing 2-22.

#### LISTING 2-22: Chapter2/YDDrillDown/YDViewController.m

```

#import "YDViewController.h"
#import "YDCar.h"
#define HEADER_HEIGHT 60.0f
@interface YDViewController ()<UITableViewDataSource,
UITableViewDelegate,
YDSectionHeaderViewDelegate>

@end

```

*continues*

**LISTING 2-22 (continued)**

```

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    CGRect appframe=[[UIScreen mainScreen] applicationFrame];
    //create the tableView
    self.mTableView = [[UITableView alloc]
        initWithFrame:appframe style:UITableViewStylePlain];
    self.mTableView.dataSource=self;
    self.mTableView.delegate=self;
    self.mTableView.rowHeight = 60;
    self.mTableView.sectionHeaderHeight = HEADER_HEIGHT;
    [self.view addSubview:self.mTableView];
    [self loadData];
}

- (void)loadData
{
    //Create some models for Ford
    NSArray *ford = [[NSArray alloc] initWithObjects:
        [[YDCar alloc] initWithMake:@"Ford"
            model:@"2013 F-150"
            imageName:@"2013f150.jpg"],
        [[YDCar alloc] initWithMake:@"Ford"
            model:@"2013 Super Duty"
            imageName:@"2013superduty.jpg"],
        [[YDCar alloc] initWithMake:@"Ford"
            model:@"Shelby GT500"
            imageName:@"shelbygt500.jpg"],
        nil];
    //create some models for Chevrolet
    NSArray *chevy = [[NSArray alloc] initWithObjects:
        [[YDCar alloc] initWithMake:@"Chevrolet"
            model:@"2013 Suburban 3/4 ton"
            imageName:@"suburban.png"],
        [[YDCar alloc] initWithMake:@"Chevrolet"
            model:@"2012 Colorado"
            imageName:@"colorado.jpg"],
        nil];
    //Create a dictionary to store them
    self.dataDict = [[NSMutableDictionary alloc] init];
    [self.dataDict setObject:ford forKey:@"Ford"];
    [self.dataDict setObject:chevy forKey:@"Chevrolet"];
    //Create an array of YDSectionHeaderViews with the title and number of rows
    self.arrayOfSectionHeaders=[[NSMutableArray alloc] init];
    int sectionNr = 0;
    for (id key in self.dataDict) {
        id anObject = [self.dataDict objectForKey:key];
        YDSectionHeaderView *make = [[YDSectionHeaderView alloc]
            initWithFrame:CGRectMake(0.0, 0.0, 320, HEADER_HEIGHT)]

```

```

        title:key section:sectionNr delegate:self
        numrow:[anObject count]];
    [self.arrayOfSectionHeaders addObject:make];
    sectionNr++;
}
self.openSectionIndex = NSNotFound;
[self.mTableView reloadData];
}

#pragma HEADER
-(UIView*)tableView:(UITableView*)tableView
viewForHeaderInSection:(NSInteger)section {
    YDSectionHeaderView *sectionHeaderView =
    (YDSectionHeaderView *)
        [self.arrayOfSectionHeaders objectAtIndex:section];
    return sectionHeaderView;
}
-(void)sectionHeaderView:(YDSectionHeaderView*)sectionHeaderView
sectionOpened:(NSInteger)sectionOpened {
    if (sectionOpened != NSNotFound)
//all sections are closed so we can't do anything
    {
        YDSectionHeaderView *sectionHeaderView =
            [self.arrayOfSectionHeaders
                objectAtIndex:sectionOpened];
        sectionHeaderView.expanded = YES;
/*
Create an array containing the index paths of the rows to insert
These correspond to the rows for each quotation in the
current section.
*/
        NSInteger numberOfRowsInSection = sectionHeaderView.numberOfRows;
        NSMutableArray *indexPathsToInsert = [[NSMutableArray alloc] init];
        for (NSInteger i = 0; i < numberOfRowsInSection; i++) {
            [indexPathsToInsert
                addObject:[NSIndexPath indexPathForRow:i
                    inSection:sectionOpened]];
        }
/*
Create an array containing the index paths of the rows to delete:
These correspond to the rows for each quotation in the
previously-open section, if there was one.
*/
        NSMutableArray *indexPathsToDelete = [[NSMutableArray alloc] init];
        NSInteger previousOpenSectionIndex =self.openSectionIndex;
        if (previousOpenSectionIndex != NSNotFound) {
            YDSectionHeaderView *previousOpenSectionHeaderView=
                [self.arrayOfSectionHeaders
                    objectAtIndex:previousOpenSectionIndex];
            previousOpenSectionHeaderView.expanded = NO;
            [previousOpenSectionHeaderView toggleOpenWithUserAction:NO];
            NSInteger countOfRowsToDelete =
                previousOpenSectionHeaderView.numberOfLines;
            for (NSInteger i = 0; i < countOfRowsToDelete; i++) {

```

*continues*

**LISTING 2-22** (continued)

```

        [indexPathsToDelete
            addObject:[NSIndexPath indexPathForRow:i
                inSection:previousOpenSectionIndex]];
    }

}

// Style the animation so that there's a smooth flow in either
// direction.
UITableViewController insertAnimation;
UITableViewController deleteAnimation;
if (previousOpenSectionIndex == NSNotFound ||
    sectionOpened < previousOpenSectionIndex) {
    insertAnimation = UITableViewRowAnimationTop;
    deleteAnimation = UITableViewRowAnimationBottom;
}
else {
    insertAnimation = UITableViewRowAnimationBottom;
    deleteAnimation = UITableViewRowAnimationTop;
}
// Apply the updates.
[self.tableView beginUpdates];
[self.tableView insertRowsAtIndexPaths:indexPathsToInsert
    withRowAnimation:insertAnimation];
[self.tableView deleteRowsAtIndexPaths:indexPathsToDelete
    withRowAnimation:deleteAnimation];
[self.tableView endUpdates];
self.openSectionIndex = sectionOpened;
}

}

-(void)sectionHeaderView:(YDSectionHeaderView*)sectionHeaderView
sectionClosed:(NSInteger)sectionClosed {
    YDSectionHeaderView *thisSectionHeaderView =
        [_arrayOfSectionHeaders objectAtIndex:sectionClosed];
    thisSectionHeaderView.expanded = NO;
    NSInteger countOfRowsToDelete =
        [self.tableView numberOfRowsInSection:sectionClosed];
    if (countOfRowsToDelete > 0) {
        NSMutableArray *indexPathsToDelete = [[NSMutableArray alloc] init];
        for (NSInteger i = 0; i < countOfRowsToDelete; i++) {
            [indexPathsToDelete addObject:[NSIndexPath indexPathForRow:i
                inSection:sectionClosed]];
        }
        [self.tableView deleteRowsAtIndexPaths:indexPathsToDelete
            withRowAnimation:UITableViewRowAnimationTop];
    }
    self.openSectionIndex = NSNotFound;
}

-(NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    return [_arrayOfSectionHeaders objectAtIndex:section];
}

```

```
#pragma UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [_arrayOfSectionHeaders count];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    YDSectionHeaderView *sectionHeaderView =
        [_arrayOfSectionHeaders objectAtIndex:section];
    NSInteger numStoriesInSection = sectionHeaderView.numberOfRows;
    return sectionHeaderView.expanded ? numStoriesInSection : 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = (UITableViewCell *)[tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    cell.selectionStyle = UITableViewCellSelectionStyleNone;
    YDSectionHeaderView *shv= (YDSectionHeaderView *)[_arrayOfSectionHeaders
        objectAtIndex:indexPath.section];
    YDCar *car = (YDCar *)[[self.dataDict objectForKey:shv.key]
        objectAtIndex:indexPath.row];
    UILabel* carlbl = [[UILabel alloc] initWithFrame:CGRectMake(70, 0, 250, 60)];
    carlbl.textColor=[UIColor blueColor];
    carlbl.backgroundColor=[UIColor whiteColor];
    carlbl.text = [NSString stringWithFormat:@"%@ %@",car.make,car.model];
    [cell.contentView addSubview:carlbl];
    UIImageView *imgView=[[UIImageView alloc] initWithFrame:CGRectMake(0, 0, 60, 60)];
    imgView.backgroundColor=[UIColor clearColor];
    [imgView setImage:[UIImage imageNamed:car.imageName]];
    imgView.contentMode=UIViewContentModeScaleAspectFit;
    [cell.contentView addSubview:imgView];

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

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

## Implementing a UISearchBar

In this example you create an application that uses a standard UITableView to list all the presidents of the United States of America and implement a UISearchBar object to enable a search function on the UITableView.

Start Xcode and create a new project using the Single View Application Project template, and name it `PresidentSearch` using the project options shown in Figure 2-13.

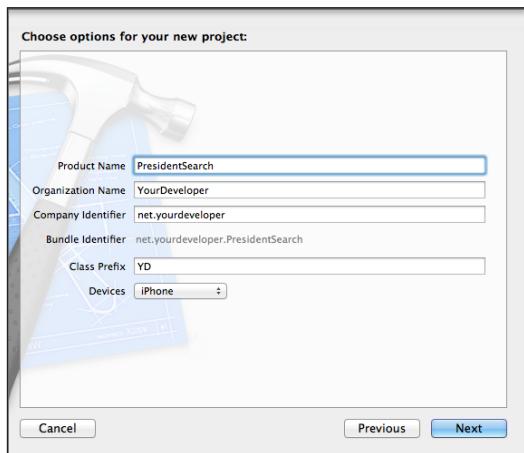


FIGURE 2-13

Start by creating a new Objective-C class named `YDPresident` that inherits from `NSObject` to hold the first name and last name of the president. Define a static method to create a `YDPresident` object and assign the `firstName` and `lastName` properties. The `YDPresident.h` class is shown in Listing 2-23.

### LISTING 2-23: Chapter2/PresidentSearch/YDPresident.h

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

@interface YDPresident : NSObject{
    NSString *firstName;
    NSString *lastName;
}

@property (nonatomic, retain) NSString *firstName;
@property (nonatomic, retain) NSString *lastName;

+ (YDPresident *)presidentWithFirstName:(NSString *)firstName
                                lastName:(NSString *)lastName;

@end
```

The implementation of the `YDPresident.m` file is straightforward and is shown in Listing 2-24.

**LISTING 2-24:** Chapter2/PresidentSearch/YDPresident.m

```
#import "YDPresident.h"

@implementation YDPresident
@synthesize firstName, lastName;

+ (YDPresident *)presidentWithFirstName:(NSString *)firstName
                                lastName:(NSString *)lastName
{
    YDPresident *president = [[YDPresident alloc] init];
    president.firstName = firstName;
    president.lastName = lastName;
    return president;
}
@end
```

Open the `YDViewController.h` file and create an `NSArray` property called `filteredPeople` that contains the search results, and a strong `NSArray` property called `presidents` that will contain the entire `YDPresident` collection. The `YDViewController.h` class is shown in Listing 2-25.

**LISTING 2-25:** Chapter2/PresidentSearch/YDViewController.h

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

@interface YDViewController : UIViewController

@property(nonatomic, strong) UITableView *mTableView;
@property (nonatomic, strong) NSArray *presidents;
@property (nonatomic, strong) NSArray *filteredPresidents;
@property (nonatomic, retain) UISearchDisplayController *searchDisplayController;
@end
```

In the `YDViewController.m` implementation you subscribe to the `UITableViewDelegate`, `UITableViewDataSource`, `UISearchBarDelegate`, and `UISearchDisplayDelegate` protocols. In the `viewDidLoad` method, create the `mTableView` instance and place it on the screen by adding it as a Subview. Create the `presidents` array and fill it with an `YDPresident` object for each president of the U.S. Create a `UISearchBar` and assign it to the `tableHeaderView` property of your `mTableView`.

As a last step in your `viewDidLoad` method, create a `UISearchDisplayController` and set its `datasource` and `delegate` properties.

The implementation of the `UITableView` delegates retrieves the information from either the `people` array or the `filteredpresidents` array. The source of data depends on whether or not the `tableview` is the `SearchResultsTableView` of the `searchDisplayController`. The complete implementation is shown in Listing 2-26.

**LISTING 2-26:** Chapter2/PresidentSearch/YDViewController.m

```

#import "YDViewController.h"
#import "YDPresident.h"
@interface YDViewController ()<UITableViewDataSource, UITableViewDelegate,
UISearchBarDelegate, UISearchDisplayDelegate>

@end

@implementation YDViewController

@synthesize searchDisplayController;
- (void)viewDidLoad
{
    [super viewDidLoad];
    CGRect appframe=[[UIScreen mainScreen] applicationFrame];
    //create your tableview
    self.mTableView=[[UITableView alloc] initWithFrame:
                    CGRectMake(0,45,appframe.size.width,appframe.size.height-44)
                    style:UITableViewStylePlain];
    self.mTableView.dataSource=self;
    self.mTableView.delegate=self;
    [self.view addSubview:self.mTableView];

    //create an array of YDPresident objects for each president of the USA
    self.presidents=[[NSArray alloc] initWithObjects:
                      [YDPresident presidentWithFirstName:@"George" lastName:@"Washington"],
                      [YDPresident presidentWithFirstName:@"John" lastName:@"Adams"],
                      [YDPresident presidentWithFirstName:@"Thomas" lastName:@"Jeffeson"],
                      [YDPresident presidentWithFirstName:@"James" lastName:@"Madison"],
                      [YDPresident presidentWithFirstName:@"James" lastName:@"Monroe"],
                      [YDPresident presidentWithFirstName:@"John Quincy" lastName:@"Adams"],
                      [YDPresident presidentWithFirstName:@"Andrew" lastName:@"Jackson"],
                      [YDPresident presidentWithFirstName:@"Martin" lastName:@"van Buren"],
                      [YDPresident presidentWithFirstName:@"William Henry" lastName:@"Harrison"],
                      [YDPresident presidentWithFirstName:@"John" lastName:@"Tyler"],
                      [YDPresident presidentWithFirstName:@"James K" lastName:@"Polk"],
                      [YDPresident presidentWithFirstName:@"Zachary" lastName:@"Taylor"],
                      [YDPresident presidentWithFirstName:@"Millard" lastName:@"Fillmore"],
                      [YDPresident presidentWithFirstName:@"Franklin" lastName:@"Pierce"],
                      [YDPresident presidentWithFirstName:@"James" lastName:@"Buchanan"],
                      [YDPresident presidentWithFirstName:@"Abraham" lastName:@"Lincoln"],
                      [YDPresident presidentWithFirstName:@"Andrew" lastName:@"Johnson"],
                      [YDPresident presidentWithFirstName:@"Ulysses S" lastName:@"Grant"],
                      [YDPresident presidentWithFirstName:@"Rutherford B" lastName:@"Hayes"],
                      [YDPresident presidentWithFirstName:@"James A" lastName:@"Garfield"],
                      [YDPresident presidentWithFirstName:@"Chester A" lastName:@"Arthur"],
                      [YDPresident presidentWithFirstName:@"Grover" lastName:@"Cleveland"],
                      [YDPresident presidentWithFirstName:@"Bejamin" lastName:@"Harrison"],
                      [YDPresident presidentWithFirstName:@"Grover" lastName:@"Cleveland"],
                      [YDPresident presidentWithFirstName:@"William" lastName:@"McKinley"],
                      [YDPresident presidentWithFirstName:@"Theodore" lastName:@"Roosevelt"],
                      [YDPresident presidentWithFirstName:@"William Howard" lastName:@"Taft"],
                      [YDPresident presidentWithFirstName:@"Woodrow" lastName:@"Wilson"],
                      [YDPresident presidentWithFirstName:@"Warren G" lastName:@"Harding"],

```

```

[YDPresident presidentWithFirstName:@"Calvin" lastName:@"Coolidge"],
[YDPresident presidentWithFirstName:@"Herbert" lastName:@"Hoover"],
[YDPresident presidentWithFirstName:@"Franklin D" lastName:@"Roosevelt"],
[YDPresident presidentWithFirstName:@"Harry S" lastName:@"Truman"],
[YDPresident presidentWithFirstName:@"Dwight D" lastName:@"Eisenhower"],
[YDPresident presidentWithFirstName:@"John F" lastName:@"Kennedy"],
[YDPresident presidentWithFirstName:@"Lyndon B" lastName:@"Johnson"],
[YDPresident presidentWithFirstName:@"Richard" lastName:@"Nixon"],
[YDPresident presidentWithFirstName:@"Gerald" lastName:@"Ford"],
[YDPresident presidentWithFirstName:@"Jimmy" lastName:@"Carter"],
[YDPresident presidentWithFirstName:@"Ronald" lastName:@"Reagan"],
[YDPresident presidentWithFirstName:@"George H W" lastName:@"Bush"],
[YDPresident presidentWithFirstName:@"Bill" lastName:@"Clinton"],
[YDPresident presidentWithFirstName:@"George W" lastName:@"Bush"],
[YDPresident presidentWithFirstName:@"Barack" lastName:@"Obama"],
nil];

UISearchBar *mySearchBar = [[UISearchBar alloc] init];
mySearchBar.delegate = self;
[mySearchBar setAutocapitalizationType:UITextAutocapitalizationTypeNone];
[mySearchBar sizeToFit];
self.mTableView.tableHeaderView = mySearchBar;
// programmatically set up search display controller
self.searchDisplayController = [[UISearchDisplayController alloc]
    initWithSearchBar:mySearchBar contentsController:self];
[self setSearchDisplayController:self.searchDisplayController];
//set the delegate and data source
[self.searchDisplayController setDelegate:self];
[self.searchDisplayController setSearchResultsDataSource:self];
}

#pragma mark Table view delegates
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    if (tableView == self.searchDisplayController.searchResultsTableView)
    {
        return [self.filteredPresidents count];
    }
    else
    {
        return [self.presidents count];
    }
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];

```

*continues*

**LISTING 2-26** (continued)

```

    }
    YDPresident *president;
    if (tableView == self.searchDisplayController.searchResultsTableView)
    {
        //search result
        president = [self.filteredPresidents objectAtIndex:indexPath.row];
    }
    else
    {
        //normal result
        president = [self.presidents objectAtIndex:indexPath.row];
    }
    if (president)
        cell.textLabel.text = [NSString stringWithFormat:@"%@ %@", president.firstName, president.lastName];
    return cell;
}

#pragma mark Content Filtering

- (void)filterContentForSearchText:(NSString *)searchText scope:(NSString *)scope
{
    //create an NSPredicate to search the properties firstName and lastName with
    //an OR operator
    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"firstName CONTAINS[cd] %@ OR lastName CONTAINS[cd]
        %@", searchText, searchText];
    self.filteredPresidents = [self.presidents
        filteredArrayUsingPredicate:predicate];
}

#pragma mark UISearchDisplayController Delegate Methods

- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
shouldReloadTableForSearchString:(NSString *)searchString
{
    [self filterContentForSearchText:searchString scope:
        [[self.searchDisplayController.searchBar scopeButtonTitles]
        objectAtIndex:[self.searchDisplayController.searchBar
        selectedScopeButtonIndex]]];
    // Return YES to cause the search result table view to be reloaded.
    return YES;
}

- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
shouldReloadTableForSearchScope:(NSInteger)searchOption
{
    [self filterContentForSearchText:[self.searchDisplayController.searchBar text]
    scope:[[self.searchDisplayController.searchBar scopeButtonTitles]
    objectAtIndex:searchOption]];
    // Return YES to cause the search result table view to be reloaded.
    return YES;
}

- (void)didReceiveMemoryWarning
{
}

```

```
[super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

The result of your efforts is shown in Figure 2-14.



FIGURE 2-14

When you start typing in the search bar, it will respond in a way similar to what you see in Figure 2-15.

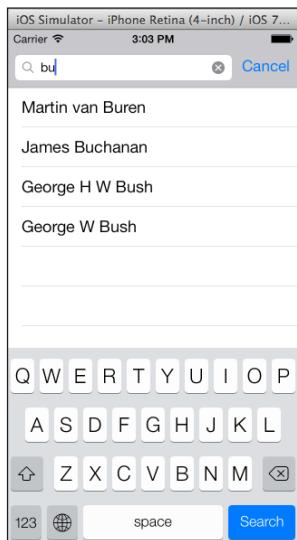


FIGURE 2-15

## Adding an Alphabet Index

When your UITableView contains a lot of rows like all the presidents of the United States of America, you can consider implementing an alphabet index, like the iPhone contacts application does. The result would be similar to what's shown in Figure 2-16.

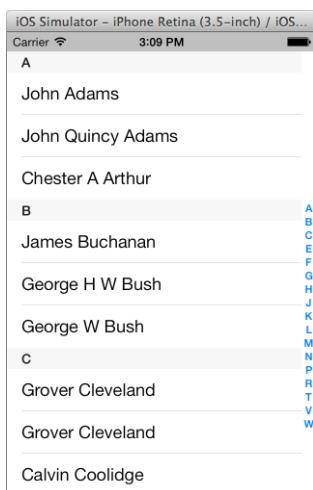


FIGURE 2-16

Start Xcode and create a new project using Single View Application Project template, and name it `IndexedTable` using the options shown in Figure 2-17.

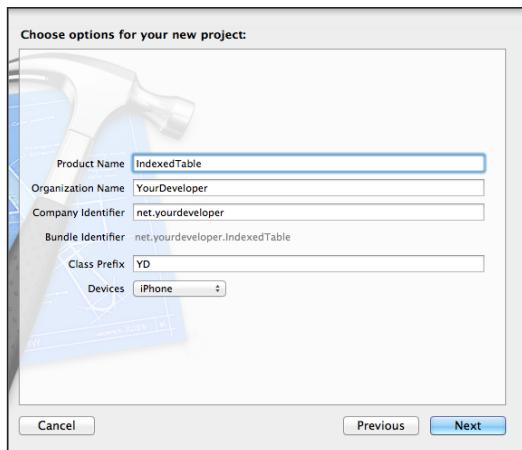


FIGURE 2-17

Import the previously created `YDPresident` class into your project, because you'll be using the same data to display.

Open your `YDViewController.h` file and create the properties as shown in Listing 2-27.

#### LISTING 2-27: Chapter2/IndexedTable/YDViewController.h

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

@interface YDViewController : UIViewController

@property(nonatomic, strong) UITableView *mTableView;
@property (weak, nonatomic) IBOutlet UITableView *mTableView;
@property (nonatomic, strong) NSMutableArray *alphabetArr;
@end
```

In the implementation of the `YDViewController` create the `presidents` array in the `viewDidLoad` method. At the end of the `viewDidLoad` method create the `alphabetArr` by processing all objects in the `presidents` array and take the first letter of the last name of the president. If this letter is not yet in the `alphabetArr`, add it. Next, sort the `alphabetArr` using an `NSSortDescriptor` into a temporary array and re-initialize the `alphabetArr` with the sorted result.

Three new `UITableView` delegate methods are implemented:

- `sectionIndexTitlesForTableView:`
- `sectionForSectionIndexTitle:`
- `tableView:titleForHeaderInSection:`

In the `numberOfRowsInSection:` method, use the current letter from the alphabet to build an `NSPredicate` on the `presidents` array and count the number of objects that are returned.

Finally, the `cellForRowAtIndexPath:` method also uses an `NSPredicate` on the last name property of the `presidents` array for the current letter in the alphabet and returns the first name and last name of the president if the `filteredArrayUsingPredicate` returns a value. The complete implementation of the `YDViewController` is shown in Listing 2-28.

#### LISTING 2-28: Chapter2/IndexedTable/YDViewController.m

```
#import "YDViewController.h"
#import "YDPresident.h"
@interface YDViewController ()<UITableViewDataSource, UITableViewDelegate>

@end

@implementation YDViewController

- (void)viewDidLoad
```

*continues*

**LISTING 2-28 (continued)**

```
{
    [super viewDidLoad];
    CGRect appframe=[[UIScreen mainScreen] applicationFrame];
    //create your tableview
    self.mTableView=[[UITableView alloc] initWithFrame:
    CGRectMake(0,0,appframe.size.width,appframe.size.height-44)
    style:UITableViewStylePlain];
    self.mTableView.dataSource=self;
    self.mTableView.delegate=self;
    [self.view addSubview:self.mTableView];
    //create an array of YDPresident objects for each president of the USA

    self.presidents=[[NSArray alloc] initWithObjects:
    [YDPresident presidentWithFirstName:@"George" lastName:@"Washington"],
    [YDPresident presidentWithFirstName:@"John" lastName:@"Adams"],
    [YDPresident presidentWithFirstName:@"Thomas" lastName:@"Jeffeson"],
    [YDPresident presidentWithFirstName:@"James" lastName:@"Madison"],
    [YDPresident presidentWithFirstName:@"James" lastName:@"Monroe"],
    [YDPresident presidentWithFirstName:@"John Quincy" lastName:@"Adams"],
    [YDPresident presidentWithFirstName:@"Andrew" lastName:@"Jackson"],
    [YDPresident presidentWithFirstName:@"Martin" lastName:@"van Buren"],
    [YDPresident presidentWithFirstName:@"William Henry" lastName:@"Harrison"],
    [YDPresident presidentWithFirstName:@"John" lastName:@"Tyler"],
    [YDPresident presidentWithFirstName:@"James K" lastName:@"Polk"],
    [YDPresident presidentWithFirstName:@"Zachary" lastName:@"Taylor"],
    [YDPresident presidentWithFirstName:@"Millard" lastName:@"Fillmore"],
    [YDPresident presidentWithFirstName:@"Franklin" lastName:@"Pierce"],
    [YDPresident presidentWithFirstName:@"James" lastName:@"Buchanan"],
    [YDPresident presidentWithFirstName:@"Abraham" lastName:@"Lincoln"],
    [YDPresident presidentWithFirstName:@"Andrew" lastName:@"Johnson"],
    [YDPresident presidentWithFirstName:@"Ulysses S" lastName:@"Grant"],
    [YDPresident presidentWithFirstName:@"Rutherford B" lastName:@"Hayes"],
    [YDPresident presidentWithFirstName:@"James A" lastName:@"Garfield"],
    [YDPresident presidentWithFirstName:@"Chester A" lastName:@"Arthur"],
    [YDPresident presidentWithFirstName:@"Grover" lastName:@"Cleveland"],
    [YDPresident presidentWithFirstName:@"Bejamin" lastName:@"Harrison"],
    [YDPresident presidentWithFirstName:@"Grover" lastName:@"Cleveland"],
    [YDPresident presidentWithFirstName:@"William" lastName:@"McKinley"],
    [YDPresident presidentWithFirstName:@"Theodore" lastName:@"Roosevelt"],
    [YDPresident presidentWithFirstName:@"William Howard" lastName:@"Taft"],
    [YDPresident presidentWithFirstName:@"Woodrow" lastName:@"Wilson"],
    [YDPresident presidentWithFirstName:@"Warren G" lastName:@"Harding"],
    [YDPresident presidentWithFirstName:@"Calvin" lastName:@"Coolidge"],
    [YDPresident presidentWithFirstName:@"Herbert" lastName:@"Hoover"],
    [YDPresident presidentWithFirstName:@"Franklin D" lastName:@"Roosevelt"],
    [YDPresident presidentWithFirstName:@"Harry S" lastName:@"Truman"],
    [YDPresident presidentWithFirstName:@"Dwight D" lastName:@"Eisenhower"],
    [YDPresident presidentWithFirstName:@"John F" lastName:@"Kennedy"],
    [YDPresident presidentWithFirstName:@"Lyndon B" lastName:@"Johnson"],
    [YDPresident presidentWithFirstName:@"Richard" lastName:@"Nixon"],
    [YDPresident presidentWithFirstName:@"Gerald" lastName:@"Ford"],
```

```

[YDPresident presidentWithFirstName:@"jimmy" lastName:@"Carter"] ,
[YDPresident presidentWithFirstName:@"Ronald" lastName:@"Reagan"] ,
[YDPresident presidentWithFirstName:@"George H W" lastName:@"Bush"] ,
[YDPresident presidentWithFirstName:@"Bill" lastName:@"Clinton"] ,
[YDPresident presidentWithFirstName:@"George W" lastName:@"Bush"] ,
[YDPresident presidentWithFirstName:@"Barack" lastName:@"Obama"] ,
nil];

//create a NSMutableArray
self.alphabetArr = [[NSMutableArray alloc] init];
for (int i=0; i<[self.presidents count]-1; i++){
    //get the first char of persons lastname
    YDPresident *president = [self.presidents objectAtIndex:i];
    char letter = [president.lastName characterAtIndex:0];
    NSString *uniChar = [NSString stringWithFormat:@"%c", letter];
    //add each letter to your array
    if (![self.alphabetArr containsObject:uniChar])
    {
        [self.alphabetArr addObject:uniChar];
    }
}
//Create a sortDescriptor so we can sort alphabetically
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
initWithKey:@"self" ascending:YES];
NSArray *sortDescriptors = [NSArray arrayWithObject:sortDescriptor];
NSArray *sortedArray;
//sortedArray will contain the sorted alphabetArr contents
sortedArray = [self.alphabetArr sortedArrayUsingDescriptors:sortDescriptors];

//Recreate it with initWithArray from sortedArray
self.alphabetArr = [[NSMutableArray alloc]
initWithArray:sortedArray copyItems:YES];
}

#pragma mark for Indexing UITableView
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView {
    return self.alphabetArr;
}
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [self.alphabetArr count];
}
- (NSInteger)tableView:(UITableView *)tableView
sectionForSectionIndexTitle:(NSString *)title
atIndex:(NSInteger)index {
    return index;
}
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    return [self.alphabetArr objectAtIndex:section];
}

#pragma mark Table view delegates

```

*continues*

**LISTING 2-28 (continued)**

```

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    NSString *alphabet = [self.alphabetArr objectAtIndex:section];
    //---build a predicate to filter---
    NSPredicate *predicate =
        [NSPredicate predicateWithFormat:
            @"lastName BEGINSWITH [cd] %@", alphabet];
    NSArray *tmp = [_presidents filteredArrayUsingPredicate:predicate];
    //---return the number of Presidents beginning with the letter---
    return [tmp count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    YDPresident* president;
    //get the letter in the current section
    NSString *alphabet = [_alphabetArr objectAtIndex:[indexPath section]];
    //get all president who's lastname begins with this letter
    NSPredicate *predicate =
        [NSPredicate predicateWithFormat:@"lastName BEGINSWITH [cd] %@", alphabet];
    NSArray *tmp = [self.presidents filteredArrayUsingPredicate:predicate];
    if ([tmp count]>0)
        president = [tmp objectAtIndex:indexPath.row];

    if (president)
        cell.textLabel.text =
            [NSString stringWithFormat:@"%@ %@", president.firstName,
                president.lastName];
    return cell;
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

## SUMMARY

In this chapter you created several applications that extend the default usage of a `UITableViewController`. You developed:

- A `ChatView` application with a custom `UITableView`, `datasource`, and custom `UITableViewCellCells`.
- A drill-down application that enables you to create a more advanced way of presenting data that can be grouped together by implementing a custom header view with a `UITapGestureRecognizer`.
- An implementation that adds search capabilities to your `UITableView`.
- Last but not least, you implemented an alphabet index on a `UITableView` to present a different kind of navigation.

In the next chapter you learn all the techniques to make full use of the `MapKit` framework. You start by developing a GPS location simulator that will be very useful during the design and testing of location-based applications.

Next you learn how to work with custom annotations and display them on the `MapView`. Finally, to support implementations with many different annotations, you develop a custom solution that clusters annotations if they are too close together and automatically unfolds upon zooming.



# 3

## Advancing with Map Kit

### **WHAT'S IN THIS CHAPTER?**

---

- Creating a simulator to help you test your GPS-based applications
- Implementing custom annotations and responding to annotations
- Developing a custom implementation to group annotations together on a custom `MKMapView`

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 3 download and individually named according to the names throughout the chapter.

Map Kit is the name of the framework for displaying and working with maps in your iOS application. Up until iOS version 6.x, these maps were provided by Google and were known as Google Maps. As of iOS version 6.x, Apple has implemented its own maps server, using maps from the Dutch provider TomTom, which is also known for its GPS devices. There has been a lot of press coverage since this new map service was released, because many users complained that information was not as accurate as with Google Maps. Apple has promised to improve the quality of the maps, but as a developer you really don't have a choice because you can't select which maps provider you want to use. Both solutions are based on the Map Kit framework, and the way a map is displayed in your application is strictly related to the iOS version of the user's device.

The Map Kit framework provides an interface for embedding maps into your application. The framework consists of a series of classes, of which `MKMapView` is the class that you can directly put on a view.

Most of the other classes relate to presenting more detailed information on the `MKMapView`, such as annotations and overlays.

You can find the full Map Kit documentation at [http://developer.apple.com/library/ios/#documentation/MapKit/Reference/MapKit\\_Framework\\_Reference/index.html](http://developer.apple.com/library/ios/#documentation/MapKit/Reference/MapKit_Framework_Reference/index.html).

The techniques explained in this chapter are the basis for a USA Restaurant application you develop and extend in the following chapters.

## SIMULATING IOS DEVICE MOVEMENT

In this section you learn how to build a GPS simulator that reads a set of coordinates from a text file and simulates your device moving around following those coordinates.

### Why You Need a GPS Simulator

In 2011 a client asked me to develop an application for a tourist organization that had created several routes, each containing multiple points of interest. The functional requirements stated that the application should guide the user to the start of the route, and when the user was near a point of interest (100 meters), the device should notify the user with a vibrating signal and open the details of this point of interest.

Developing the application was not so hard, but when I needed to test the application, especially the proximity test in relation to a point of interest, I had a problem. The routes that needed to be presented and tested on the device were more than 1,000 kilometers away from my location, so I had no way to simulate driving the actual route with a device. I needed to come up with a solution to simulate my actually being there with my device.

For this purpose, I developed a GPS simulator that is capable of simulating my device being somewhere I want to be, and uses an input file created with Google Maps that simulates my being on the move.

Xcode and the iOS Simulator have had a built-in capability to work with GPX files since iOS 5.0. GPX stands for *GPS eXchange Format*, a device-independent data format used for GPS devices. You can use Xcode to create a GPX file; an XML file containing a set of coordinates and attach it to the simulator. I chose a custom implementation to have maximum control over the implementation that mimics the movement of the device.

### Creating the Simulator

The simulator itself is an Objective-C subclass that inherits from `CLLocationManager` and implements various methods of the `CLLocationManagerDelegate` protocol. A special method is implemented that loads data from a simple text file containing a set of GPS coordinates.

Create a new project in Xcode using the Single View Application Project template and name it `GPSSimulator` using the project options shown in Figure 3-1.

Create a new Objective-C class that inherits from `CLLocationManager` and name it `YDLocationSimulator`.

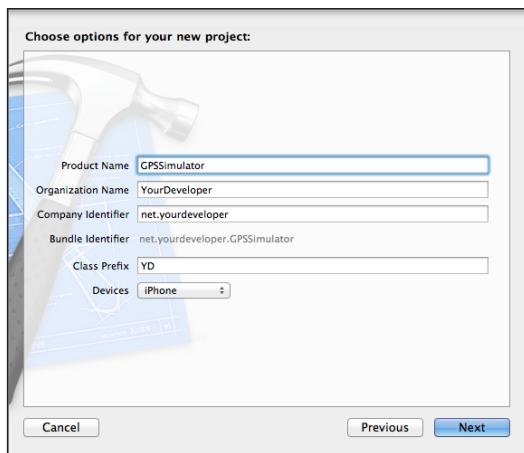


FIGURE 3-1

Now import the `MapKit.framework` and the `CoreLocation.framework` into your project.

The `YDLocationSimulator.h` file is shown in Listing 3-1.

#### LISTING 3-1: Chapter3/GPSSimulator/YDSimulator.h

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
@interface YDLocationSimulator : CLLocationManager
@property (nonatomic, strong) CLLocation *currentlocation;
@property (nonatomic, strong) CLLocation *previousLocation;
@property (nonatomic, strong) MKMapView *mapView;
@property (nonatomic, assign) BOOL bKeepRunning;
    //You can use the class as a singleton
+ (YDLocationSimulator*)sharedInstance;
//this methods loads data from a file in your NSBundle
-(void)loadLocationFile:(NSString *)pathToFile;
-(void)startUpdatingLocation;
-(void)stopUpdatingLocation;
@end
```

You create two properties of type `CLLocation` to keep track of the current and the previous location, and a property of type `MKMapView` that will be set by the `ViewController` containing the `MKMapView` object to display the simulated location.

The class is designed to operate as a singleton by defining the `+ (YDLocationSimulator*) - sharedInstance` definition.

A singleton is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

The implementation of the `YDLocationSimulator` contains the `loadLocationFile:` method that reads a file with coordinates from your `NSBundle` and uses those coordinates to simulate movement

of the device. The `fakeNewLocation` method is doing the real work here. It starts by informing the delegates with the location information, and then it selects the next location from the array of locations created during the `loadLocationFile:` method. The `bKeepRunning` boolean is used to check if it should create a continuous loop, after all locations have been processed.

Open the `YDLocationSimulator.m` implementation file and implement the code as shown in Listing 3-2.

**LISTING 3-2: Chapter3/GPSSimulator/YDLocationSimulator.m**

```
#import "YDLocationSimulator.h"

@implementation YDLocationSimulator
{
    @private
        id<CLLocationManagerDelegate> delegate;
        YDLocationSimulator *sharedInstance;
        BOOL updatingLocation;
        NSMutableArray *fakeLocations;
        NSInteger index;
        NSTimeInterval updateInterval;
        CLLocationDistance distanceFilter;
}
static YDLocationSimulator *sharedInstance = nil;

+ (YDLocationSimulator*) sharedInstance
{
    if (sharedInstance == nil) {
        sharedInstance = [[super allocWithZone:NULL] init];
    }
    return sharedInstance;
}
- (void)loadLocationFile:(NSString *)pathToFile
{
    // read everything from the file
    NSString* fileContents =
    [NSString stringWithContentsOfFile:pathToFile
        encoding:NSUTF8StringEncoding error:nil];
    // first, separate by new line
    NSArray* allLines =
    [fileContents componentsSeparatedByCharactersInSet:
        [NSCharacterSet newlineCharacterSet]];
    //If the fakeLocations array is nil create it
    if (fakeLocations == nil)
        fakeLocations = [[NSMutableArray alloc] init];
    //Parse each line and add the coordinate to the array
    for (int i=0;i<[allLines count];i++)
    {
        NSString *line = [[allLines objectAtIndex:i]
            stringByReplacingOccurrencesOfString:@" " withString:@""];
        NSArray *latlong = [line componentsSeparatedByString:@","];
        CLLocationDegrees lat = [[latlong objectAtIndex:1] doubleValue];
        CLLocationDegrees lon = [[latlong objectAtIndex:0] doubleValue];
        CLLocation* loc = [[CLLocation alloc] initWithLatitude:lat longitude:lon];
        [delegate locationManager:delegate didUpdateLocation:loc];
        if (index < updateInterval)
            index++;
        else
            [NSThread sleepForTimeInterval:updateInterval];
    }
}
```

```
        [fakeLocations addObject:loc];
    }
}

- (void)fakeNewLocation {
    if ((!self.previousLocation || [self.currentlocation
        distanceFromLocation:self.previousLocation] > distanceFilter) ) {
        if (!self.previousLocation)
            self.previousLocation= self.currentlocation;
        //Create an NSArray with the old location and new location and call
        //the delegate
        NSArray* locs = [[NSArray alloc]
            initWithObjects:self.previousLocation,
                        self.currentlocation,nil];
        [self.delegate locationManager:nil didUpdateLocations:locs];
        self.previousLocation = self.currentlocation;

    }
    // update the userlocation in the mapView if one is assigned
    if (_mapView) {
        [self.mapView.userLocation setCoordinate:self.currentlocation.coordinate];
    }
    // iterate to the next fake location
    if (updatingLocation) {
        index++;
        if (index == fakeLocations.count) {
            index = 0;
            if (!_bKeepRunning)
            {
                [self stopUpdatingLocation];
                updatingLocation = NO;
            }
            else
                self.currentlocation = [fakeLocations objectAtIndex:index];
        }
        else
        {
            self.currentlocation = [fakeLocations objectAtIndex:index];
        }
        [self performSelector:@selector(fakeNewLocation)
            withObject:nil afterDelay:updateInterval];
    }
}

- (void)startUpdatingLocation {
    updatingLocation = YES;
    //updateInterval in seconds will trigger a new location every xx seconds
    updateInterval = 1.0f;
    if ([fakeLocations count]>0)
        self.currentlocation = [fakeLocations objectAtIndex:0];
    [self fakeNewLocation];
}

- (void)stopUpdatingLocation {
    updatingLocation = NO;
}

@end
```

After creating the `sharedInstance` of the `YDLocationSimulator`, the `loadLocationFile:` method is called that will read the passed file with coordinates, parse it, and store the locations from this file into the `NSMutableArray` named `fakeLocations`. When the `ViewController` calls the `startUpdatingLocation` method, the next location is assigned to `_currentlocation` from the `fakelocations` array and the `fakeNewLocation` method is called. This `fakeNewLocation` method is testing if the `currentlocation`'s distance from the `previouslocation`'s distance is larger than the defined `distanceFilter`. If this is the case, the `currentlocation` is assigned to the `previouslocation` and an array is created with the `previouslocation` and `currentlocation` and passed to the `CLLocationManager` delegate method `didUpdateLocations`. If the `mapView` property is set, the `currentlocation` is passed to the `userLocation` property of the `mapView`, simulating that the `currentlocation` is the `userlocation`. Finally, it's checked if the end of the `fakeLocations` array has been reached or not and whether the method should run in a continuous loop. If there are more locations available in the `fakelocations` array, the next object is passed to the `currentlocation` property and the method calls itself again after a delay as specified in the `updateInterval`.

The just implemented `YDLocationSimulator` can only operate properly if your application bundle contains a file with a set of GPS coordinates it can parse and use for the simulation.

## Creating a GPS Route File with Google Maps

The easiest way to create a set of GPS coordinates to use in your `YDLocationSimulator` is by visiting [maps.google.com](http://maps.google.com) and using it to create a route, as shown in Figure 3-2.

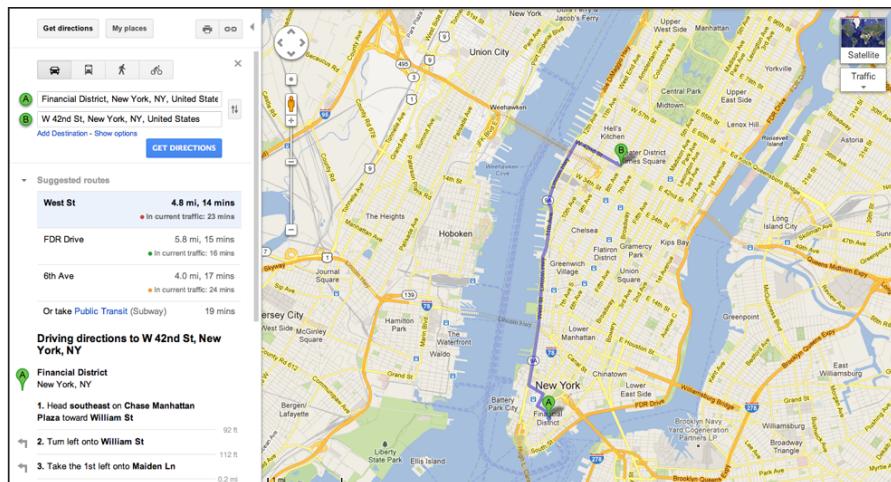


FIGURE 3-2



On the left side of the screen are the driving directions, and all the way at the bottom is a link named `Save to My Maps`. This link is available only if you are logged in with a Google account.

FIGURE 3-3

If you don't have a Google account, you can create one free. Click Save to My Maps and select Create a New Map in the drop-down box, as shown in Figure 3-3.

After this action, the screen updates as shown in Figure 3-4.

The small link named KML is a direct link to a KML file, and if you click it the file is downloaded to your computer. KML stands for *Keyhole Markup Language*, which is an XML-based format for expressing geographic data for Internet applications.

The KML format is also usable as is, but for that you need to parse the XML structure and implement some regular expressions to parse the coordinates from the file. Because XML parsing hasn't been covered yet, just open the file and start stripping all the XML-related information.

The KML file you downloaded looks similar to Listing 3-3.

### LISTING 3-3: Chapter3/GPSSimulator/coordinates.txt

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
<Document>
    <name>Driving directions to W 42nd St, New York, NY, United States</name>
    <description><![CDATA[]]></description>
    <Style id="style1">
        <IconStyle>
            <Icon>
                <href></href>
            </Icon>
        </IconStyle>
    <Style id="style2">
        <LineStyle>
            <color>73FF0000</color>
            <width>5</width>
        </LineStyle>
    <Style id="style3">
        <IconStyle>
            <Icon>
                <href></href>
            </Icon>
        </IconStyle>
    <Placemark>
        <name>From: Financial District, New York, NY, United States</name>
        <styleUrl>#style1</styleUrl>
        <Point>
            <coordinates>-74.008499,40.707916,0.000000</coordinates>
        </Point>
    </Placemark>
```



FIGURE 3-4

*continues*

**LISTING 3-3** (*continued*)

```
<Placemark>
    <name>Driving directions to W 42nd St, New York, NY, United States</name>
    <styleUrl>#style2</styleUrl>
    <ExtendedData>
        <Data name="_SnapToRoads">
            <value>true</value>
        </Data>
    </ExtendedData>
    <LineString>
        <tessellate>1</tessellate>
        <coordinates>
            -74.008499,40.707920,0.000000
            -74.008270,40.707729,0.000000
            -74.008270,40.707729,0.000000
            -74.008011,40.707958,0.000000
            -74.008011,40.707958,0.000000
            -74.009163,40.709259,0.000000
            -74.009399,40.709461,0.000000
            -74.010017,40.709770,0.000000
            -74.010017,40.709770,0.000000
            -74.011253,40.710331,0.000000
            -74.011253,40.710331,0.000000
            -74.010460,40.711521,0.000000
            -74.010117,40.712132,0.000000
            -74.009697,40.712688,0.000000
            -74.009697,40.712688,0.000000
            -74.013474,40.714378,0.000000
            -74.013474,40.714378,0.000000
            -74.012291,40.719341,0.000000
            -74.010902,40.726181,0.000000
            -74.009872,40.736969,0.000000
            -74.009827,40.737549,0.000000
            -74.009972,40.738708,0.000000
            -74.009956,40.739029,0.000000
            -74.009872,40.739391,0.000000
            -74.009872,40.739391,0.000000
            -74.009171,40.740662,0.000000
            -74.008926,40.741501,0.000000
            -74.008003,40.746078,0.000000
            -74.007652,40.748161,0.000000
            -74.007652,40.748161,0.000000
            -74.007683,40.748669,0.000000
            -74.007858,40.749119,0.000000
            -74.008186,40.749691,0.000000
            -74.008331,40.750278,0.000000
            -74.008163,40.750912,0.000000
            -74.007050,40.753811,0.000000
            -74.006638,40.754452,0.000000
            -74.002922,40.759640,0.000000
            -74.002083,40.760731,0.000000
            -74.001556,40.761490,0.000000
            -74.001083,40.761978,0.000000
            -74.001083,40.761978,0.000000
            -73.998322,40.760849,0.000000
        </coordinates>
    </LineString>
</Placemark>
```

```

        -73.989792,40.757229,0.000000
    </coordinates>
</LineString>
</Placemark>
<Placemark>
    <name>To: W 42nd St, New York, NY, United States</name>
    <styleUrl>#style3</styleUrl>
    <Point>
        <coordinates>-73.989792,40.757229,0.000000</coordinates>
    </Point>
</Placemark>
</Document>
</kml>
```

Remove everything that is not highlighted and save the file with a different name, such as coordinates.txt. The result is shown in Listing 3-4.

**LISTING 3-4: Chapter3/GPSSimulator/coordinates.txt**

```

-74.008499,40.707920,0.000000
-74.008270,40.707729,0.000000
-73.998322,40.760849,0.000000
-73.989792,40.757229,0.000000
```

Add this file to your project, and you can start implementing the YDLocationSimulator.

## Implementing the YDLocationSimulator

To use the YDLocationSimulator in your application, open the YDViewController.h file and import the YDLocationSimulator header file.

Open the YDViewController.xib file with Interface Builder and place an MKMapView object on the main view. Use the Assistant Editor to create a property and set the delegate.

The code for the YDViewController.h file is shown in Listing 3-5.

**LISTING 3-5: Chapter3/GPSSimulator/YDViewController.h**

```

#import <UIKit/UIKit.h>
#import "YDLocationSimulator.h"
#import <MapKit/MapKit.h>

@interface YDViewController : UIViewController

@property (nonatomic, weak) IBOutlet MKMapView *mapView;

@end
```

Open the YDViewController.m implementation file and declare conditionally the locationManager ivar. If the target device is the iOS Simulator, the locationManager is declared as a YDLocationSimulator; otherwise, it is of type CLLocationManager.

In the `viewDidLoad` method, initialize the `mapView` property with the starting coordinates and create the simulator by calling the `YDLocationSimulator sharedInstance` static method. Call the `loadLocationFile:` method by passing the `coordinates.txt` file and set the properties for the `locationManager`. At the end of the `viewDidLoad` method, you call the `startUpdating` method with a delay of 5 seconds so the device has some time to initialize and draw the map. The `startUpdated` method calls the `startUpdatingLocation` to start the process of updating locations, either by using the `CLLocationManager` or the `YDLocationSimulator`, depending on the target. In the `didUpdateLocations` delegate implementation you call the `updateMap` method and pass the `oldLocation` and `newLocation` so the `mapview` is updated and the region is set to center the map on the current location.

The complete `YDViewController.m` implementation is shown in Listing 3-6.

#### LISTING 3-6: Chapter3/GPSSimulator/YDViewController.m

```
#import "YDViewController.h"
@interface YDViewController : NSObject<CLLocationManagerDelegate>

@end

@implementation YDViewController
#if TARGET_IPHONE_SIMULATOR
    YDLocationSimulator *locationManager;
#else
    CLLocationManager *locationManager;
#endif
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.mapView.showsUserLocation=YES;
    CLLocationCoordinate2D lat = (double)40.709881;
    CLLocationCoordinate2D lon = (double)-74.014771;
    self.mapView.centerCoordinate = CLLocationCoordinate2DMake(lat,lon);
    self.mapView.delegate=self;
    MKCoordinateRegion region = MKCoordinateRegionMakeWithDistance(
        CLLocationCoordinate2DMake(lat,lon), 1000, 1000);
    [self.mapView setRegion:region animated:YES];
    YDLocationSimulator *simulator = [YDLocationSimulator sharedInstance];
    [simulator loadLocationFile:[[NSBundle mainBundle]
        pathForResource:@"coordinates" ofType:@"txt"]];
    locationManager = simulator;
    locationManager.delegate=self;
    locationManager.mapView=_MapView;
    locationManager.desiredAccuracy = kCLLocationAccuracyBestForNavigation;
    //wait 5 seconds for the map to draw itself
    [self performSelector:@selector(startUpdating) withObject:nil afterDelay:5.0f];
}

- (void)startUpdating
{
    [locationManager startUpdatingLocation];
}
#pragma CLLocationManager delegates
- (void)locationManager:(CLLocationManager *)manager
```

```

        didUpdateLocations:(NSArray *)locations
    {
        CLLocation* oldLocation = [locations objectAtIndex:0];
        CLLocation* newLocation = [locations lastObject];
        [self updateMap:oldLocation andNewLocation:newLocation];

    }

- (void)updateMap:(CLLocation*)oldLocation andNewLocation:(CLLocation*) newLocation
{
    if (newLocation)
    {
        // make sure the old and new coordinates are different
        if ((oldLocation.coordinate.latitude != newLocation.coordinate.latitude) &&
            (oldLocation.coordinate.longitude != newLocation.coordinate.longitude))
        {
            MKCoordinateRegion region =
            MKCoordinateRegionMakeWithDistance(
            newLocation.coordinate, 100, 100);
            [_mapView setRegion:region animated:YES];
        }
    }
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

The result of your efforts is shown in Figure 3-5.



**FIGURE 3-5**

## WORKING WITH ANNOTATIONS

Annotations are used on an `MKMapView` to indicate, for example, a point of interest or a data point coming from your data set that is linked to the `MKMapView`.

Next, you develop an application that displays custom annotations, as shown in Figure 3-6.

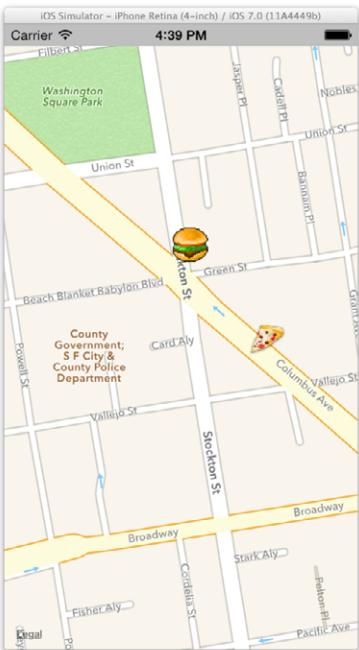


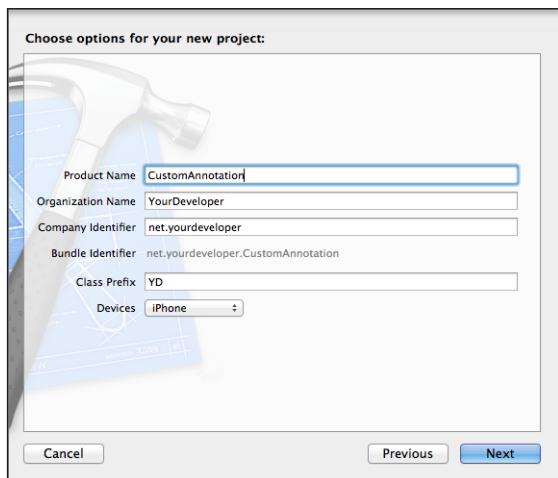
FIGURE 3-6

## Creating Custom Annotations

By default, annotations look like pins placed on an `MKMapView`. However, in most implementation scenarios you would like to display customized annotations that are more in line with the user interface design you are implementing throughout the rest of the application.

In this section, you create a simple application that shows two restaurants in San Francisco.

Start a new project in Xcode using the Single View Application Project template and name it `CustomAnnotation` using the project options shown in Figure 3-7.



**FIGURE 3-7**

Add the `MapKit.framework` and the `CoreLocation.framework` to your project.

Create a new Objective-C class that inherits from `NSObject` and call it `YDRestaurantAnnotation`. The `YDRestaurantAnnotation` object subscribes to the `MKAnnotation` protocol and has variables for the coordinate, a title, an address, and a category.

Open the `YDRestaurantAnnotation.h` file and modify it as shown in Listing 3-7.

**LISTING 3-7: Chapter3/CustomAnnotation/YDRestaurantAnnotation.h**

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

@interface YDRestaurantAnnotation : NSObject<MKAnnotation>

- (id)initWithLat:(float)lat longitude:(float)lon
    title:(NSString *)restaurantTitle address:(NSString *)restaurantAddress
    category:(NSString *)restaurantCategory;
@property (nonatomic, strong) NSString *title;
@property (nonatomic, strong) NSString *address;
@property (nonatomic, strong) NSString *category;
@property (nonatomic, assign) float latitude;
@property (nonatomic, assign) float longitude;
@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;

@end
```

Now you can open the `YDRestaurantAnnotation.m` implementation file and modify the code as shown in Listing 3-8. The constructor simply assigns the passed values to the properties.

**LISTING 3-8:** Chapter3/CustomAnnotation/YDRestaurantAnnotation.m

```
#import "YDRestaurantAnnotation.h"

@implementation YDRestaurantAnnotation

- (CLLocationCoordinate2D)coordinate;
{
    CLLocationCoordinate2D theCoordinate;
    theCoordinate.latitude = self.latitude;
    theCoordinate.longitude = self.longitude;
    return theCoordinate;
}
- (id)initWithLat:(float)lat longitude:(float)lon
    title:(NSString *)restaurantTitle address:(NSString *)restaurantAddress
    category:(NSString *)restaurantCategory
{
    self = [super init];
    if (self != nil)
    {
        self.latitude=lat;
        self.longitude=lon;
        self.address=restaurantAddress ;
        self.title=restaurantTitle ;
        self.category=restaurantCategory;
    }
    return self;
}
@end
```

In your `YDViewController`, you use `NSMutableArray` to store a collection of restaurants and you use an `MKMapView` object to display the map. The `YDViewController` must subscribe to the `MKMapViewDelegate` protocol. Now open your `YDViewController.h` file and implement the code as shown in Listing 3-9.

**LISTING 3-9:** Chapter3/CustomAnnotation/YDViewController.h

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
@interface YDViewController : UIViewController<MKMapViewDelegate>

@property (nonatomic, weak) IBOutlet MKMapView *myMapView;
@property (nonatomic, strong) NSMutableArray* restaurants;
@end
```

In your `YDViewController.m` implementation file, you start with importing the `YDRestaurantAnnotation.h` file. In the `viewDidLoad` method, center the `MKMapView` object to a location in downtown San Francisco. Call the `loadRestaurants` method that initializes the

restaurants array with two objects of type YDRestaurantAnnotation. You need to implement the vmapView:viewForAnnotation: method. First a test is performed to check if the annotation is an MKUserLocation object and, if so, a nil is returned. Second, a test is performed to check if the annotation is a YDRestaurantAnnotation object and, if so, a new MKAnnotationView object is created and the annotationView.image property is set based on the value of the category of the annotation. The complete implementation is shown in Listing 3-10.

**LISTING 3-10: Chapter3/CustomAnnotation/YDViewController.m**

```
#import "YDViewController.h"
#import "YDRestaurantAnnotation.h"
@interface YDViewController : UIViewController

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.myMapView.showsUserLocation=YES;
    [self.view addSubview:self.myMapView];
    //Center your map in SF
    MKCoordinateRegion newRegion;
    newRegion.center.latitude = 37.799052;
    newRegion.center.longitude = -122.408187;
    newRegion.span.latitudeDelta = 0.01;
    newRegion.span.longitudeDelta = 0.01;
    [self.myMapView setRegion:newRegion animated:YES];
    [self loadRestaurants];
}

-(void)loadRestaurants
{
    //Initialize your Array
    self.restaurants = [[NSMutableArray alloc] initWithObjects:
        [[YDRestaurantAnnotation alloc] initWithLat:37.799052
            longitude:-122.408187
            title:@"Calzone's Pizza Cucina "
            address:@"430 Columbus Ave, San Francisco, CA"
            category:@"pizza"],
        [[YDRestaurantAnnotation alloc] initWithLat:37.799770 longitude:-122.408936
            title:@"North Beach Restaurant"
            address:@"512 Stockton Street San Francisco, CA"
            category:@"fastfood"]
        , nil];
    [self.myMapView addAnnotations:self.restaurants];
}

#pragma delegates
- (MKAnnotationView *)mapView:(MKMapView *)theMapView
    viewForAnnotation:(id <MKAnnotation>)annotation
{
}
```

*continues*

**LISTING 3-3** (*continued*)

```

// in case it's the user location, we already have an annotation,
// so just return nil
if ([annotation isKindOfClass:[MKUserLocation class]])
    return nil;
if ([annotation isKindOfClass:[YDRestaurantAnnotation class]])
{
    YDRestaurantAnnotation *thisRestaurant = (YDRestaurantAnnotation*)annotation;
    static NSString *RestaurantAnnotationIdentifier =
    @"RestaurantAnnotationIdentifier";
    MKAnnotationView *restaurantAnnotationView =
    [self.myMapView dequeueReusableCellWithIdentifier:RestaurantAnnotationIdentifier];
    if (restaurantAnnotationView == nil)
    {
        MKAnnotationView *annotationView =
        [[MKAnnotationView alloc]
         initWithAnnotation:annotation
         reuseIdentifier:RestaurantAnnotationIdentifier];
        annotationView.canShowCallout = YES;
        if ([thisRestaurant.category isEqualToString:@"pizza"])
        {
            annotationView.image = [UIImage imageNamed:@"pizza.png"];
        }
        else if ([thisRestaurant.category isEqualToString:@"fastfood"])
        {
            annotationView.image = [UIImage imageNamed:@"fastfood.png"];
        }
        annotationView.opaque = NO;
        return annotationView;
    }
    else
    {
        restaurantAnnotationView.annotation = annotation;
    }
    return restaurantAnnotationView;
}
return nil;
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

## Responding to Annotation Call-Outs

You've created and implemented custom annotations, but you still need to respond to the call-out action of the annotation. If you start the previous application, you see two custom annotations—one with a pizza image and one with a hamburger image.

When you click either one of the images, it shows a call-out view with the title that you've passed to the title property of the `YDRestaurantAnnotation`.

Start Xcode and create a new project using the Single View Application Project template, and name it `CalloutAction` using the project options shown in Figure 3-8.

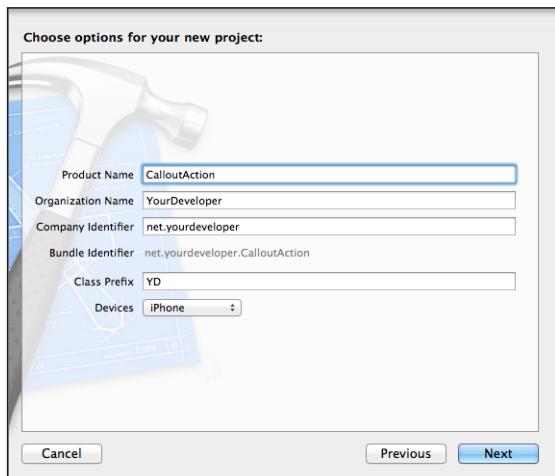


FIGURE 3-8

Import the `MapKit.framework` and delete all the `YDViewController` files from your project. Copy the following files from your previous project:

- `YDRestaurantAnnotation.h`
- `YDRestaurantAnnotation.m`
- `YDViewController.h`
- `YDViewController.m`
- `YDViewController.xib`

Create a new Objective-C class that subclasses from `UIViewController` and name it `YDDetailViewController`.

Open the `YDDetailViewController.h` file and implement the code as shown in Listing 3-11.

**LISTING 3-11:** Chapter3/CalloutAction/YDDetailViewController.h

```
#import <UIKit/UIKit.h>
#import "YDRestaurantAnnotation.h"
@interface YDDetailViewController : UIViewController

@property(nonatomic,strong) YDRestaurantAnnotation* annotation;
@end
```

In the implementation of the YDDetailViewController, add a `UIButton` object to dismiss the view, add a `UIImageView` to display the category image, and add some `UILabel` objects to display details of the `YDRestaurantAnnotation` that has been passed as a property. The complete implementation is shown in Listing 3-12.

**LISTING 3-12:** Chapter3/CalloutAction/YDDetailViewController DetailViewController.m

```
#import "YDDetailViewController.h"
#import <QuartzCore/QuartzCore.h>
@interface YDDetailViewController ()
```

```
@end
```

```
@implementation YDDetailViewController
```

```
- (id)initWithNibName:(NSString *)NibNameOrNil bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}
```

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.
    self.view.backgroundColor = [UIColor whiteColor];
    UIButton* closeButton = [[UIButton alloc]
        initWithFrame:CGRectMake(10, 10, 100, 25)];
    closeButton.titleLabel.textColor = [UIColor whiteColor];
    closeButton.backgroundColor = [UIColor blackColor];
    closeButton.layer.cornerRadius = 6.0f;
    [closeButton setTitle:@"Close" forState:UIControlStateNormal];
    [closeButton addTarget:self action:@selector(closeView)
        forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview: closeButton];

    UIImageView* categoryImage = [[UIImageView alloc]
        initWithFrame:CGRectMake(130, 10, 32, 32)];
```

```
categoryImage.backgroundColor=[UIColor clearColor];
if (_annotation.category isEqualToString:@"pizza"])
{
    categoryImage.image=[UIImage imageNamed:@"pizza.png"];
}
else
{
    categoryImage.image=[UIImage imageNamed:@"fastfood.png"];
}
[self.view addSubview:categoryImage];
UILabel* title = [[UILabel alloc]
    initWithFrame:CGRectMake(10,50,250,25)];
title.textColor=[UIColor blueColor];
title.backgroundColor=[UIColor whiteColor];
title.text = _annotation.title;
[self.view addSubview:title];

UILabel* address = [[UILabel alloc]
    initWithFrame:CGRectMake(10,80,300,25)];
address.textColor=[UIColor blueColor];
address.backgroundColor=[UIColor whiteColor];
address.text = _annotation.address;
[self.view addSubview:address];

UILabel* lat = [[UILabel alloc]
    initWithFrame:CGRectMake(10,110,300,25)];
lat.textColor=[UIColor blueColor];
lat.backgroundColor=[UIColor whiteColor];
lat.text = [NSString stringWithFormat:@"latitude: %0.6f",_annotation.latitude];
[self.view addSubview:lat];

UILabel* lon = [[UILabel alloc] initWithFrame:CGRectMake(10,140,300,25)];
lon.textColor=[UIColor blueColor];
lon.backgroundColor=[UIColor whiteColor];
lon.text = [NSString stringWithFormat:
    @"longitude: %0.6f",_annotation.longitude];
[self.view addSubview:lon];

}

-(void)closeView
{
    [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Now you need to apply some changes to your `YDViewController.m` file. First, you need to import the `YDViewController.h` file. Next, you need to update the `mapView:viewForAnnotation:` method by setting the `rightCalloutAccessoryView` to a disclosure button, and finally, you need to implement the delegate method `mapView:annotationView:calloutAccessoryTapped:` that presents the `YDDetailViewController`. The complete modified `YDViewController` implementation is shown in Listing 3-13.

**LISTING 3-13: Chapter3/CalloutAction/YDViewController.m**

```
#import "YDViewController.h"
#import "YDRestaurantAnnotation.h"
#import "YDDetailViewController.h"
@interface YDViewController : UIViewController

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.myMapView.showsUserLocation=YES;
    [self.view addSubview:_myMapView];
    //Center your map in SF
    MKCoordinateRegion newRegion;
    newRegion.center.latitude = 37.799052;
    newRegion.center.longitude = -122.408187;
    newRegion.span.latitudeDelta = 0.01;
    newRegion.span.longitudeDelta = 0.01;
    [self.myMapView setRegion:newRegion animated:YES];
    [self loadRestaurants];
}

- (void)loadRestaurants
{
    //Initialize your Array
    self.restaurants = [[NSMutableArray alloc] initWithObjects:
        [[YDRestaurantAnnotation alloc] initWithLat:37.799052
            longitude:-122.408187
            title:@"Calzone's Pizza Cucina "
            address:@"430 Columbus Ave, San Francisco, CA"
            category:@"pizza"],
        [[YDRestaurantAnnotation alloc] initWithLat:37.799770 longitude:-122.408936
            title:@"North Beach Restaurant"
            address:@"512 Stockton Street San Francisco, CA"
            category:@"fastfood"]
        , nil];
    [self.myMapView addAnnotations:self.restaurants];
}

#pragma mark delegates
- (MKAnnotationView *)mapView:(MKMapView *)theMapView viewForAnnotation:(id <MKAnnotation>)annotation
{
```

```

// in case it's the user location, we already have an annotation,
// so just return nil
if ([annotation isKindOfClass:[MKUserLocation class]])
    return nil;
if ([annotation isKindOfClass:[YDRestaurantAnnotation class]])
{
    YDRestaurantAnnotation *thisRestaurant =
        (YDRestaurantAnnotation*)annotation;
    static NSString *RestaurantAnnotationIdentifier =
        @"RestaurantAnnotationIdentifier";
    MKAnnotationView *restaurantAnnotationView =
        [self.myMapView
dequeueReusableAnnotationViewWithIdentifier:
    RestaurantAnnotationIdentifier];
    if (restaurantAnnotationView == nil)
    {
        MKAnnotationView *annotationView = [[MKAnnotationView alloc]
            initWithAnnotation:annotation
            reuseIdentifier:RestaurantAnnotationIdentifier];
        annotationView.canShowCallout = YES;
        if ([thisRestaurant.category isEqualToString:@"pizza"])
        {
            annotationView.image = [UIImage imageNamed:@"pizza.png"];
        }
        else if ([thisRestaurant.category
            isEqualToString:@"fastfood"])
        {
            annotationView.image = [UIImage
                imageNamed:@"fastfood.png"];
        }
        annotationView.opaque = NO;
        //Add this line to set a UIButton to the
        //rightCalloutAccessoryView
        annotationView.rightCalloutAccessoryView = [UIButton
            buttonWithType:UIButtonTypeDetailDisclosure];
        return annotationView;
    }
    else
    {
        restaurantAnnotationView.annotation = annotation;
    }
    return restaurantAnnotationView;
}
return nil;
}
- (void)mapView:(MKMapView *)mv annotationView:(MKAnnotationView *)pin
    calloutAccessoryControlTapped:(UIControl *)control {
    YDRestaurantAnnotation *thisRestaurantAnnotation =
        ((YDRestaurantAnnotation *)pin.annotation);
    YDDetailViewController *dvc = [[YDDetailViewController alloc] init];
    dvc.annotation=thisRestaurantAnnotation;
    [self presentViewController:dvc animated:YES completion:nil];
}

```

*continues*

**LISTING 3-13 (continued)**

```

    }

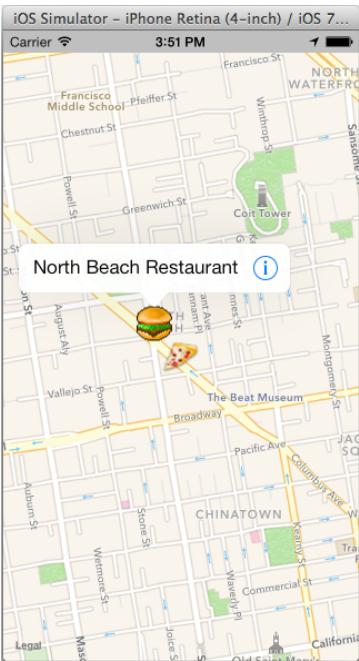
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

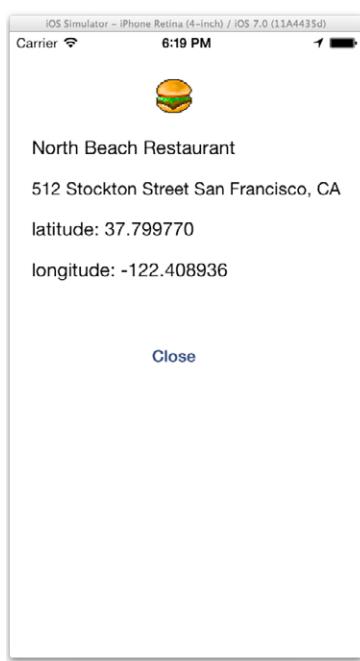
```

When you launch the application and click on one of the annotations, you will see the title and the callout button as shown in Figure 3-9.

If you click the disclosure button, the `YDDetailViewController` is pushed with the selected annotation and will display the details of the disclosed annotation as shown in Figure 3-10.



**FIGURE 3-9**



**FIGURE 3-10**

## Clustering Annotations

When many annotations are very close to each other on a `MapView`, it's impossible for the user to select a specific annotation.

Figure 3-11 shows an implementation with many different points plotted very close to each other, making it impossible for the user to select one.

In this section, you learn how to cluster annotations in such a way that they display the number of underlying points if they are too close together, as shown in Figure 3-12.



FIGURE 3-11



FIGURE 3-12

To realize this solution, develop and implement custom subclasses for the `MKMapView` and `MKAnnotationView` as well as some helper classes.

Start Xcode and create a new project using the Single View Application Project template, and name it `ClusterMap` using the settings shown in Figure 3-13.

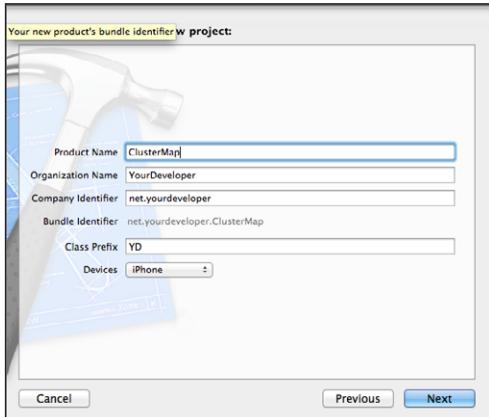


FIGURE 3-13

Create a new Objective-C class that subclasses from `MKMapView` and name it `YDClusterMapView`. Open your `YDClusterMapView.h` file and implement the code as shown in Listing 3-14.

**LISTING 3-14:** Chapter3/ClusterMap/YDClusterMapView.h

```
#import <MapKit/MapKit.h>
#import <Foundation/Foundation.h>
@interface YDClusterMapView : MKMapView<MKMapViewDelegate>
{
    NSArray *copiedAnnotations;
    double zoomLevel;
}
@property (nonatomic,assign)NSUInteger tiles;
@property (nonatomic,assign) NSUInteger minimumClusterLevel;
@property(nonatomic,assign) id<MKMapViewDelegate> delegate;
@end
```

Open the `YDClusterMapView.m` file and write the code as shown in Listing 3-15. The `YDClusterMapView` implementation has some initializers as always, and most of the magic happens in the `mapView:regionDidChangeAnimated:` and `addAnnotations:` methods. They make a call to the `YDClusterManager` with a collection of annotations.

At the `#pragma mark` section, implement the standard delegate calls for the `MKMapViewDelegate`.

The complete implementation is shown in Listing 3-15.

**LISTING 3-15:** Chapter3/ClusterMap/YDClusterMapView.m

```
#import "YDClusterManager.h"
#import "YDClusterMapView.h"
@interface YDClusterMapView (Private)
- (void) setup;
- (BOOL) mapViewDidZoom;
@end
@implementation YDClusterMapView
//@synthesize minimumClusterLevel;
//@synthesize tiles;
@synthesize delegate;

- (id) initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        [self setup];
    }
    return self;
}
```

```

- (id) initWithCoder:(NSCoder *)aDecoder
{
    self = [super initWithCoder:aDecoder];
    if (self)
    {
        [self setup];
    }
    return self;
}

- (void) setup
{
    copiedAnnotations = nil;
    self.minimumClusterLevel = 100000;
    self.tiles = 4;
    super.delegate = self;
    zoomLevel = self.visibleMapRect.size.width * self.visibleMapRect.size.height;
}

- (void) mapView:(MKMapView *)mapView regionDidChangeAnimated:(BOOL)animated
{
    if( [self mapViewDidZoom] )
    {
        //remove all the annotations
        [super removeAnnotations:self.annotations];
        self.showsUserLocation = self.showsUserLocation;
    }
    //Call YDClusterManager to return visible annotations clustered and normal
    NSArray *add = [YDClusterManager clusterAnnotationsForMapView:self
        forAnnotations:copiedAnnotations tiles:self.tiles
        minClusterLevel:self.minimumClusterLevel];
    [super addAnnotations:add];
    if( [delegate respondsToSelector:@selector(mapView:regionDidChangeAnimated:)] )
    {
        [delegate mapView:mapView regionDidChangeAnimated:animated];
    }
}

- (BOOL) mapViewDidZoom
{
    if( zoomLevel == self.visibleMapRect.size.width * self.visibleMapRect.size.height )
    {
        zoomLevel = self.visibleMapRect.size.width * self.visibleMapRect.size.height;
        return NO;
    }
    zoomLevel = self.visibleMapRect.size.width * self.visibleMapRect.size.height;
    return YES;
}

- (void) addAnnotations:(NSArray *)annotations
{
    if (copiedAnnotations!=nil)
        copiedAnnotations=nil;
}

```

*continues*

**LISTING 3-15 (continued)**

```

//copy to our private array
copiedAnnotations = [annotations copy];
//Call YDClusterManager to return visible annotations clustered and normal
NSMutableArray *add = [YDClusterManager clusterAnnotationsForMapView:self
                    forAnnotations:annotations tiles:self.tiles
                    minClusterLevel:self.minimumClusterLevel];
[super addAnnotations:add];
}

- (void) setMaximumClusterLevel:(NSUInteger)value
{
    if ( value > 100000 )
        _minimumClusterLevel = 100000;
    else
        _minimumClusterLevel = round(value);
}

- (void) setTiles:(NSUInteger)value
{
    if( value > 1024 )
        _tiles = 1024;
    else if ( value < 2 )
        _tiles = 2;
    else
        _tiles = round(value);
}

#pragma mark implement standard delegates
- (MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(id <MKOverlay>)overlay
{
    if( [delegate respondsToSelector:@selector(mapView:viewForOverlay:)] )
    {
        return [delegate mapView:mapView viewForOverlay:overlay];
    }
    return nil;
}

- (MKAnnotationView *)mapView:(MKMapView *)mapView
    viewForAnnotation:(id <MKAnnotation>)annotation
{
    if( [delegate respondsToSelector:@selector(mapView:viewForAnnotation:)] )
    {
        return [delegate mapView:mapView viewForAnnotation:annotation];
    }
    return nil;
}

```

```

- (void)mapView: (MKMapView *)mapView regionWillChangeAnimated:(BOOL)animated
{
    if( [delegate respondsToSelector:@selector(mapView:regionWillChangeAnimated:)] )
    {
        [delegate mapView:mapView regionWillChangeAnimated:animated];
    }
}

- (void)mapViewWillStartLoadingMap: (MKMapView *)mapView
{
    if( [delegate respondsToSelector:@selector(mapViewWillStartLoadingMap:)] )
    {
        [delegate mapViewWillStartLoadingMap:mapView];
    }
}
- (void)mapViewDidFinishLoadingMap: (MKMapView *)mapView
{
    if( [delegate respondsToSelector:@selector(mapViewDidFinishLoadingMap:)] )
    {
        [delegate mapViewDidFinishLoadingMap:mapView];
    }
}
- (void)mapViewDidFailLoadingMap: (MKMapView *)mapView withError: (NSError *)error
{
    if( [delegate respondsToSelector:@selector(mapViewDidFailLoadingMap:withError:)] )
    {
        [delegate mapViewDidFailLoadingMap:mapView withError:error];
    }
}
- (void)mapView: (MKMapView *)mapView didAddAnnotationViews: (NSArray *)views
{
    if( [delegate respondsToSelector:@selector(mapView:didAddAnnotationViews:)] )
    {
        [delegate mapView:mapView didAddAnnotationViews:views];
    }
}

- (void)mapView: (MKMapView *)mapView annotationView: (MKAnnotationView *)view
    calloutAccessoryControlTapped: (UIControl *)control
{
    if( [delegate respondsToSelector:@selector(mapView:annotationView:
        calloutAccessoryControlTapped:)] )
    {
        [delegate mapView:mapView annotationView:view
            calloutAccessoryControlTapped:control];
    }
}

- (void)mapView: (MKMapView *)mapView didSelectAnnotationView: (MKAnnotationView *)view
{
    if( [delegate respondsToSelector:@selector(mapView:didSelectAnnotationView:)] )
    {
        [delegate mapView:mapView didSelectAnnotationView:view];
    }
}

```

*continues*

**LISTING 3-15 (continued)**

```

}
- (void)mapView:(MKMapView *)mapView didDeselectAnnotationView:(MKAnnotationView *)view
{
    if( [delegate respondsToSelector:@selector(mapView:didDeselectAnnotationView:)] )
    {
        [delegate mapView:mapView didDeselectAnnotationView:view];
    }
}

- (void)mapViewWillStartLocatingUser:(MKMapView *)mapView
{
    if( [delegate respondsToSelector:@selector(mapViewWillStartLocatingUser:)] )
    {
        [delegate mapViewWillStartLocatingUser:mapView];
    }
}

- (void)mapViewDidStopLocatingUser:(MKMapView *)mapView
{
    if( [delegate respondsToSelector:@selector(mapViewDidStopLocatingUser:)] )
    {
        [delegate mapViewDidStopLocatingUser:mapView];
    }
}

- (void)mapView:(MKMapView *)mapView didUpdateUserLocation:
    (MKUserLocation *)userLocation
{
    if( [delegate respondsToSelector:@selector(mapView:didUpdateUserLocation:)] )
    {
        [delegate mapView:mapView didUpdateUserLocation:userLocation];
    }
}

- (void)mapView:(MKMapView *)mapView didFailToLocateUserWithError:(NSError *)error
{
    if( [delegate respondsToSelector:@selector(mapView:didFailToLocateUserWithError:)] )
    {
        [delegate mapView:mapView didFailToLocateUserWithError:error];
    }
}

- (void)mapView:(MKMapView *)mapView annotationView:(MKAnnotationView *)view
    didChangeDragState:(MKAnnotationViewDragState)newState
    fromOldState:(MKAnnotationViewDragState)oldState
{
    if( [delegate respondsToSelector:@selector(mapView:annotationView:
        didChangeDragState:fromOldState:)] )
    {
        [delegate mapView:mapView annotationView:view
            didChangeDragState:newState fromOldState:oldState];
    }
}

```

```

- (void)mapView: (MKMapView *)mapView didAddOverlayViews: (NSArray *)overlayViews
{
    if( [delegate respondsToSelector:@selector(mapView:didAddOverlayViews:)] )
    {
        [delegate mapView:mapView didAddOverlayViews:overlayViews];
    }
}

@end

```

That was a large piece of code for the `YDClusterMapView.m` implementation; let's break it down into smaller pieces to explain what's happening in this class.

The class starts with two initializers: the `initWithFrame` and the `initWithCoder`. Both initializers call the `setup` method that will assign default values to some of the ivars.

The `mapView:regionDidChangeAnimated:` method is called if the region of the `mapview` is changed. It checks if this was a zoom action on the `mapview`, and if that is true, the annotations are removed from the `mapview` object. Next, the `YDClusterManager` is called to retrieve and copy the annotations and add them to the `mapview`. The `mapViewDidZoom` method is checking if the `mapview` visible area has changed and returns a `BOOL` value.

The `addAnnotations` method accepts an `NSArray` of annotations, and will make a copy and call the `YDClusterManager` to get only the visible annotations and add them to the `mapview`.

To display your custom annotations, create a new Objective-C class named `YDClusterPin` as an `NSObject` subclass that implements the `MKAnnotation` protocol. The `YDClusterPin.h` file declares some instance variables, of which the `NSArray` `pinnodes` is the one that contains all the annotations in a clustered situation. Implement the code as shown in Listing 3-16.

#### LISTING 3-16: Chapter3/ClusterMap/YDClusterPin.h

```

#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
@interface YDClusterPin : NSObject<MKAnnotation>
{
}

@property(nonatomic, strong) NSArray* pinnodes;
@property(nonatomic, assign) CLLocationCoordinate2D coordinate;
@property(nonatomic, copy) NSString* title;
@property(nonatomic, copy) NSString* subtitle;
- (NSUInteger) nodeCount;
@end

```

Now, for the implementation of the `YDClusterPin.m` you only need to implement the `nodeCount` method that returns the number of underlying annotations, as shown in Listing 3-17.

**LISTING 3-17:** Chapter3/ClusterMap/ YDClusterPin.m

```
#import "YDClusterPin.h"

@implementation YDClusterPin

- (NSUInteger) nodeCount
{
    if( _pinnodes )
        return [_pinnodes count];
    return 0;
}
@end
```

To display your annotations, create a new Objective-C class named `YDClusterAnnotationView` that subclasses `MKAnnotationView`. The `YDClusterAnnotationView` has one simple method called `setClusterAnnotationText`, which you'll use to set the `UILabel` object to display the requested value. The code for the `YDClusterAnnotationView` is shown in Listing 3-18.

**LISTING 3-18:** Chapter3/ClusterMap/YDClusterAnnotationView.h

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
@interface YDClusterAnnotationView : MKAnnotationView<MKAnnotation> {
    UILabel* clusterlabel;
}
- (void) setClusterAnnotationText:(NSString *)text;
@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@end
```

The implementation of the `YDClusterAnnotationView.m` is straightforward, implementing the `initWithAnnotation` method. The created `clusterLabel` is used to display the value that will be set by the `setClusterAnnotationText` method. Finally, the `setClusterAnnotationText` method is implemented as shown in Listing 3-19.

**LISTING 3-19:** Chapter3/ClusterMap/YDClusterAnnotationView.m

```
#import "YDClusterAnnotationView.h"

@implementation YDClusterAnnotationView
@synthesize coordinate;
- (id) initWithAnnotation:(id<MKAnnotation>)annotation
    reuseIdentifier:(NSString *)reuseIdentifier
{
    self = [super initWithAnnotation:annotation reuseIdentifier:reuseIdentifier];
    if ( self )
    {
        clusterlabel = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, 32, 16)];
        clusterlabel.textColor = [UIColor whiteColor];
```

```

        clusterlabel.backgroundColor = [UIColor clearColor];
        clusterlabel.font = [UIFont boldSystemFontOfSize:11];
        clusterlabel.textAlignment = NSTextAlignmentCenter;
        [self addSubview:clusterlabel];
    }
    return self;
}

- (void) setClusterAnnotationText:(NSString *)text
{
    clusterlabel.text = text;
}

@end

```

Now you need a helper class that will keep track of your annotations during clustering. Create a new Objective-C class named `YDAnnotationsArray` that subclasses `NSObject`.

In the `YDAnnotationsArray` class, define an `NSMutableArray` to store your annotations and two double objects to summarize the x and y coordinates of the underlying annotations.

Modify the `YDAnnotationsArray.h` file as shown in Listing 3-20.

#### LISTING 3-20: Chapter3/ClusterMap/YDAnnotationsArray.h

```

#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
@interface YDAnnotationsArray : NSObject {
    NSMutableArray *collection;
    double totalX;
    double totalY;
}
@property (nonatomic,readonly) NSMutableArray *collection;
- (void) addObject:(id<MKAnnotation>)annotation;
- (id) objectAtIndex:(NSUInteger)index;
- (NSUInteger) count;
- (double) xPoint;
- (double) yPoint;
@end

```

The implementation of the `YDAnnotationsArray` starts with an `init` method that initializes the `NSMutableArray` and sets default values to the `totalX` and `totalY` variables. In the `addObject` method you add the annotation to the collection and calculate the `MKMapPoint` for the annotation. The `MKMapPoint` X and Y values are summarized in the `totalX` and `totalY` variables.

Implement two methods for the `xPoint` and the `yPoint` of the annotations that are stored in the collections array that return a double that will return the average values. The complete implementation is shown in Listing 3-21.

**LISTING 3-21:** Chapter3/ClusterMap/YDAnnotationsArray.m

```

#import "YDAnnotationsArray.h"
#import "YDClusterPin.h"
@implementation YDAnnotationsArray

@synthesize collection;
- (id) init
{
    self = [super init];
    if( self )
    {
        collection = [[NSMutableArray alloc] init];
        totalX = 0;
        totalY = 0;

    }
    return self;
}

- (void) addObject:(id<MKAnnotation>)annotation;
{
    [collection addObject:annotation];
    MKMapPoint mapPoint = MKMapPointForCoordinate( [annotation coordinate] );
    //increase totalX with the mappoint x coordinate for this annotation
    totalX += mapPoint.x;
    //do the same for the y coordinate
    totalY += mapPoint.y;
}
- (double)xPoint
{
    return totalX / [collection count];
}
- (double)yPoint
{
    return totalY / [collection count];
}
- (id) objectAtIndex:(NSUInteger)index
{
    return [collection objectAtIndex:index];
}
- (NSUInteger) count
{
    return [collection count];
}

@end

```

To cluster the annotations, you need both a helper class and an object to store the clustered information. Start by creating a new Objective-C class named `YDCluster` that subclasses `NSObject`. Declare an array of your custom type `YDAnnotationsArray` and a variable of type `MKMapRect`. Enter the code in `YDCluster.h` as shown in Listing 3-22.

**LISTING 3-22:** Chapter3/ClusterMap/YDCluster.h

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
#import "YDAnnotationsArray.h"
@interface YDCluster : NSObject{
    YDAnnotationsArray *annotationsArray;
}
- (void) addAnnotation:(id<MKAnnotation>)annotation;
- (id<MKAnnotation>) getClusteredAnnotation;
- (NSInteger) count;
@end
```

Open the `YDCluster.m` file and create the `addAnnotation` method that adds an annotation to your custom `NSArray`. Implement the `getClusteredAnnotation` method that returns a `YDClusteredPin` from the custom array. The complete implementation is shown in Listing 3-23.

**LISTING 3-23:** Chapter3/ClusterMap/YDCluster.m

```
#import "YDCluster.h"
#import "YDClusterPin.h"
#import "YDCluster.h"
@implementation YDCluster
- (void) addAnnotation:(id<MKAnnotation>)annotation
{
    //if annotationsArray is not initialized, create it first
    if( !annotationsArray )
    {
        annotationsArray = [[YDAnnotationsArray alloc] init];
    }
    [annotationsArray addObject:annotation];
}

- (id<MKAnnotation>) getClusteredAnnotation
{
    if( [self count] == 1 )
    {
        //if only one simply return it from the array
        return [annotationsArray objectAtIndex:0];
    } else if ( [self count] > 1 )
    {
        //create a new location based on the xPoint and yPoint from the
        //annotationsArray which are basically average x and y points
        CLLocationCoordinate2D location = MKCoordinateForMapPoint(MKMapPointMake(
            [annotationsArray xPoint], [annotationsArray yPoint]));
        YDClusterPin *pin = [[YDClusterPin alloc] init];
        pin.coordinate = location;
        pin.pinnodes = [annotationsArray collection];
        return pin;
    }
    return nil;
}
```

*continues*

**LISTING 3-23 (continued)**

```

- (NSInteger) count
{
    return [annotationsArray count];
}

@end

```

Create a new Objective-C class that subclasses `NSObject` and name it `YDClusterManager`. Implement the code as shown in Listing 3-24.

**LISTING 3-24: Chapter3/ClusterMap/YDClusterManager.h**

```

#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
#import "YDCluster.h"
#import "YDClusterPin.h"

@interface YDClusterManager : NSObject {

}

+ (NSArray *) clusterAnnotationsForMapView:(MKMapView *)mapView forAnnotations:
(NSArray *)pins tiles:(NSUInteger)tiles
minClusterLevel:(NSUInteger)minClusterLevel;

@end

```

The implementation of the `YDClusterManager` contains two methods, of which the `clusterAnnotationsForMapView` is the only one called from outside the object. It is called from your `YDClusterMapView` where the annotations are passed in. This method initializes an `NSMutableArray` named `visibleAnnotations`, which calculates various variables from the annotation's coordinates, and if the annotation's coordinate is within the visible range of the `MKMapView` region, it is added to the array.

The `clusteredTiles` array is initialized and populated with empty initialized `YDCluster` objects in a loop that will create `tiles` instances, where `tiles` is the integer value that is passed in this method's call.

Next, all `YDClusterPin` objects from the `visibleAnnotations` array are processed, and the `MKMapPoint`-related information is calculated and stored to the `YDCluster` object.

The `clusterAlreadyExistsForMapView` method will check if the passed `YDCluster` object already exists in the `mapView` annotations collection and return the result as a `BOOL`. The complete implementation is shown in Listing 3-25.

**LISTING 3-25:** Chapter3/ClusterMap/YDClusterManager.m

```
#import "YDClusterManager.h"

@implementation YDClusterManager

+ (NSArray *) clusterAnnotationsForMapView:(MKMapView *)mapView
    forAnnotations:(NSArray *)pins tiles:(NSUInteger)tiles
    minClusterLevel:(NSUInteger)minClusterLevel
{
    NSMutableArray *visibleAnnotations = [NSMutableArray array];
    double tileX = mapView.visibleMapRect.origin.x;
    double tileY = mapView.visibleMapRect.origin.y;
    float tileSize = mapView.visibleMapRect.size.width/tiles;
    float tileHeight = mapView.visibleMapRect.size.height/tiles;
    MKMapRect mapRect = MKMapRectWorld;
    NSUInteger maxWidthBlocks = round(mapRect.size.width / tileSize);
    float zoomLevel = maxWidthBlocks / tiles;
    float tileStartX = floorf(tileX/tileSize)*tileSize;
    float tileStartY = floorf(tileY/tileHeight)*tileHeight;
    MKMapRect visibleMapRect = MKMapRectMake(tileStartX, tileStartY, tileSize*(tiles+1),
        , tileHeight*(tiles+1));
    for (id<MKAnnotation> point in pins)
    {
        MKMapPoint mapPoint = MKMapPointForCoordinate(point.coordinate);
        if( MKMapRectContainsPoint(visibleMapRect, mapPoint) )
        {
            if( ![mapView.annotations containsObject:point] )
            {
                [visibleAnnotations addObject:point];
            }
        }
    }
    if( zoomLevel > minClusterLevel )
    {
        return visibleAnnotations;
    }
    NSMutableArray *clusteredTiles = [NSMutableArray array];
    int length = (tiles+1)*(tiles+1);
    for (int i=0 ; i < length ; i ++ )
    {
        YDCluster *block = [[YDCluster alloc] init];
        [clusteredTiles addObject:block];
    }
    for (YDClusterPin *pin in visibleAnnotations)
    {
        MKMapPoint mapPoint = MKMapPointForCoordinate(pin.coordinate);
        double localPointX = mapPoint.x - tileStartX;
        double localPointY = mapPoint.y - tileStartY;
        int localTileNumberX = floor( localPointX / tileSize );
        int localTileNumberY = floor( localPointY / tileHeight );
        int localTileNumber = localTileNumberX + (localTileNumberY * (tiles+1));
    }
}
```

*continues*

**LISTING 3-25** (continued)

```

        [(YDCluster *) [clusteredTiles objectAtIndex:localTileNumber] addAnnotation:pin];
    }
    //create New Pins
    NSMutableArray *newPins = [NSMutableArray array];
    for ( YDCluster *cluster in clusteredTiles )
    {
        if( [cluster count] > 0 )
        {
            if( ![self clusterAlreadyExistsForMapView:mapView andCluster:cluster] )
            {
                [newPins addObject:[cluster getClusteredAnnotation]];
            }
        }
    }
    return newPins;
}

+ (BOOL) clusterAlreadyExistsForMapView:(MKMapView *)mapView
    andCluster:(YDCluster *)cluster
{
    for ( YDClusterPin *pin in mapView.annotations )
    {
        //if YDClusterPin doesn't have pinnodes we can skip
        if( [pin isKindOfClass:[YDClusterPin class]] && [[pin pinnodes] count] > 0 )
        {
            MKMapPoint point1 = MKMapPointForCoordinate([pin coordinate]);
            MKMapPoint point2 = MKMapPointForCoordinate(
                [[cluster getClusteredAnnotation] coordinate]);
            if( MKMapPointEqualToPoint(point1,point2) )
            {
                return YES;
            }
        }
    }
    return NO;
}

@end

```

To create some annotations without too much coding, you can copy the `coordinates.txt` file used earlier into your project. Now open the `YDViewController.h` file and implement the code as shown in Listing 3-26.

Notice that you are not implementing an `MKMapView`, but your own custom `YDClusterMapView` object.

**LISTING 3-26:** Chapter3/ClusterMap/YDViewController.h

```

#import <MapKit/MapKit.h>
#import "YDClusterMapView.h"

#import <UIKit/UIKit.h>

```

```

@interface YDViewController : UIViewController<MKMapViewDelegate>

@property (nonatomic, retain) IBOutlet YDClusterMapView* myMapView;
@property(nonatomic,retain)NSMutableArray* someLocations;
@end

```

The implementation of the `YDViewController.m` file is simply importing the objects you've just developed. In the `viewDidLoad` method you create your `YDClusterMapView` object and load the annotations from the `coordinates.txt` file. The last line in `loadLocationFile:` calls `mapView:addAnnotations:someLocations`, which is picked up by your custom `YDClusterMapView` class that calls the `YDClusterManager` to create the clusters and `YDClusterPin` objects.

The `mapView:viewForAnnotation:` method converts the `MKAnnotation` to your customer `YDClusterPin` object, and if the `[pin nodeCount] > 0`, you know it's a clustered pin with underlying objects. So, you create your `YDClusterAnnotationView` and assign the correct image you want to display. In all other cases, simply return an `MKAnnotationView` with a custom image. The complete implementation is shown in Listing 3-27.

#### LISTING 3-27: Chapter3/ClusterMap/YDViewController.m

```

#import "YDViewController.h"
#import "YDClusterMapView.h"
#import "YDClusterPin.h"
#import "YDClusterAnnotationView.h"
@interface YDViewController : UIViewController<MKMapViewDelegate>

@end

@implementation YDViewController

CGRect appFrame;
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    self.view.backgroundColor=[UIColor whiteColor];
    _myMapView.delegate=self;
    _myMapView.showsUserLocation=YES;
    [self.view addSubview:_myMapView];
    [self loadLocationFile:[[NSBundle mainBundle]
        pathForResource:@"coordinates" ofType:@"txt"]];
    //Center your map in NY - Manhattan
    MKCoordinateRegion newRegion;
    newRegion.center.latitude = 40.713951;
    newRegion.center.longitude = -74.005821;
    newRegion.span.latitudeDelta = 0.1;
    newRegion.span.longitudeDelta = 0.1;
    [_myMapView setRegion:newRegion animated:YES];
}

- (void)loadLocationFile:(NSString *)pathToFile

```

*continues*

**LISTING 3-26** (continued)

```

{
    // read everything from the file
    NSString* fileContents =
    [NSString stringWithContentsOfFile:pathToFile
                                encoding:NSUTF8StringEncoding error:nil];
    // first, separate by new line
    NSArray* allLines =
    [fileContents componentsSeparatedByCharactersInSet:
     [NSCharacterSet newlineCharacterSet]];
    //If the fakeLocations array is nil create it
    if (_someLocations == nil)
        _someLocations = [[NSMutableArray alloc] init];
    //Parse each line and add the coordinate to the array
    for (int i=0;i<[allLines count];i++)
    {
        NSString *line = [[allLines objectAtIndex:i]
            stringByReplacingOccurrencesOfString:@" " withString:@""];
        NSArray *latLong = [line componentsSeparatedByString:@",,"];
        CLLocationDegrees lat = [[latLong objectAtIndex:1] doubleValue];
        CLLocationDegrees lon = [[latLong objectAtIndex:0] doubleValue];
        CLLocationCoordinate2D coord = {lat,lon};
        YDClusterPin *pin = [[YDClusterPin alloc] init];
        pin.coordinate=coord;
        pin.title = [NSString stringWithFormat:@"Pin %i",i];
        pin.subtitle = [NSString stringWithFormat:@"item %i",i];
        [_someLocations addObject:pin];
    }
    [_myMapView addAnnotations:_someLocations];
}
#pragma mark Map view delegate
-(void)mapView:(MKMapView *)mapView regionDidChangeAnimated:(BOOL)animated
{
    //Since we are changing the region let's remove all annotations and reload
    // them so again only visibles remain in memory
    [_myMapView removeAnnotations:_someLocations];
    [_myMapView addAnnotations:_someLocations];
}
-(MKAnnotationView *)mapView:(MKMapView *)mapView
viewForAnnotation:(id <MKAnnotation>)annotation
{
    //skip the User location
    if([annotation class] == MKUserLocation.class) {
        return nil;
    }
    YDClusterPin *pin = (YDClusterPin *)annotation;
    MKAnnotationView *annView;
    if( [pin nodeCount] > 0 ){
        annView = (YDClusterAnnotationView*)
        [mapView dequeueReusableCellWithIdentifier:@"YDclusterAnnotation"];
        if( !annView )
            annView = (YDClusterAnnotationView*)
            [[YDClusterAnnotationView alloc]
             initWithAnnotation:annotation
}

```

```

        reuseIdentifier:@"YDclusterAnnotation"];
    annView.image = [UIImage imageNamed:@"cluster.png"];
    [(YDClusterAnnotationView*)annView
    setClusterAnnotationText:[NSString stringWithFormat:@"%i", [pin nodeCount]]];
    annView.canShowCallout = NO;
} else {
    annView = [mapView dequeueReusableCellWithIdentifier:@"YDpin"];
    if( !annView )
        annView = [[MKAnnotationView alloc] initWithAnnotation:annotation
                                                 reuseIdentifier:@"YDpin"];

    annView.image = [UIImage imageNamed:@"pinPurple.png"];
    annView.canShowCallout = YES;
    //re-align the offset for the callout
    annView.calloutOffset = CGPointMake(-6.0, 0.0);
}
return annView;
}

- (void)mapView:(MKMapView *)mapView
didSelectAnnotationView:(MKAnnotationView *)view
{
    if (![[view isKindOfClass:[YDClusterAnnotationView class]]])
        return;

    CLLocationCoordinate2D centerCoordinate =
    [(YDClusterPin *)view.annotation coordinate];
    MKCoordinateSpan newSpan =
    MKCoordinateSpanMake(mapView.region.span.latitudeDelta/2.0,
                         mapView.region.span.longitudeDelta/2.0);
    [mapView setRegion:MKCoordinateRegionMake(centerCoordinate, newSpan)
            animated:YES];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

Now open the `YDViewController.xib` file with Interface Builder and drag an `MKMapView` object to the view. Click the identity inspector and change the custom class to `YDClusterMapView`, because you'll be using your own `YDClusterMapView` object. Set the delegate and connect the `YDClusterMapView` object to the `IBOutlet`, or use the Assistant Editor to create the object and its property.

Now run your application and you will see that the map is focusing on Manhattan, New York, and displays several annotations with numbers in them. The number presented is based on the number of underlying annotations that have been clustered into this `YDClusterAnnotationView`. When you zoom in, you will notice that the numbers are changing, and the individual annotations are shown with a purple pin.

## SUMMARY

In this chapter, you learned some advanced techniques to implement in relation to the `MKMapView` object.

You learned how to:

- Build a custom GPS simulator that will help you test your location-based applications by simulating a certain route.
- Create and implement custom annotations to be displayed on the `MKMapView`.
- How to respond to call-outs of annotations.
- Create and implement a custom `MKMapView` class with helper classes, which cluster annotations on your custom `MKMapView` object.

In the next chapter you learn the techniques to use `UIActionSheet` objects to present choices to the user interface and accept and process the user's selection. You also learn how to present the `UIAlertView` for simple user notifications, and extend the `UIAlertView` to collect simple, pre-defined user input without the need to code a complete `UIViewController`.

# 4

## Understanding Action Views and Alerts

### **WHAT'S IN THIS CHAPTER?**

---

- How to implement a `UIAlertView`
- How to implement and present a `UIActionSheet` using the different available presentation methods
- Processing the user response of a `UIActionSheet`

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 4 download and individually named according to the names throughout the chapter.

### **ASKING FOR USER INPUT**

Sometimes your application logic requires you either to attract the user's attention with a message he can't ignore, or present the user with one or more options to choose from. Based on the choice the user has made, your application flow can continue.

Within iOS, you can use two main objects for this purpose. The `UIActionSheet`, as the name implies, is a sheet that requires the user to select an action. The `UIAlertView` notifies the user with a title and a message. The `UIAlertView` remains visible until the user presses one of the presented button(s) that will remove the `UIAlertView` from the presentation stack. Both the `UIActionSheet` and the `UIAlertView` are presented in a modal state to the user interface, blocking the user interface until the user has made a selection.

## Creating a UIActionSheet with Multiple Options

When you want the user of your application to make a choice from a series of actions that can be performed on the `UIViewController`, you can create a `UIActionSheet` with the options you want the user to choose from. Normally, you present a `UIActionSheet` if there is more than one action to choose from. Don't forget to also present a cancel action to the user so the application can return to its previous state if the user doesn't want to make a selection.

Create a new project in Xcode using the Single View Application Project template, and name it `PresentActionSheet` using the configuration shown in Figure 4-1.

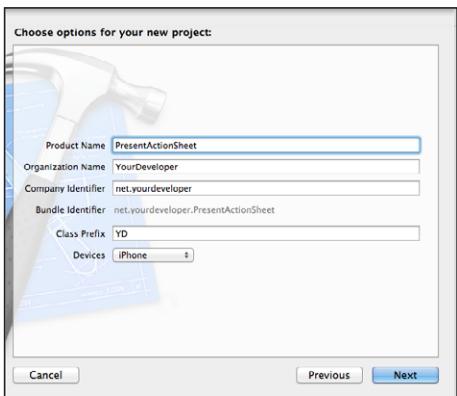


FIGURE 4-1

Open the `YDViewController.xib` file with Interface Builder and place a `UIButton` on top of the View as shown in Figure 4-2.

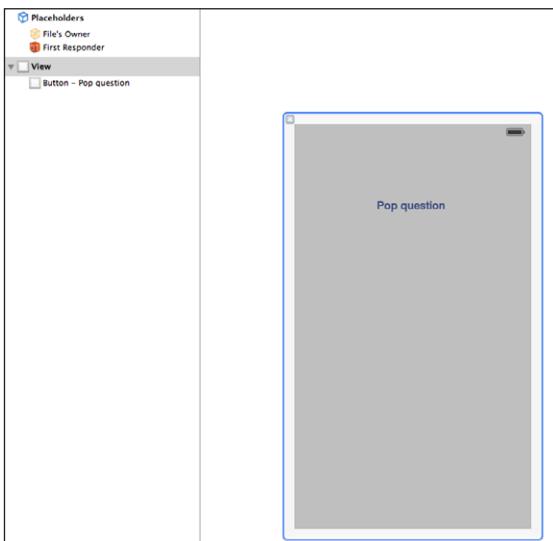
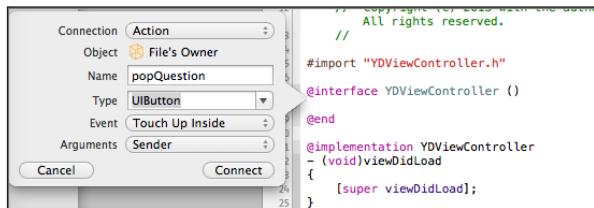


FIGURE 4-2

Using the Assistant Editor, create an `IBAction` declaration for this `UIButton` in the `YDViewController.m` named `popQuestion` as shown in Figure 4-3.



**FIGURE 4-3**

Open your `YDViewController.m` file and implement the code as shown in Listing 4-1.

LISTING 4-1: Chapter4/PresentActionSheet/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()
- (IBAction)popQuestion:(UIButton *)sender;
@end

@implementation YDViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (IBAction)popQuestion:(UIButton *)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Choose action"
        delegate:nil
        cancelButtonTitle:nil
        destructiveButtonTitle:@"OK"
        otherButtonTitles:nil];
    actionSheet.actionSheetStyle = UIActionSheetStyleDefault;
    [actionSheet showInView:self.view]; // show from your view
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
}
```

The `UIActionSheet` object is initialized in the `popQuestion:` method.

For now, don't set the delegate; you do this in the section where you learn how to respond to the user selection. You present the Action Sheet by calling the `showInView:` method of the `UIActionSheet` class.

Other methods to present the `UIActionSheet` are explained throughout the text.

Launch your application, and when you click the Pop question button the result is as shown in Figure 4-4.

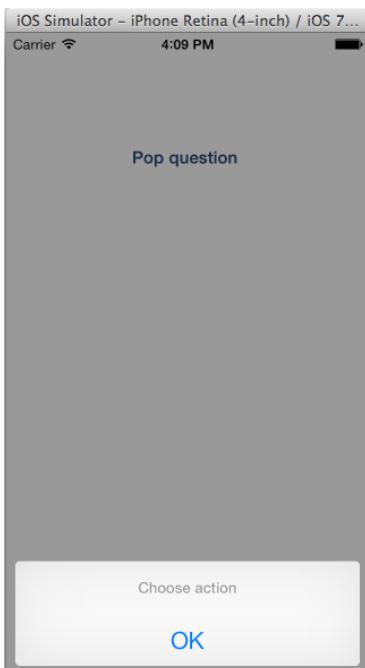


FIGURE 4-4

In this example, you created a `UIActionSheet` with a single button with the title @"OK" because that string has been assigned to the `destructiveButtonTitle`.

In a real-world application, you would use the `UIActionSheet` to ask the user to choose an action from a series of possible options. To achieve this, create a new project in which you add more buttons to the `UIActionSheet`. Add buttons by calling the `addButtonWithTitle:` method of the `UIActionSheet` class. You have to make sure that you don't create more buttons than the application window can show, of course.

Start Xcode, create a new project using the Single View Application Project template, and name it `MultipleChoice` using the options shown in Figure 4-5.

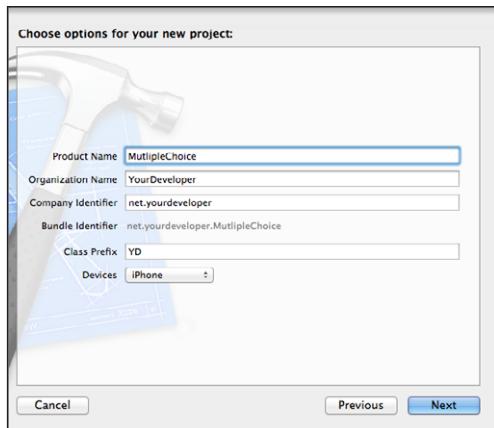


FIGURE 4-5

Like you've done in the previous application, open the `YDViewController.xib` file with Interface Builder, create a `UIButton`, and use the Assistant Editor to create the `popQuestion:` declaration.

The implementation of the `popQuestion:` method is different because you want to present multiple options to the user. For this you use the `addButtonWithTitle:` method of the `UIActionSheet` class. Because you have multiple options, you need to set the `destructiveButtonIndex` to the index of the button that you consider to be the cancel button. The index is determined based on the sequence of the `addButtonWithTitle:` methods you've called, where, as always, it's a zero-based index. The implementation is shown in Listing 4-2.

#### LISTING 4-2: Chapter4/MultipleChoice/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()

- (IBAction)popQuestion:(UIButton *)sender;

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}
- (IBAction)popQuestion:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Choose action"
        delegate:nil
```

*continues*

**LISTING 4-2 (continued)**

```
cancelButtonTitle:nil  
destructiveButtonTitle:nil  
otherButtonTitles:  
[actionSheet addButtonWithTitle:@"Take picture"];  
[actionSheet addButtonWithTitle:@"Choose picture"];  
[actionSheet addButtonWithTitle:@"Take video"];  
[actionSheet addButtonWithTitle:@"Cancel"];  
//set the destructiveButtonIndex to the button that is responsible for  
//the cancel function  
actionSheet.destructiveButtonIndex=3;  
[actionSheet showInView:self.view];  
// show from your view  
}  
- (void) didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
    // Dispose of any resources that can be recreated.  
}
```

@end

The result of this change is shown in Figure 4-6.

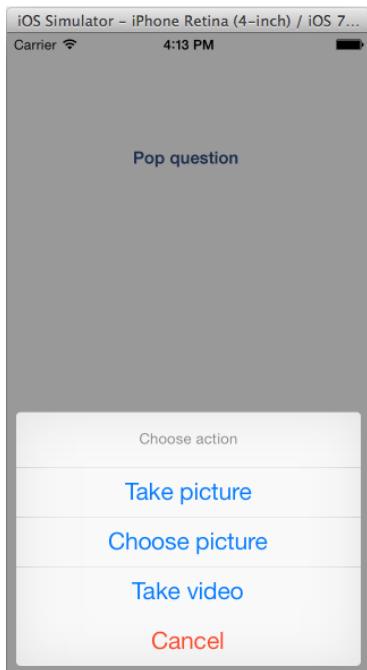


FIGURE 4-6

So far, you learned how to create a `UIActionSheet` and how to add additional buttons to it, and you used the `showInView:` method to present the `UIActionSheet`. In the following examples, you learn different methods to present your `UIActionSheet`.

## Presenting the `UIActionSheet`

You can present a `UIActionSheet` to the user in several different ways related to the type of navigation your application is supporting.

The following methods are available for presenting the `UIActionSheet`:

- `showInView`
- `showFromTabBar`
- `showFromBarButtonItem`
- `showFromRect inView`
- `showFromToolbar`

### Presenting with `showInView`

The `showInView:` method of the `UIActionSheet` class shows the `actionSheet` within the View container you assign to it, as in the previous implementation.

### Presenting with `showFromTabBar`

If your application is using a `UITabBarController` to manage its navigation stack, you can use the `showFromTabBar:` method to present your `UIActionSheet`.

Create a new project in Xcode, and choose the Tabbed Application Project template to create an application with a `UITabBarController` named `TabbedActionSheet` using the options shown in Figure 4-7. Make sure to select Universal for the devices because you need the ability to run this application on both an iPhone as well as an iPad.

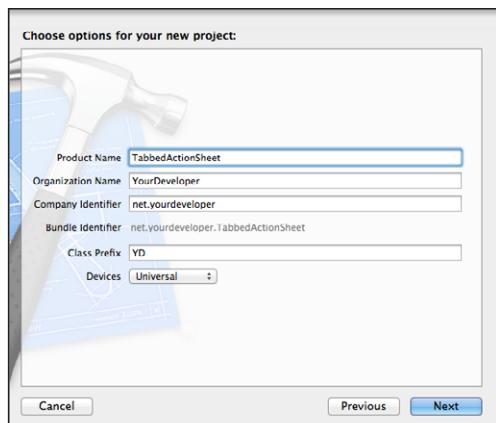


FIGURE 4-7

The project template automatically creates the `FirstViewController` and `SecondViewController` classes as well as the Interface Builder files. Use Interface Builder and the Assistant Editor to place a `UIButton` with the title Pop question on the View for both ViewControllers and create the `popQuestion:` method declaration, like you've done in the previous samples.

In Listing 4-3 you see the same logic as you've seen before to create and initialize the `UIActionSheet` and add the buttons to it by calling the `addButtonWithTitle:` method of the `UIActionSheet` class.

**LISTING 4-3: Chapter4/TabbedActionSheet/YDFFirstViewController.m**

```
#import "YDFFirstViewController.h"

@interface YDFFirstViewController ()
-(IBAction)popQuestion:(id)sender;
@end

@implementation YDFFirstViewController

- (id)initWithNibName:(NSString *)NibNameOrNil bundle:(NSBundle *)nibBundleOrNilOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNilOrNil];
    if (self) {
        self.title = NSLocalizedString(@"First", @"First");
        self.tabBarItem.image = [UIImage imageNamed:@"first"];
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
}
-(IBAction)popQuestion:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
                                  initWithTitle:@"Choose action"
                                  delegate:nil
                                  cancelButtonTitle:nil
                                  destructiveButtonTitle:nil
                                  otherButtonTitles:nil];
[actionSheet addButtonWithTitle:@"Take picture"];
[actionSheet addButtonWithTitle:@"Choose picture"];
[actionSheet addButtonWithTitle:@"Take video"];
[actionSheet addButtonWithTitle:@"Cancel"];
//set the destructiveButtonIndex to the button that is responsible
//for the cancel function
actionSheet.destructiveButtonIndex=3;
[actionSheet showFromTabBar:self.tabBarController.tabBar];

}
- (void)didReceiveMemoryWarning
{
}
```

```
{  
    [super didReceiveMemoryWarning];  
    // Dispose of any resources that can be recreated.  
}  
  
@end
```

As you may have noticed, instead of calling the `showInView:` method of the `UIActionSheet` class, you are now presenting the `UIActionSheet` by calling `showFromTabBar:` method of the `UIActionSheet` class. This line is highlighted in the preceding code. When you run the application on an iPhone or on the iPhone simulator, you will not see much difference compared to the `showInView:` method because the `UIActionSheet` will cover the width of the device's screen and be presented from the bottom. However, when you run the same application on an iPad or on the iPad simulator you still won't see any difference in the presentation of the `UIActionSheet`.

So what's the reason for using the `showFromTabBar:` method instead of the `showInView:` method?

Try pressing the pop Question button on either one of the ViewControllers and tap somewhere else on the screen. You will see the `UIActionSheet` is removed from the screen and doesn't act as a modal View demanding the user to make a selection. Also, when you would implement the `delegate` method to handle the user's selection you will see it is not responding. The reason for this is simply because the `UIActionSheet` will never become the first responder.

The code in Listing 4-3 presents the `UIActionSheet` as shown in Figure 4-8.

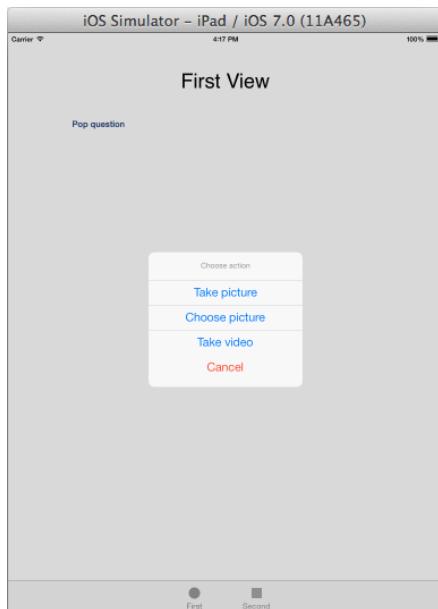


FIGURE 4-8

## Presenting with showFromBarButtonItem

If you want to present the `UIActionSheet` related to a specific `BarButtonItem`, you use the `showFromBarButtonItem:` method instead of the `showFromTabBar:` method. When developing for iPhone only, you will not notice a difference because the `UIActionSheet` is, by default, presented for the full width of the device's screen. However, when developing for iPad on which the screen's width is larger, it might be convenient to present it directly from the `BarButtonItem` you are using. To achieve this, change the `popQuestion:` method as shown in the following code snippet:

```
//get the current selectedItem and typecast it to a UIBarButtonItem
UIBarButtonItem* current =
    (UIBarButtonItem *)self.tabBarController.tabBar.selectedItem;
[actionSheet showFromBarButtonItem:current animated:YES];
```

The preceding code change presents the `UIActionSheet` as shown in Figure 4-9.



**FIGURE 4-9**

## Presenting with showFromRect

You use the `showFromRect:inView:animated:` method of the `UIActionSheet` class to present the `UIActionSheet` in a rectangular area within a View.

To try this out, start Xcode and create a new project using the Single View Application Project template, and name it `ShowFromRect` using the options shown in Figure 4-10. This time choose iPad as the device target, so you'll have a wider screen available for this tryout.

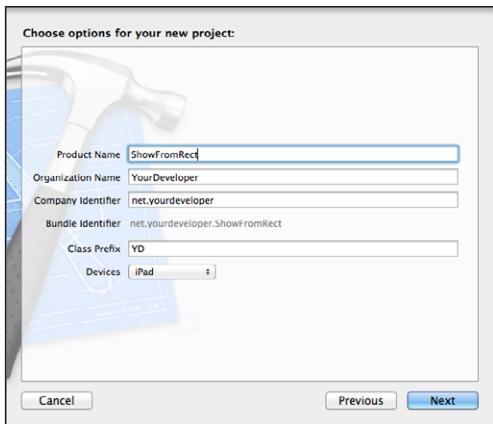


FIGURE 4-10

Use Interface Builder and the Assistant Editor to place a `UIButton` with the title `pop Question` on the View of the `YDViewController`, and create the `popQuestion:` method declaration like you've done in the previous examples. Because you want to present the `UIActionSheet` relative to the `UIButton` just created, you need to make an `IBOutlet` weak property named `popBtn` for this `UIButton` in the `YDViewController.m` file.

In the implementation of the `popQuestion:` method, in which you again create the `UIActionSheet` and add the buttons using the `addButtonWithTitle:` method of the `UIActionSheet` class, you present the `UIActionsheet` with the method `showFromRect:inView:animated:` as shown in Listing 4-4.

#### LISTING 4-4: Chapter4/ShowFromRect/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()
-(IBAction)popQuestion:(id)sender;
@property(nonatomic,weak) IBOutlet UIButton* popBtn;
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}
-(IBAction)popQuestion:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Choose action"
        delegate:self
        cancelButtonTitle:@"Cancel"
        destructiveButtonTitle:@"Delete"
        otherButtonTitles:@"OK", @"Cancel"];
    [actionSheet showFromRect:[popBtn bounds] inView:popBtn animated:YES];
}
```

*continues*

**LISTING 4-4 (continued)**

```
delegate:nil  
cancelButtonTitle:nil  
destructiveButtonTitle:nil  
otherButtonTitles:nil];  
[actionSheet addButtonWithTitle:@"Take picture"];  
[actionSheet addButtonWithTitle:@"Choose picture"];  
[actionSheet addButtonWithTitle:@"Take video"];  
[actionSheet addButtonWithTitle:@"Cancel"];  
    //set the destructiveButtonIndex to the button that is responsible  
    //for the cancel function  
actionSheet.destructiveButtonIndex=3;  
[actionSheet showFromRect:self.popBtn.frame inView:self.view animated:YES];  
}  
- (void)didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
    // Dispose of any resources that can be recreated.  
}  
  
@end
```

The `showFromRect:inView:animated:` method accepts a `CGRect` for the coordinates from where it should show, and a `UIView` in which to present the `UIActionSheet`. In this implementation you show the `UIActionSheet` directly from the frame of the `popBtn`.

The result of your implementation looks similar to Figure 4-11.

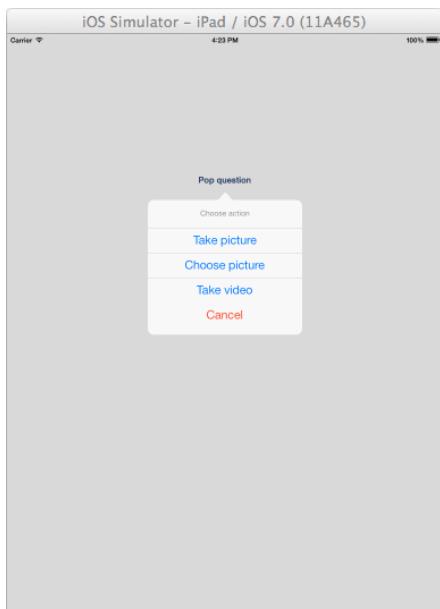
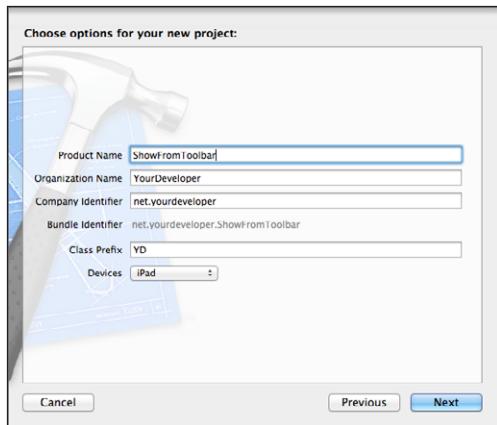


FIGURE 4-11

## Presenting with showFromToolbar

If your application is using a `UIToolbar` or a `UINavigationController` for managing your navigation stack, you can use the `showFromToolbar:` method of the `UIActionSheet` class to present the `UIActionSheet`.

Start Xcode and create a new project using the Single View Application Project template, and name it `ShowFromToolbar` using the options shown in Figure 4-12.



**FIGURE 4-12**

Open your `YDViewController.h` file and create a `UIToolbar` property as shown in Listing 4-5.

### LISTING 4-5: Chapter4/ShowFromToolbar/YDViewController.h

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

@interface YDViewController : UIViewController
@property (nonatomic, strong) UIToolbar* toolbar;
@end
```

Now open your `YDViewController.m` implementation file and create the `UIToolbar` object programmatically in the `viewDidLoad` method. For the `item1` `UIBarButtonItem`, assign the `@selector(presentActionSheet:)` to the `action` property that will call that method. In your `presentActionSheet:` method, change the presentation logic to `showFromToolbar:`. You will see the `UIActionSheet` is indeed presented from the toolbar with animation. The complete implementation is shown in Listing 4-6.

### LISTING 4-6: Chapter4/ShowFromToolbar/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()
```

*continues*

**LISTING 4-6** (*continued*)

```
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    //Create a UIToolbar
    self.toolbar =[[UIToolbar alloc] initWithFrame:CGRectMake(0, 0, 320, 44)];
    self.toolbar.translucent=YES;
    //Create UIBarButtonItems
    UIBarButtonItem *flexibleItem = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
        target:self action:nil];
    UIBarButtonItem *item1 = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAction
        target:self action:@selector(presentActionSheet:)];
    //Create the Array with items
    NSArray *items = [[NSArray alloc] initWithObjects:flexibleItem,item1, nil];
    //set the items to the toolbar
    [self.toolbar setItems:items animated:NO];
    //add the toolbar to the View
    [self.view addSubview:self.toolbar];
}
- (void)presentActionSheet:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Choose action"
        delegate:nil
        cancelButtonTitle:nil
        destructiveButtonTitle:nil
        otherButtonTitles:nil];
    [actionSheet addButtonWithTitle:@"Take picture"];
    [actionSheet addButtonWithTitle:@"Choose picture"];
    [actionSheet addButtonWithTitle:@"Take video"];
    [actionSheet addButtonWithTitle:@"Cancel"];
    //set the destructiveButtonIndex to the button that is responsible
    //for the cancel function
    actionSheet.destructiveButtonIndex=3;
    actionSheet.actionSheetStyle = UIActionSheetStyleBlackOpaque;
    [actionSheet showFromToolbar:self.toolbar];
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
@end
```

The result of your implementation looks similar to Figure 4-13.

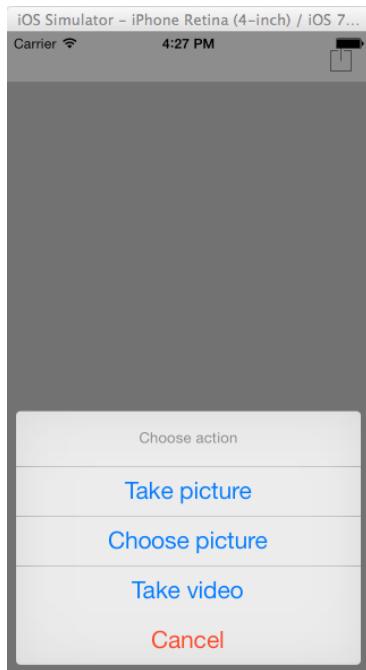


FIGURE 4-13

## RESPONDING TO USER INPUT

Up until now, you've learned how to present the `UIActionSheet` in different scenarios related to your application's navigation logic. You implemented several additional buttons, but did not yet implement anything to respond to the user selection.

### Processing the User Selection

The whole purpose of presenting a `UIActionSheet` is to allow the user to make a choice of the options that you have presented and to process the response by performing the associated application logic.

Start Xcode and create a new project using the Single View Application Project template, and name it `ActionSheetResponding` using the project options shown in Figure 4-14.

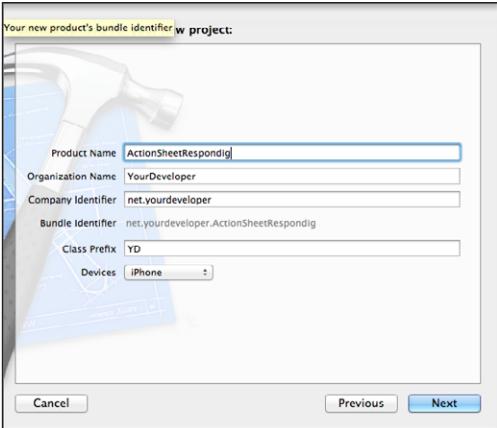


FIGURE 4-14

To be able to respond to the user selection, you need to subscribe to the `UIActionSheetDelegate` protocol in your `YDViewController` class extension and implement the delegate method `actionSheet:clickedButtonAtIndex:`:

When you create the `UIActionSheet` you now also set the delegate to `self`. This delegate is called once the user clicks one of the buttons in the `UIActionSheet`. The `actionSheet:clickedButtonAtIndex:` delegate method is called, and the index (in the same sequence as you've created the buttons starting with index 0) is used to identify what choice the user has made. Because you've captured the user's choice, you can now perform the desired action. The complete `YDViewController` implementation is shown in Listing 4-7.

#### LISTING 4-7: Chapter4/ActionSheetResponding/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()<UIActionSheetDelegate>
-(IBAction)popQuestion:(id)sender;
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}
-(IBAction)popQuestion:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Choose action"
        delegate:nil
        cancelButtonTitle:@"Cancel"
        destructiveButtonTitle:@"Delete"
        otherButtonTitles:@"Edit", @"Share", nil];
    [actionSheet showInView:self.view];
}
```

```
        cancelButtonTitle:nil
        destructiveButtonTitle:nil
        otherButtonTitles:nil];
[actionSheet addButtonWithTitle:@"Take picture"];
[actionSheet addButtonWithTitle:@"Choose picture"];
[actionSheet addButtonWithTitle:@"Take video"];
[actionSheet addButtonWithTitle:@"Cancel"];
//set the destructiveButtonIndex to the button that is
// responsible for the cancel function
actionSheet.destructiveButtonIndex=3;
actionSheet.delegate=self;
[actionSheet showInView:self.view];
// show from your view
}
- (void)actionSheet:(UIActionSheet *)actionSheet
clickedButtonAtIndex:(NSInteger)buttonIndex
{
    int index = buttonIndex;

    [actionSheet dismissWithClickedButtonIndex:buttonIndex animated:YES];
    switch (index) {
        case 0:
        {
            NSLog(@"Take picture selected");
        }
        break;
        case 1:
        {
            NSLog(@"Choose picture selected");
        }
        break;
        case 2:
        {
            NSLog(@"Take video selected");
        }
        break;
        case 3:
        {
            NSLog(@"Cancel selected");
        }
        break;
        default:
        {
            break;
        }
    }
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

The `dismissWithClickedButtonIndex:animated:` method is called to dismiss the `UIActionSheet`, and a switch on the index is used to identify what choice the user has made. In this implementation, you're just writing an `NSLog` with the result of the selected action.

## EXTENDING THE UIALERTVIEW

The `UIAlertView` object is an easy-to-use object to alert the user with a message containing a title, a message element, and one or more buttons, like the traditional OK and Cancel buttons.

With the introduction of iOS 5, you have the capability to add other objects to the `UIAlertView`.

### Adding a UITextField to a UIAlertView

In many application scenarios you simply want the user to enter a username, a password, or a code, and confirm his input by clicking an OK or Cancel button. Very often, you don't want to make a specific `UIViewController` object for this purpose. This is where the `UIAlertView` comes into play. Create a new project in Xcode using the Single View Application Project template and name it `AlertViews` using the project options shown in Figure 4-15.

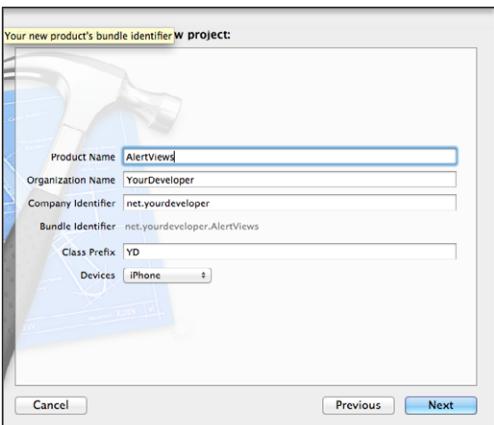


FIGURE 4-15

Use Interface Builder and the Assistant Editor to create the user interface for the `YDViewController` as shown in Figure 4-16.

Open the `YDViewController.m` file and subscribe to the `UIAlertViewDelegate` protocol. Use the Assistant Editor to create five different method declarations, one for each of the `UIButton` objects created.

In your `YDViewController.m` file you implement the various methods you've just declared as shown in Listing 4-8.

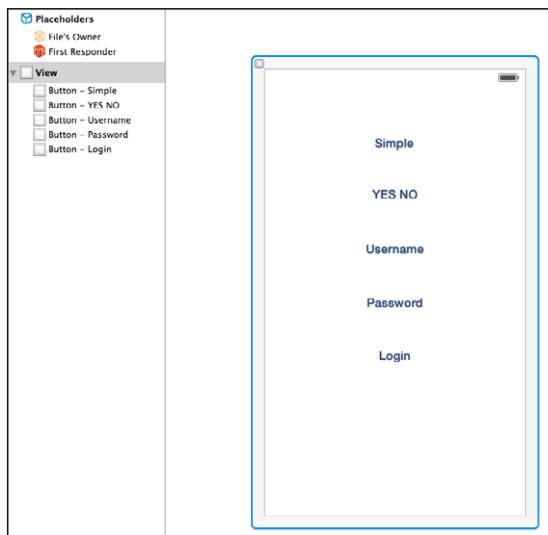


FIGURE 4-16

The `alertViewStyle` property of the `UIAlertView` class determines what kind of `UITextField` is added to the `UIAlertView`. This can be a plaintext field, a secure text field, or a combination of both. This last scenario is very useful to ask for a username and a password. The `delegate` property of the `UIAlertView` is set to `self` and in the `alertView:clickedButtonAtIndex:` delegate method you can determine which button is pressed by the user and what the input was that the user has made.

#### LISTING 4-8: Chapter4/AlertViews/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()<UIAlertViewDelegate>
-(IBAction)simpleAlert:(id)sender;
-(IBAction)simpleAlertWithCancel:(id)sender;
-(IBAction)alertWithUserName:(id)sender;
-(IBAction)alertWithPassword:(id)sender;
-(IBAction)alertWithUserNameAndPassword:(id)sender;
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}
-(IBAction)simpleAlert:(id)sender
{
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:
    @"Information" message:@"simpleAlert"
    delegate:self cancelButtonTitle:nil
    otherButtonTitles:@"Ok", nil];
}
```

*continues*

**LISTING 4-8** (*continued*)

```
[alert show];  
  
}  
-(IBAction)simpleAlertWithCancel:(id)sender  
{  
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:  
        @"Information" message:@"simpleAlertWithCancel"  
        delegate:self cancelButtonTitle:@"NO"  
        otherButtonTitles:@"YES", nil];  
    [alert show];  
}  
-(IBAction)alertWithUserName:(id)sender  
{  
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:  
        @"Information" message:@"alertWithUserName"  
        delegate:self cancelButtonTitle:nil  
        otherButtonTitles:@"Ok", nil];  
    alert.alertViewStyle = UIAlertViewStylePlainTextInput ;  
    [alert show];  
}  
-(IBAction)alertWithPassword:(id)sender  
{  
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:  
        @"Information" message:@"alertWithPassword"  
        delegate:self cancelButtonTitle:nil  
        otherButtonTitles:@"Ok", nil];  
    alert.alertViewStyle = UIAlertViewStyleSecureTextInput ;  
    [alert show];  
}  
-(IBAction)alertWithUserNameAndPassword:(id)sender  
{  
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:  
        @"Information" message:@"alertWithUserNameAndPassword"  
        delegate:self cancelButtonTitle:nil  
        otherButtonTitles:@"Ok", nil];  
    alert.alertViewStyle = UIAlertViewStyleLoginAndPasswordInput ;  
    [alert show];  
}  
#pragma mark delegate  
-(void)alertView:(UIAlertView *)alertView  
    clickedButtonAtIndex:(NSInteger)buttonIndex  
{  
    switch (buttonIndex) {  
        case 0:  
        {  
            //NO Selection or single button action  
        }  
        break;  
        case 1:  
        {//OK  
        }  
        break;  
    default:  
    }
```

```
        break;
    }
    //to access the UITextFields you can use
    if (alertView.alertViewStyle == UIAlertViewStylePlainTextInput)
    {
        UITextField *username = [alertView textFieldAtIndex:0];
    }
    else if (alertView.alertViewStyle == UIAlertViewStyleSecureTextInput)
    {
        UITextField *password = [alertView textFieldAtIndex:0];
    }
    else if (alertView.alertViewStyle == UIAlertViewStyleLoginAndPasswordInput)
    {

        UITextField *username = [alertView textFieldAtIndex:0];
        UITextField *password = [alertView textFieldAtIndex:1];
    }
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

When you launch the application it will look like Figure 4-17.

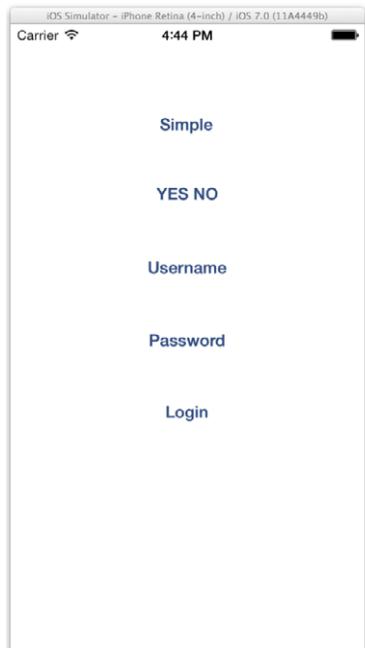


FIGURE 4-17

Figures 4-18, 4-19, and 4-20 show the result of the `UIalertView` actions that have been extended with one or more `UITextField` objects by setting the `alertViewStyle` property of the `UIalertView` class instance.

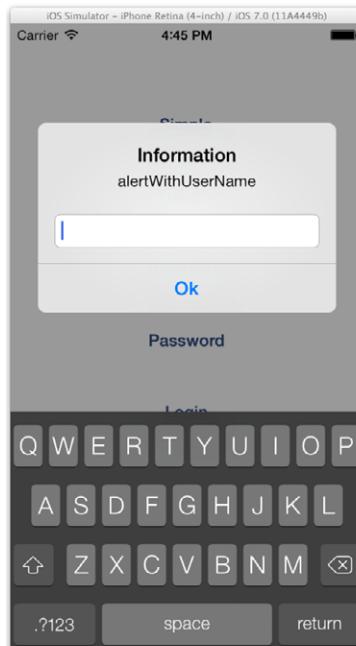


FIGURE 4-18

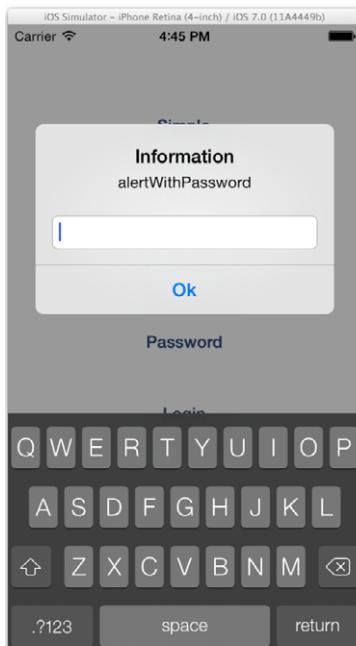


FIGURE 4-19

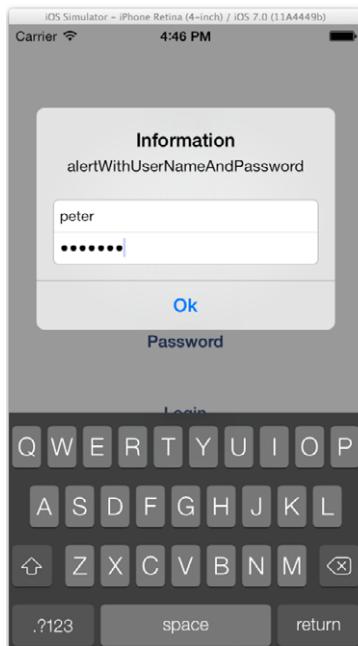


FIGURE 4-20

## SUMMARY

In this chapter you learned how to:

- Implement `UIActionSheet` objects to present the user with a series of actions from which he can choose.
- Present your `UIActionSheet` using different methods depending on your application's specific requirements and navigation structure.
- Implement a `UIalertView` with an input field to capture simple user input without the need to implement a complete `ViewController`.

In the next chapter, you learn how to internationalize your applications so they can be used by users from all over the world while respecting their local settings for date and number formats.

You also learn how to localize your application's resources, such as strings and images, to present the user interface of your application in the user's language.

# 5

# Internationalization: Building Apps for the World

## **WHAT'S IN THIS CHAPTER?**

---

- Localizing `NSString` objects
- Localizing images
- Working with calendars and date formats
- Working with number formatters

## **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 5 download and individually named according to the names throughout the chapter.

Not every iPhone user in the world speaks or reads English. To improve the chances of your application becoming a top-10 application in the App Store, you must realize that the App Store is distributing iOS applications in more than 150 countries in more than 40 different languages.

## **LOCALIZING YOUR APPLICATION**

When your application is launched on an iOS device, it determines its localization settings from the user interface, as controlled in the Settings ⇔ General ⇔ Language menu. The language set by the user will be used by your application to select localized resources.

To make it ready for the global market, it's essential that you localize your application. This means that the application must:

- Display text elements in the user's language.
- Display images related to the language of the user.
- Support data entry in any supported language.
- Work with dates in a generic way.

Of course, this applies only for the languages your application supports. If the user selects a language that is not supported by your application, your application will fall back to a default set of localized resources.

Start Xcode and create a new project using the Tabbed Application Project template, and name it InternationalApp using the options shown in Figure 5-1.

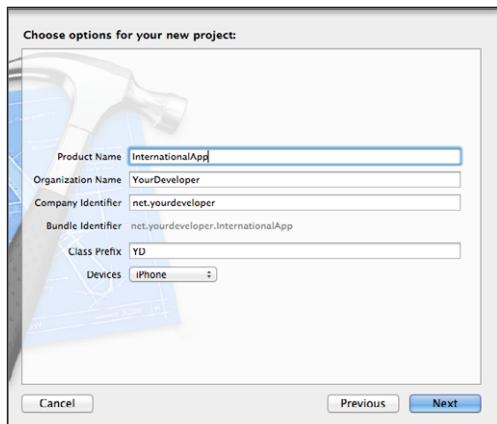


FIGURE 5-1

By default, the project template creates the following files for your application:

- YAppDelegate.h and YAppDelegate.m
- YDFirstViewController.h
- YDFirstViewController.m
- YDSecondViewController.h
- YDSecondViewController.m
- The first and second images shown on the UITabbar

When you launch the application without any modifications it will look like Figure 5-2.

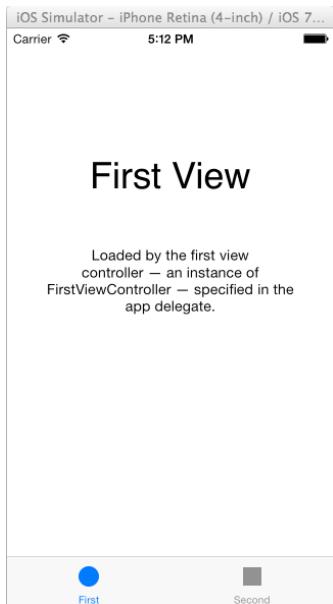


FIGURE 5-2

## Setting Up Localization

If you select the Project  $\leftrightarrow$  Info pane in Xcode, you see the available localizations for this application; by default, only English is shown here.

To add support for multiple languages, click the + button and select and add the languages your application will also support. For this exercise, add support for the French language.

After selecting the language, a pop-up screen displays the objects that are being localized, as shown in Figure 5-3.

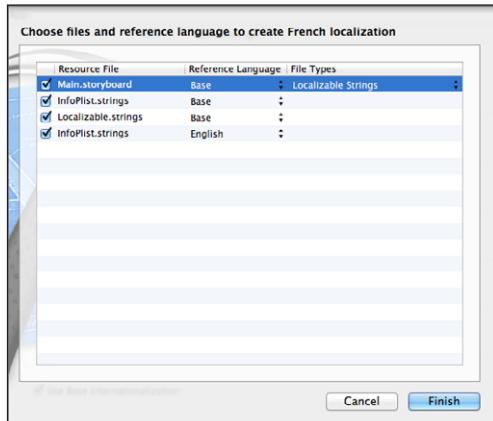


FIGURE 5-3

If you now expand the Supporting Files section in your project navigator, you'll see that the `InfoPlist.strings` file contains subnodes, and if you open it, you'll find two individual files: one for English and one for French.

With the introduction of Xcode 5 and iOS 7 Interface Builder files are not automatically localized anymore. The Storyboard however will be localized as you can see in Figure 5-4.

## Localizing Interface Builder Files

I'm not a fan of using Interface Builder files at all. Normally, I delete the created Interface Builder files from the project and create the complete UI logic in code. If your application supports multiple languages, you will have to edit each localized `.xib` file individually. This can be quite time consuming.

To create a localized version of your Interface Builder file, select the `xib` file and use the Localize button in the property pane on the right hand side. A localized version of the `xib` file will be created and you can modify it—for example, the text of a `UILabel` object as shown in Figure 5-5.

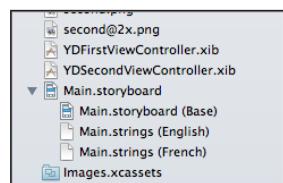


FIGURE 5-4

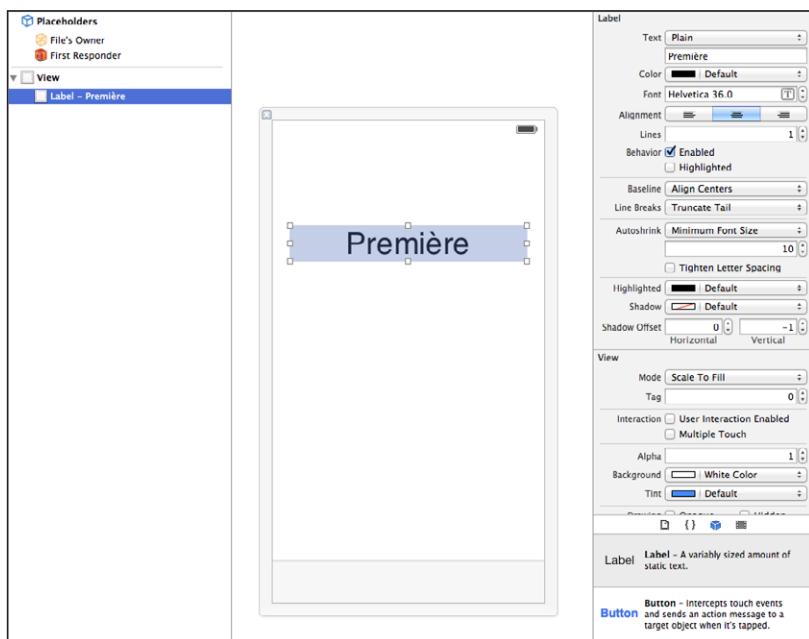


FIGURE 5-5

If you now change the `Settings → General → International → Language` in the simulator from English to French and launch the application again, you'll see the `YDFirstViewController` with the localized text you created earlier, as shown in Figure 5-6.



FIGURE 5-6

## Localizing Strings

As you can see in Figure 5-6, the label in the bottom-left Tabbar item still mentions “First,” although your settings are set to French. The reason for this is simple—you haven’t localized it yet.

If you take a look at your `YDFFirstViewController.m` file, you’ll see that in the `initWithNibName:bundle:` method the title is set with this call:

```
self.title = NSLocalizedString(@"First", @"First");
```

`NSLocalizedString` is a Foundation function that accepts two parameters and attempts to obtain a localized version of a string from a file. The first parameter is the key that will be looked up in the localized strings file, and the second parameter is the default value that will be shown if no localization is found. You can find more information on Foundation functions online in the Foundation functions reference at [https://developer.apple.com/library/mac/#documentation/cocoa/reference/foundation/miscellaneous/foundation\\_functions/reference/reference.html](https://developer.apple.com/library/mac/#documentation/cocoa/reference/foundation/miscellaneous/foundation_functions/reference/reference.html).

To create a localized string file, add a new file to the Supporting Files group by selecting `File` → `New` → `File` from the context menu.

In the iOS Resource category, select the Strings File template, and click the Next button as shown in Figure 5-7.

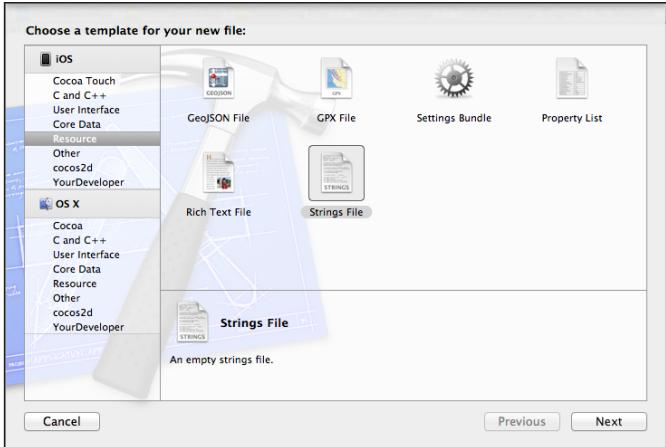


FIGURE 5-7

Save the file as `Localizable.strings`.

In the project navigator, select the `Localizable.strings` file, and on the File Inspector click the Localize button to create a localized version as shown in Figure 5-8.

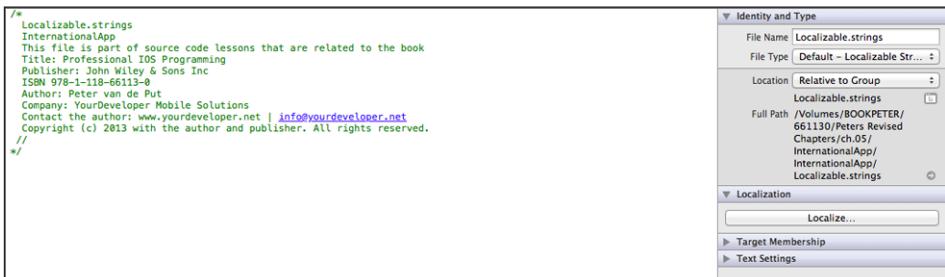


FIGURE 5-8

When you click the Localize button a pop-up window prompts you to select a default language as shown in Figure 5-9. Accept the default language and click Localize.

In the File Inspector, under Localization, you'll find two entries: English, which is checked, and French, which is unchecked. Check the French box to create a localized version for the French language.

The Project Navigator has now been updated and the individual localized versions of `Localizable.strings` can now be accessed by expanding the

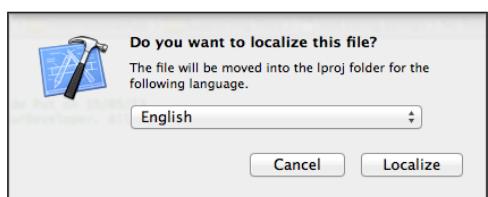


FIGURE 5-9

Localizable.string node. When you expand the Localizable.string node it will show a localized version of the file for each of the created localizations, as shown in Figure 5-10.

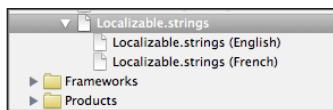


FIGURE 5-10

In these localized files, you manage your string localization by entering key-value pairs in the format shown in Listing 5-1. Take over the code from Listing 5-1 in your project.

#### LISTING 5-1: Chapter5/InternationalApp/Localizable.strings (English)

```
"First" = "First";  
"Second" = "Second";
```

The French localization is managed in Localizable.strings (French), as shown in Listing 5-2. Take over the code from Listing 5-2 in your project.

#### LISTING 5-2: Chapter5/InternationalApp/Localizable.strings (French)

```
"First" = "Première";  
"Second" = "Seconde";
```

Launch the application again (make sure the language is still set to French), and the result is as shown in Figure 5-11.

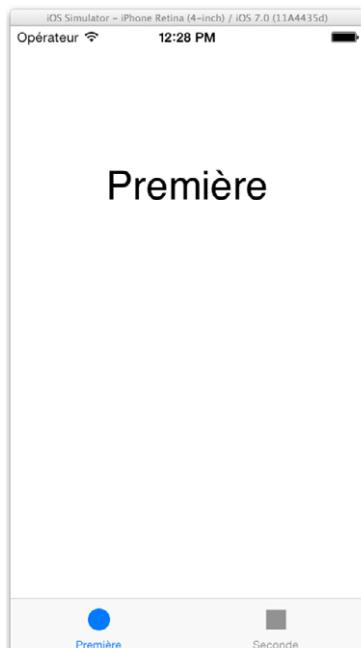


FIGURE 5-11

## Localizing Images

When you present an image on a `UIImageView` you can use the `imageNamed:` method of the `UIImage` class in case the image file is part of your bundle. Because you are passing an `NSString` as a parameter when you call this method, this can, of course, also be a localized string. To demonstrate this, add two images to your project. If you have downloaded the source code from the Wrox website, you'll see two images: one is called `usaflag.png` and the other is called `frenchflag.png`. Import these two images into your project, or pick any other image you want. Open the `YDSecondViewController.xib` file with Interface Builder and place a `UIImageView` on the View as shown in Figure 5-12.

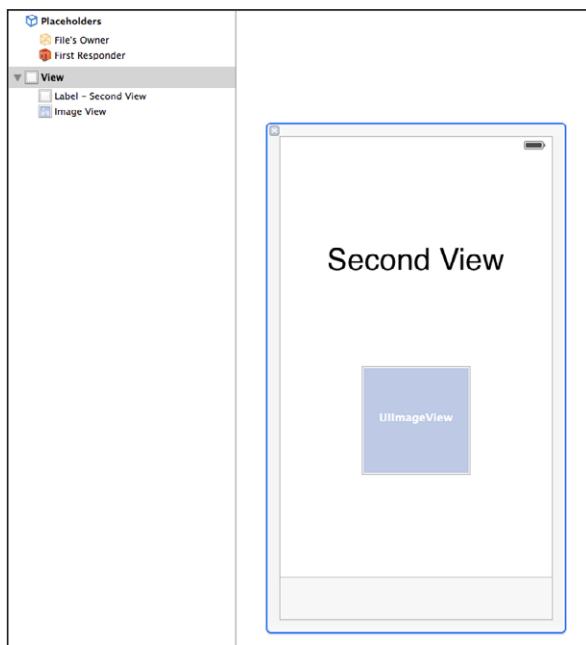


FIGURE 5-12

Use the Assistant Editor to create a weak property named `flag`. Listing 5-3 shows the `YDSecondViewController` header file.

### LISTING 5-3: Chapter5/InternationalApp/YDSecondViewController.h

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

@interface YDSecondViewController : UIViewController
@property(nonatomic,weak) IBOutlet UIImageView* flag;
@end
```

Open the `YDSecondViewController.m` file and implement the code as shown in Listing 5-4. In the `viewDidLoad` method the following code line is responsible for displaying the correct localized image:

```
self.flag.image=[UIImage imageNamed:[NSLocalizedString(@"FLAG", @"usaflag.png")]];
```

The string passed to the `imageNamed:` method is simply using the  `NSLocalizedString` function to pass a localized version of the FLAG string. What you are doing here is passing the result of the  `NSLocalizedString` function with the FLAG key to the  `imageNamed:` method of the `UIImage` class.

#### LISTING 5-4: Chapter5/InternationalApp/YDSecondViewController.m

```
#import "YDSecondViewController.h"

@interface YDSecondViewController : UIViewController

@end

@implementation YDSecondViewController
@synthesize flag=_flag;
- (id)initWithNibName:(NSString *)NibName bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:NibName bundle:bundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"Second", @"Second");
        self.tabBarItem.image = [UIImage imageNamed:@"second"];
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.flag.image=[UIImage
                    imageNamed:[NSLocalizedString(@"FLAG",
                                              @"usaflag.png")]];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

To make this work, it's essential that you modify the `Localizable.strings` file and add the FLAG key with the appropriate value, as shown in Listings 5-5 and 5-6.

**LISTING 5-5:** Chapter5/InternationalApp/Localizable.strings (English)

```
"First" = "First";
"Second" = "Second";
"FLAG" = "usaflag.png";
```

**LISTING 5-6:** Localizable.strings (French)

```
"First" = "Première";
"Second" = "Seconde";
"FLAG" = "frenchflag.png";
```

The result when you launch the application and select the Seconde tab in the Tab bar is shown in Figures 5-13 (French) and 5-14 (English).



FIGURE 5-13

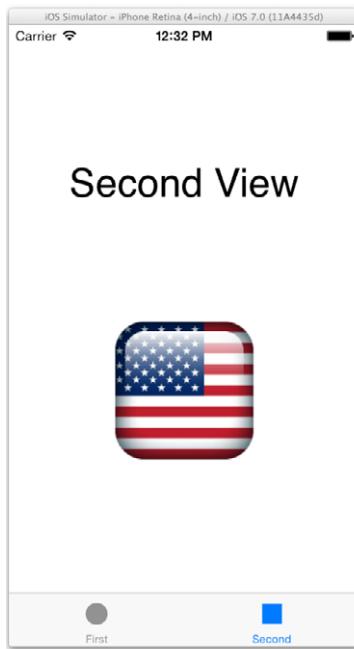


FIGURE 5-14

## Localize the Name of Your Application

To localize the name of your application, as it's shown under the application icon on the device, you can simply add the localized name of your application in the `InfoPlist.strings` file.

The key that is used to retrieve the application name is `CFBundleDisplayName`, so you simply add that key with the desired value to the localized `InfoPlist.strings` file as shown in Listings 5-7 and 5-8.

**LISTING 5-7: Chapter5/InternationalApp/InfoPlist.strings(English)**

```
CFBundleDisplayName = "English";
```

**LISTING 5-8: InfoPlist.strings(French)**

```
CFBundleDisplayName = "Francais";
```

As a result of this localization, the application displays itself on the Home Screen shown in Figures 5-15 and 5-16.



FIGURE 5-15



FIGURE 5-16

## WORKING WITH DATE FORMATS

Dates are presented in many different formats. In the United States, the standard format used is Month-Day-Year; in most European countries the standard format is Day-Month-Year.

When accepting data entry for dates or when presenting dates, it is important to implement full support in your application for displaying and accepting dates in the format the user is expecting.

If your application accepts a date only in the format month/day/year, such as 7/24/2013 (July 24, 2013), your application will crash if a European user enters the date in the format 24/7/2013. When you convert the input to an `NSDate` object, you get an exception because you consider 24 as the month instead of as the day of the month.

## What Is a Locale?

A *locale* is a set of conventions defined for handling both languages and units (such as currency, date format, time format, and the decimal separator). In Objective-C, a locale is defined as an `NSLocale` object and it consists of a locale identifier, which is a label to cluster a series of conventions. The most used and best known locale identifier is `en`, representing the English locale. Subsets are available, like `en_US` (for English United States), `en_AU` (for English Australian), and so on. Users can set the locale on their device by selecting `Settings → General → International → Region Format`. Understanding this means it's also obvious that the locale setting has nothing to do with the language setting of the user device.

You can find the complete `NSLocale` class reference doc at [https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSLocale\\_Class/Reference/Reference.html](https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSLocale_Class/Reference/Reference.html).

Start Xcode, create a new project using the Single View Application Project template, and name it InternationalDate using the options shown in Figure 5-17.

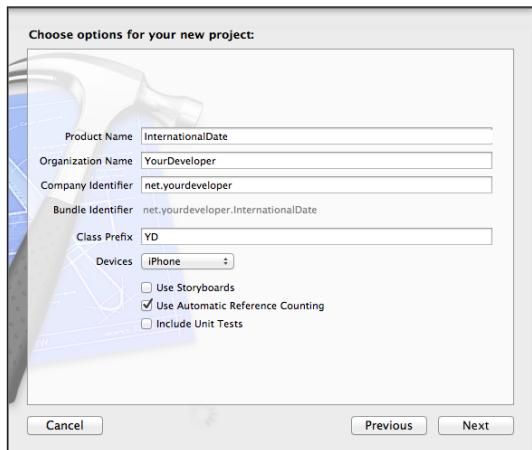


FIGURE 5-17

Open the `YDViewController.m` file, and in the `viewDidLoad` method create a `UILabel` object that displays the current locale, as shown in Listing 5-9. For this example I'm going to create the user interface programmatically, but you are welcome to use Interface Builder if you wish.

#### LISTING 5-9: Chapter5/InternationalDate/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController : UIViewController

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    UILabel* currentLbl = [[UILabel alloc]
                           initWithFrame:CGRectMake(10,10,300,20)];
    currentLbl.textColor=[UIColor blackColor];
    currentLbl.backgroundColor=[UIColor clearColor];
    [self.view addSubview:currentLbl];
    //Create an instance variable from the currentLocale
    NSLocale *currentLocale = [NSLocale currentLocale];
```

```
currentlbl.text = [NSString stringWithFormat:  
    @"Current localeIdentifier: %@",  
    currentLocale.localeIdentifier];  
  
}  
  
- (void)didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
    // Dispose of any resources that can be recreated.  
}  
  
@end
```

When you launch the application, it displays the current locale setting as shown in Figure 5-18.



FIGURE 5-18

The `NSLocale` object is important when working with `NSDate` objects. When you are working with dates, the two methods you'll use most of the time are `dateFromString:` and `stringFromDate:`. When converting an `NSDate` to an `NSString`, it's important to respect the user's `NSLocale` to ensure the user will understand and interpret the presented date as you want him to.

Open your `YDViewController.m` file and make changes to implement a second `UILabel`, to display the same date according to the specifications of the French `NSLocale`. Do this by creating an `NSLocale` object that is initialized with a `localeIdentifier` and applied to the `NSDateFormatter`, as shown in Listing 5-10.

**LISTING 5-10:** Modified Chapter5/InternationalDate/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController : UIViewController

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor whiteColor];

    UILabel* currentLbl = [[UILabel alloc] initWithFrame:CGRectMake(10,10,300,20)];
    currentLbl.textColor=[UIColor blackColor];
    currentLbl.backgroundColor=[UIColor clearColor];
    [self.view addSubview:currentLbl];
    //Create an instance variable from the currentLocale
    NSLocale *currentLocale = [NSLocale currentLocale];
    currentLbl.text = [NSString stringWithFormat:
                       @"Current localeIdentifier: %@", currentLocale.localeIdentifier];

    //create a UILabel to display a date for NSLocale current
    UILabel* dateLabel = [[UILabel alloc]
                          initWithFrame:CGRectMake(10,40,300,20)];
    dateLabel.textColor=[UIColor blackColor];
    dateLabel.backgroundColor=[UIColor clearColor];
    [self.view addSubview:dateLabel];

    //create an instance variable and initialize it with a date
    NSDate *date = [NSDate dateWithTimeIntervalSinceReferenceDate:162000];
    //create a NSDateFormatter
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    //set the current locale
    [formatter setLocale:currentLocale];
    [formatter setDateStyle:NSDateFormatterMediumStyle];
    [formatter setTimeStyle:NSDateFormatterNoStyle];
    dateLabel.text=[NSString stringWithFormat:
                   @"Date for locale: %@", [formatter stringFromDate:date]];

    //create a UILabel to display a date for NSLocale current

    UILabel* frenchLabel = [[UILabel alloc]
                           initWithFrame:CGRectMake(10,70,300,20)];
    frenchLabel.textColor=[UIColor blackColor];
    frenchLabel.backgroundColor=[UIColor clearColor];
    [self.view addSubview:frenchLabel];
    //create an NSLocale variable and initialize it with a LocaleIdentifier
    NSLocale *frenchLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"fr"];
```

```
[formatter setLocale:frenchLocale];
frenchLabel.text=[NSString stringWithFormat:
                  @"Date for French: %@", 
                  [formatter stringFromDate:date]];

}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

The result of your modification is shown in Figure 5-19.



FIGURE 5-19

## Understanding Calendars

A *calendar* encapsulates information required to calculate dates, such as the beginning, length, and division of years. In iOS, an `NSCalendar` object represents a calendar. In the international settings of the device, a user can select the calendar he wants to use. By default, the Gregorian calendar is used, but other calendars are available depending on the configuration of the device.

You can find the `NSCalendar` class reference doc at [https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSCalendar\\_Class/Reference/NSCalendar.html](https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSCalendar_Class/Reference/NSCalendar.html).

In your simulator go to Settings → General → International → Calendar, change the value from Gregorian to Buddhist, and launch your application again. You will see the date for locale has a completely different value because it's presented against the user's calendar you just changed. The result is shown in Figure 5-20.



FIGURE 5-20

You can define a `defaultCalendar` in your application's logic. For example, you can apply the `GregorianCalendar` to the `NSDateFormatter` by calling the `setCalendar:` method of the `NSDateFormatter` class, as shown in Listing 5-11.

#### LISTING 5-11: Chapter5/InternationalDate/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController : UIViewController

@end

@implementation YDViewController
```

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor whiteColor];

    UILabel* currentlbl = [[UILabel alloc]
        initWithFrame:CGRectMake(10,10,300,20)];
    currentlbl.textColor=[UIColor blackColor];
    currentlbl.backgroundColor=[UIColor clearColor];
    [self.view addSubview:currentlbl];
    //Create an instance variable from the currentLocale
    NSLocale *currentLocale = [NSLocale currentLocale];
    currentlbl.text = [NSString stringWithFormat:
        @"Current localeIdentifier: %@",
        currentLocale.localeIdentifier];

    //create a UILabel to display a date for NSLocale current
    UILabel* dateLabel = [[UILabel alloc]
        initWithFrame:CGRectMake(10,40,300,20)];
    dateLabel.textColor=[UIColor blackColor];
    dateLabel.backgroundColor=[UIColor clearColor];
    [self.view addSubview:dateLabel];

    //Create a default Calendar
    NSCalendar *defaultCalendar = [[NSCalendar alloc]
        initWithCalendarIdentifier:NSGregorianCalendar];

    //create an instance variable and initialize it with a date
    NSDate *date = [NSDate dateWithTimeIntervalSinceReferenceDate:162000];
    //create a NSDateFormatter
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    //set the defaultCalendar to the formatter
    [formatter setCalendar:defaultCalendar];
    //set the current locale
    [formatter setLocale:currentLocale];
    [formatter setDateStyle:NSDateFormatterMediumStyle];
    [formatter setTimeStyle:NSDateFormatterNoStyle];
    dateLabel.text=[NSString stringWithFormat:
        @"Date for locale: %@",
        [formatter stringFromDate:date]];

    //create a UILabel to display a date for NSLocale current

    UILabel* frenchLabel = [[UILabel alloc]
        initWithFrame:CGRectMake(10,70,300,20)];
    frenchLabel.textColor=[UIColor blackColor];
    frenchLabel.backgroundColor=[UIColor clearColor];
    [self.view addSubview:frenchLabel];
    //create an NSLocale variable and initialize it with a LocaleIdentifier
    NSLocale *frenchLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"fr"];
    [formatter setLocale:frenchLocale];
    frenchLabel.text=[NSString stringWithFormat:
        @"Date for French: %@",
        [formatter stringFromDate:date]]];

```

*continues*

**LISTING 5-11 (continued)**

```
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

As you can see in Figure 5-21, the locale date is displayed as Jan 2, 2001 despite the user's Calendar setting.



**FIGURE 5-21**

The following calendar identifiers are available:

- NSGregorianCalendar
- NSBuddhistCalendar
- NSChineseCalendar
- NSHebrewCalendar

- NSIslamicCalendar
- NSIslamicCivilCalendar
- NSJapaneseCalendar

## Storing Dates in a Generic Way

If your application requires dates to be stored, you know that a string presentation of a date is ambiguous and, once stored, can no longer be converted to the intended date in all cases because the user might have changed his device settings.

A universal way to solve this potential problem is by storing dates in a generic way. The generic solution, sometimes called a unixDateTimeStamp, is simply calling a method on an `NSDate` object called `timeIntervalSince1970`. This method calculates the interval between January 1, 1970 and the date provided and returns a double.

If you'll be working a lot with dates, you can extend your application framework by creating a static `DateHelper` class and adding it to the Personal Library you created in Chapter 1.

Create a new class called `DateHelper` that subclasses `NSObject` and create two static methods, as shown in Listing 5-12.

**LISTING 5-12:** Chapter5/InternationalDate/DateHelper.h

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

@interface DateHelper : NSObject
+(int)getGenericDateForDate:(NSDate *)date;
+(NSDate *)getDateforGenericDate:(int)genericDate;
@end
```

Now open the `DateHelper.m` implementation file and implement the methods as shown in Listing 5-13.

**LISTING 5-13:** Chapter5/InternationalDate/DateHelper.m

```
#import "DateHelper.h"

@implementation DateHelper
+(int)getGenericDateForDate:(NSDate *)date
{
    return (int)[date timeIntervalSince1970];
}
+(NSDate *)getDateforGenericDate:(int)genericDate
{
    return [NSDate dateWithTimeIntervalSinceReferenceDate:genericDate];
}
@end
```

## WORKING WITH NUMBERS

As you learned in the previous section, the locale settings of a user's device impact the presentation of dates. It also impacts the way numbers are displayed.

### Introducing Number Formatters

When working with Numeric objects for which you want to implement a textual representation of the value, you use an `NSNumberFormatter` to convert the numeric value to a textual representation.

The `NSNumberFormatter` class reference doc is available at [https://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSNumberFormatter\\_Class/Reference.html](https://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSNumberFormatter_Class/Reference.html).

Start Xcode and create a new project using the Single View Application Project template, and name it `NumberFormats` using the options shown in Figure 5-22.

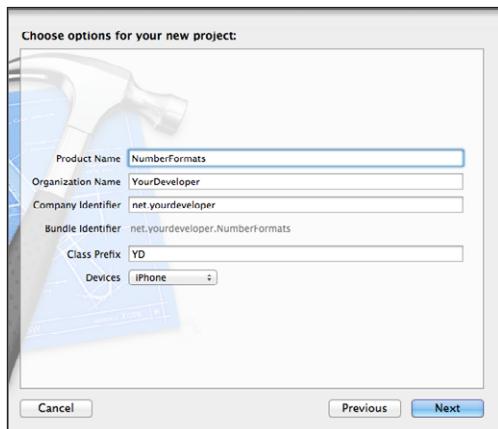


FIGURE 5-22

You'll be using this application to learn how to implement an `NSNumberFormatter` object. In this application, you show some number formatter options for English and French `NSLocale` settings, so the best way to present this is using a grouped `UITableView`.

Open your `YDViewController.h` file and create a strong property of type `UITableView` named `mTableView` as shown in Listing 5-14.

LISTING 5-14: Chapter5/NumberFormats/YDViewController.h

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

@interface YDViewController : UIViewController

@property(nonatomic,strong) UITableView* mTableView;
@end
```

Now open your `YDViewController.m` file, and subscribe to the `UITableViewDelegate` and `UITableViewDataSource` protocol. Create the `UITableView` object, and set the properties in the `viewDidLoad` method. Of course, you can use Interface Builder and the Assistant Editor if you prefer to do so.

For this example, hard-code two sections to be displayed with four rows in each section. This example uses a `UITableView` to present the different `NSNumberFormatter` options.

Create an `NSNumberFormatter` object in `tableView:cellForRowAtIndexPath:` and, using a `switch` operator on the `indexPath.row`, display different formatter options. The complete implementation is shown in Listing 5-15.

**LISTING 5-15: Chapter5/NumberFormats/YDViewController.m**

```
#import "YDViewController.h"

@interface YDViewController ()<UITableViewDataSource, UITableViewDelegate>

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    self.view.backgroundColor=[UIColor whiteColor];
    self.mTableView=[[UITableView alloc]
                    initWithFrame:CGRectMake(0, 0, 320, 460)
                    style:UITableViewStyleGrouped];

    self.mTableView.delegate=self;
    self.mTableView.dataSource=self;
    [self.view addSubview:self.mTableView];
}

#pragma mark UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 2;
}
- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    switch (section) {
        case 0:
            return @"English local for 1234.99";
            break;
        case 1:
            return @"French local for 1234.99";
            break;
        default:
            break;
    }
}

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

    cell.textLabel.text=[self.formatters[indexPath.section] stringFromNumber:@(1234.99)];
    return cell;
}
```

*continues*

**LISTING 5-15 (continued)**

```
        break;
    }
}

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

    return 4;
}
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = (UITableViewCell *)[tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    cell.selectionStyle = UITableViewCellSelectionStyleNone;
    NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];

    switch (indexPath.row) {
        case 0:
        {
            if (indexPath.section==0){
                [numberFormatter setLocale:[[NSLocale alloc]
                    initWithLocaleIdentifier:@"en"]];
            }
            else
                [numberFormatter setLocale:[[NSLocale alloc]
                    initWithLocaleIdentifier:@"fr"]];
            [numberFormatter setNumberStyle:NSNumberFormatterDecimalStyle];
            cell.textLabel.text = [NSString
                stringWithFormat:@
                @"DecimalStyle: %@",[numberFormatter stringFromNumber:@1234.99]];
        }
        break;
    case 1:
    { if (indexPath.section==0){
        [numberFormatter setLocale:[[NSLocale alloc]
            initWithLocaleIdentifier:@"en"]];
    }
    else
        [numberFormatter setLocale:[[NSLocale alloc]
            initWithLocaleIdentifier:@"fr"]];
        [numberFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
        cell.textLabel.text = [NSString
            stringWithFormat:@
            @"CurrencyStyle: %@",[numberFormatter stringFromNumber:@1234.99]];
    }
}
```

```
        @"CurrencyStyle: %@",  
        [numberFormatter stringFromNumber:@1234.99]];  
    }  
    break;  
case 2:  
{ if (indexPath.section==0){  
    [numberFormatter setLocale:[[NSLocale alloc]  
    initWithLocaleIdentifier:@"en"]];  
}  
else  
    [numberFormatter setLocale:[[NSLocale alloc]  
    initWithLocaleIdentifier:@"fr"]];  
    [numberFormatter setNumberStyle:NSNumberFormatterPercentStyle];  
    cell.textLabel.text = [NSString  
        stringWithFormat:  
        @"PercentStyle: %@",  
        [numberFormatter stringFromNumber:@1234.99]];  
}  
    break;  
case 3:  
{ if (indexPath.section==0){  
    [numberFormatter setLocale:[[NSLocale alloc]  
    initWithLocaleIdentifier:@"en"]];  
}  
else  
    [numberFormatter setLocale:[[NSLocale alloc]  
    initWithLocaleIdentifier:@"fr"]];  
    [numberFormatter setPositiveFormat:@"###0.##"];  
    cell.textLabel.text = [NSString  
        stringWithFormat:  
        @"###0.##: %@",  
        [numberFormatter stringFromNumber:@1234.99]];  
}  
    break;  
default:  
    break;  
}  
  
    return cell;  
}  
  
- (void)didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
    // Dispose of any resources that can be recreated.  
}  
  
@end
```

The result is shown in Figure 5-23.

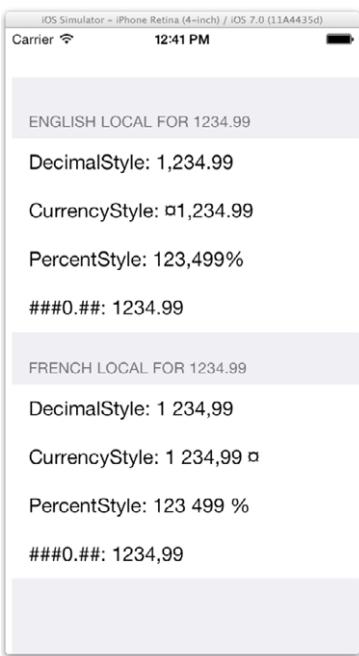


FIGURE 5-23

## SUMMARY

In this chapter you learned:

- The importance of internationalization for your application by implementing localization technologies for strings, images, and localizing the application title.
- How the user's settings impact application behavior.
- How to implement solutions to correctly present dates and numbers in relation to the user's preference.

In the next chapter you learn how to use multimedia in your applications. You learn to display PDF documents with a low memory footprint and how to create a thumbnail image from a PDF page. Then you learn how to play audio and video files from the bundle and from the iTunes library on your device.

# 6

# Using Multimedia

## WHAT'S IN THIS CHAPTER?

---

- Working with PDF documents
- Playing and recording audio
- Accessing the iTunes library

## WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 6 download and individually named according to the names throughout the chapter.

## PORTABLE DOCUMENT FORMAT

A *Portable Document Format* (PDF) file is a file that uses a file format to represent documents in a software- and hardware-independent way. PDF documents can contain text, images, audio files, hyperlinks, and bitmaps. Originally the PDF definition was created by Adobe, but since 2008 it has become an open standard that's widely used for many purposes. The Quartz framework is used to work with PDF documents and it's good to know that not all versions of PDF documents are supported, such as password-protected PDF documents.

To display a PDF document as shown in this section, download the iOS Human Interface Guidelines from the Apple website via the following link and save it on your machine so you can include this document in the examples: <http://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/MobileHIG.pdf>.

You can use this document in the different lessons in this chapter, and because of its size of 27 MB with 219 pages, it provides a serious payload for testing and analyzing the performance of your applications.

## Displaying a PDF Document with a UIWebView

Using a `UIWebView` object to display the contents of a PDF document is the most-used solution in iOS applications. Unfortunately, it's also one of the major causes of iOS applications crashing because it's very memory expensive, as you'll see after creating this project.

Start Xcode and create a new project named `WebPDFViewer` using the Single View Application Project template, using the options shown in Figure 6-1.

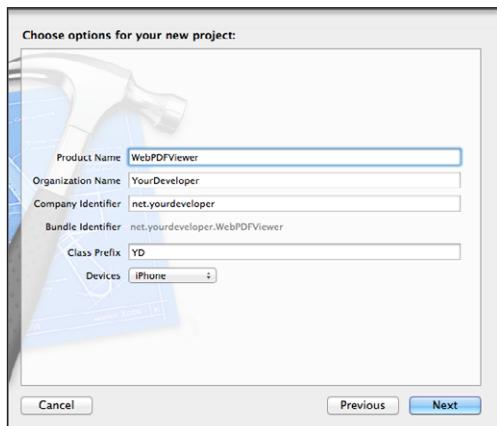


FIGURE 6-1

Use Interface Builder and the Assistant Editor to create a simple user interface with a `UIWebView` object, and create a weak property named `pdfViewer` for it, as shown in Figure 6-2.

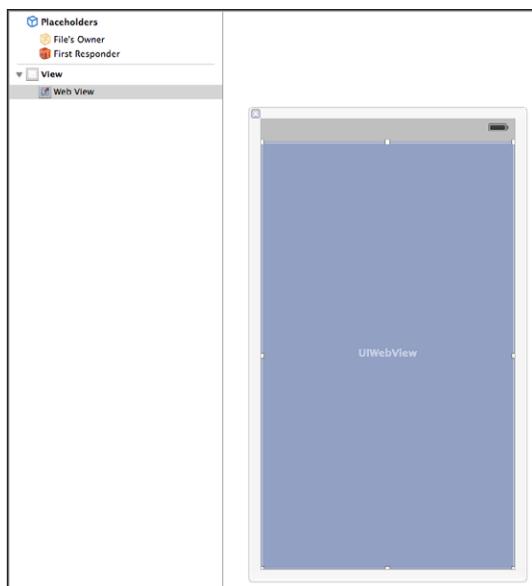


FIGURE 6-2

Include in your project the `MobileHIG.pdf` document you have already downloaded and stored in your machine.

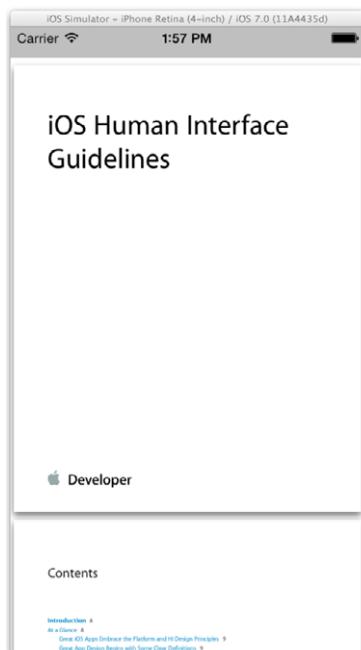
Implement the logic to display the PDF document by adding the following code to the implementation of the `viewDidLoad` method, as shown in Listing 6-1.

**LISTING 6-1: The `viewDidLoad` method**

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    //load PDF file from the Bundle
    NSString *fileToLoad = [[NSBundle mainBundle]
        pathForResource:@"MobileHIG" ofType:@"pdf"];
    NSData *data1 = [[NSData alloc] initWithContentsOfFile:fileToLoad];
    [self.pdfViewer loadData:data1
        MIMEType:@"application/pdf"
        textEncodingName:@"UTF-8" baseURL:nil];
}
```

The `viewDidLoad` implementation creates an `NSData` object with the contents of the PDF file you want to display, and calls the `loadData:MIMEType:textEncodingName:baseURL:` method of the `UIWebView` and passes the `NSData` object.

The result is shown in Figure 6-3.



**FIGURE 6-3**

## Introducing Profiling with Instruments

With Instruments, you use special tools (known as instruments) to trace different aspects of a process's behavior. You can also use the tool to record a sequence of user interface actions and replay them, using one or more instruments to gather data. To launch Instruments, select the following menu path in Xcode: Xcode  $\Rightarrow$  Open Developer Tool  $\Rightarrow$  Instruments.

Alternatively, if you know you want to profile your project you can select Product  $\Rightarrow$  Profile from the Xcode menu.

If you are new to profiling projects and using instruments, read the Instruments User Guide, which you can find at [https://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html#/apple\\_ref/doc/uid/TP40004652](https://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html#/apple_ref/doc/uid/TP40004652).

To demonstrate how much memory this application is using, let's profile the application. Select Product  $\Rightarrow$  Profile from the Xcode menu. Select the Allocations instrument from the screen as shown in Figure 6-4 and click the Profile button.

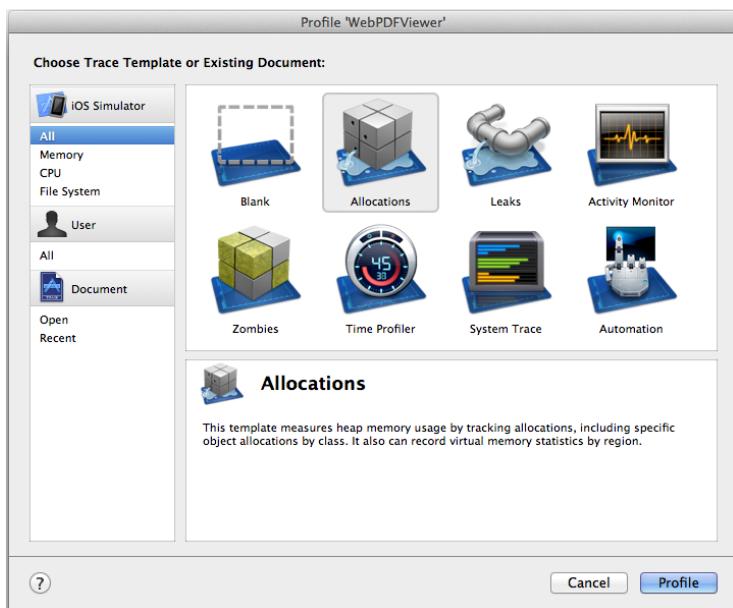


FIGURE 6-4

The application will be launched, and in the instruments window you will see statistics measured in real time for the running application. You will see that your project is using 35 MB of Overall Bytes. Your screen will look like Figure 6-5, and when you have collected all the information you need you can click the Stop button in the instruments window to stop the profiling process.

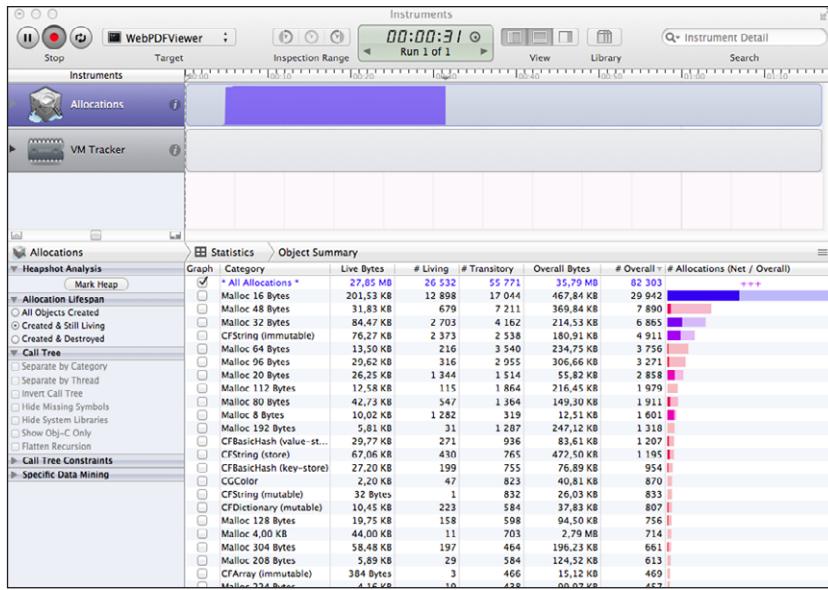


FIGURE 6-5

It will get worse when you do some scrolling in the PDF by scrolling down several pages; the memory usage increases very rapidly. This is very often the reason an application that is displaying PDF documents using a UIWebView object will crash because they run low on memory. The memory usage after some scrolling is shown in Figure 6-6.

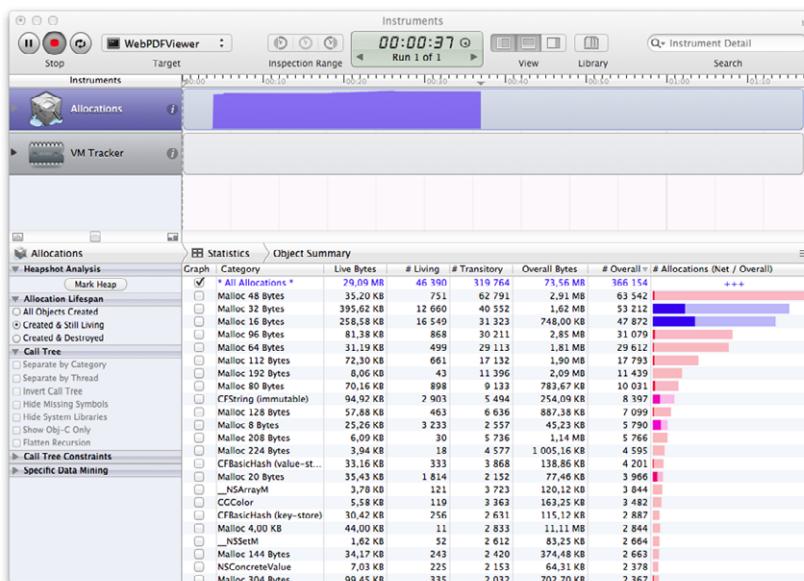


FIGURE 6-6

You can definitely conclude that it's easy to display a PDF document using a `UIWebView`, but it's also a very memory-expensive solution that might result in application crashes when the application is running low on memory. Therefore, it is not an advised solution.

## Displaying a PDF Document using QuickLook

Fortunately there is no need to use the `UIWebView` object to display documents because the QuickLook framework also supports the display of PDF documents. QuickLook supports the display of many document formats, such as Microsoft Word, Excel, and PowerPoint.

To learn more about the QuickLook framework and how to use it, read the documentation at [http://developer.apple.com/library/ios/#documentation/QuickLook/Reference/QuickLookFrameworkReference\\_iPhoneOS/\\_index.html](http://developer.apple.com/library/ios/#documentation/QuickLook/Reference/QuickLookFrameworkReference_iPhoneOS/_index.html).

Start Xcode and create a new project using the Single View Application Project template, and name it `QLPDFViewer` using the options shown in Figure 6-7.

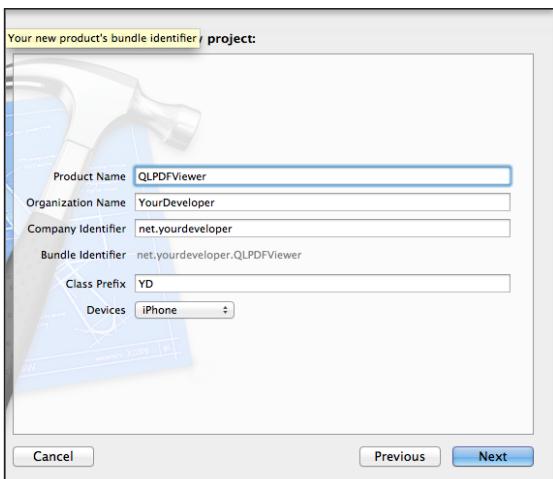


FIGURE 6-7

Add the QuickLook framework to your project and also add the `MobileHIG.pdf` document you used earlier.

Open your `YDViewController.h` file and declare a strong property named `documents`, as shown in Listing 6-2.

### LISTING 6-2: Chapter6/QLPDFViewer/YDViewController.h

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

@interface YDViewController : UIViewController

@property(nonatomic,strong) NSArray* documents;

@end
```

In the `YDViewController.m` implementation you start by importing the QuickLook header file and subscribe to the `QLPreviewControllerDataSource` and `QLPreviewControllerDelegate` protocols. In the `viewDidLoad` method you initialize the `documents` array and initialize it with your `MobileHIG.pdf` document. You initialize and configure the `previewController` instance and set the properties.

Next, you create and configure your `previewController`, set its delegate and datasource, and use a `performSelector:withObject:afterDelay:` method to call the `showDocument` method.

The `showDocument` method is presenting the `previewController`.

Because you've subscribed to the `QLPreviewControllerDataSource` and `QLPreviewControllerDelegate` protocols you must implement both these mandatory methods: `numberOfPreviewItemsInPreviewController:` and `previewController:previewItemAtIndex::`

The `numberOfPreviewItemsInPreviewController:` returns the number of items in the `self.documents` array you have created in the `viewDidLoad` method. The `previewController:previewItemAtIndex:` method is returning a `QLPreviewItem` object that is constructed by the `NSURL` to the document from the `self.documents` array you have created in the `viewDidLoad` method.

A caveat of using the QuickLook framework is that when the `QLPreviewItem` is presented, it automatically creates a button in the navigation controller that allows the user to e-mail or print the document.

The complete implementation is shown in Listing 6-3.

#### LISTING 6-3: Chapter6/QLPDFViewer/YDViewController.m

```
#import "YDViewController.h"
#import <QuickLook/QuickLook.h>
@interface YDViewController ()<QLPreviewControllerDataSource,
QLPreviewControllerDelegate,
QLPreviewItem>
{
    QLPreviewController* previewController;
}
@end

@implementation YDViewController

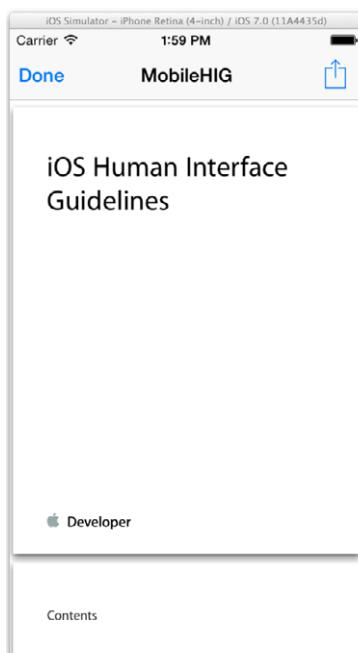
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor whiteColor];
    self.documents=[[NSArray alloc] initWithObjects:@"MobileHIG.pdf", nil];
    //initialize the QLPreviewController
    previewController = [[QLPreviewController alloc] init];
    previewController.dataSource=self;
    previewController.delegate=self;
    [self performSelector:@selector(showDocument) withObject:nil afterDelay:3];
}
- (void)showDocument
{
```

*continues*

**LISTING 6-3 (continued)**

```
[self presentViewController:previewController animated:YES completion:nil];  
  
}  
#pragma mark delegates  
- (NSInteger)numberOfPreviewItemsInPreviewController:  
    (QLPreviewController *)controller  
{  
    return [self.documents count];  
}  
-(id<QLPreviewItem>)previewController:(QLPreviewController *)controller  
previewItemAtIndex:(NSInteger)index  
{  
    return [NSURL fileURLWithPath:[[NSBundle mainBundle]  
pathForResource:[self.documents objectAtIndex:index] ofType:nil]];  
}  
- (void)didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
    // Dispose of any resources that can be recreated.  
}  
  
@end
```

When you run this application, it will look as shown in Figure 6-8.



**FIGURE 6-8**

Now let's see what the difference is in memory usage when presenting the same PDF using the QuickLook framework by profiling this project.

Select Product  $\Rightarrow$  Profile, select Allocations from the instruments view, and click the Profile button.

The application now uses less than 1 MB of memory, and when scrolling through the PDF document there is no significant change in memory usage. If you take a look at the Overall Bytes column, you will see that instead of the 27 MB your application was using with the UIWebView, now it's only using 3 MB—a dramatic, important difference for the stability and performance of your application.

The result is shown in Figure 6-9.

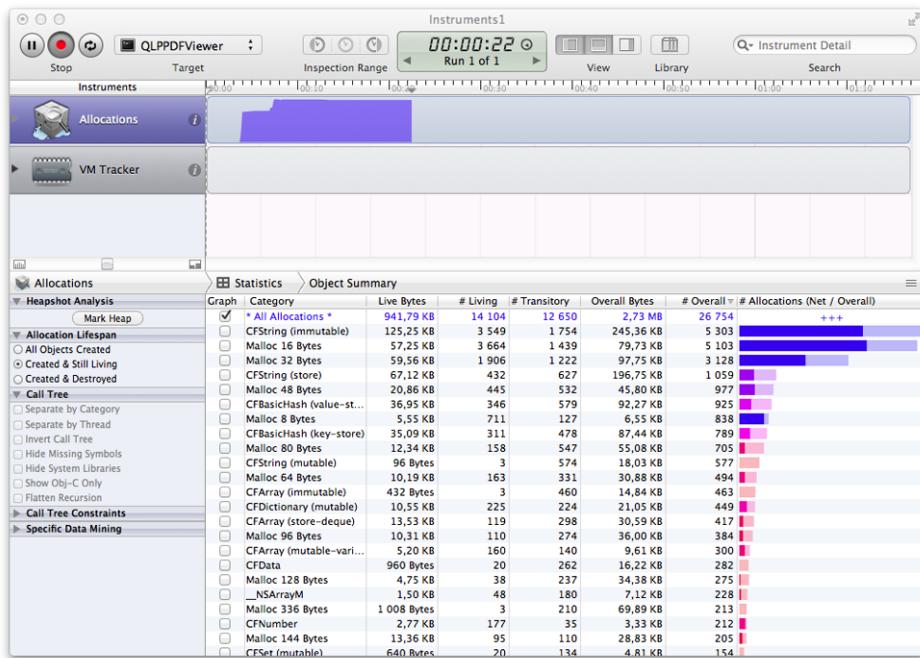


FIGURE 6-9

## Creating a Thumbnail from a PDF Document

When your application is working with multiple PDF documents and you want to present some kind of user interface for the user to select and view the PDF document, you can use a standard image, like a PDF icon to indicate the document is a PDF document, or you can use the title of the document. Another more elegant solution is to create a thumbnail from the PDF document itself so it shows a real preview of the document the user is able to select.

Start Xcode and create a new project using the Single View Application Project template, and name it PDFCreateThumbnail using the options shown in Figure 6-10.

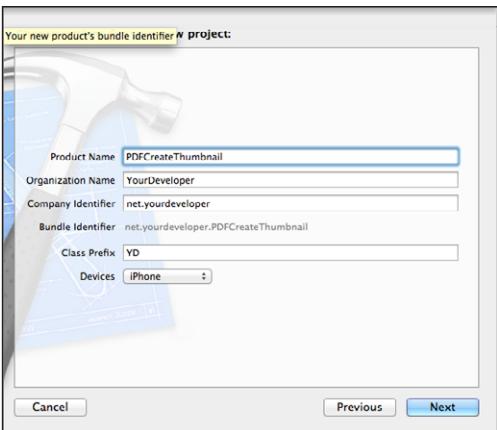


FIGURE 6-10

Import the QuickLook framework and add the `MobileHIG.pdf` document you used earlier to your project.

Now you open your `YDViewController.h` file and implement the code as shown in Listing 6-4.

#### LISTING 6-4: Chapter6/PDFCreateThumbnail/YDViewController.h

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

@interface YDViewController : UIViewController

@property(nonatomic,strong) NSArray* documents;

@end
```

In the `YDViewController.m` file, you import the QuickLook header file and subscribe to the `QLPreviewControllerDataSource`, `QLPreviewControllerDelegate`, and `QLPreviewItem` protocols.

In the `viewDidLoad` method of your `YDViewController.m` file you create your `previewController` and `documents` array like in the previous example. You now also create a `UIButton` object named `pdfButton` programmatically.

The thumbnail will be created by the method called `createThumbnailForPage:` that takes the page number as an argument and returns a `UIImage` instance. This `UIImage` instance is used as the image for the `pdfButton` you have just created.

This method is opening the context of the document and creating a `CGRect` object. It then reads the passed page number of the document and performs the scaling and transformation using `CoreGraphics` functions and returns a `UIImage` object.

The Quartz2D programming guide is a good starting point for learning and understanding how Quartz is related to PDF documents. You can find this programming guide at [http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/drawingwith-quartz2d/dq\\_pdf/dq\\_pdf.html#/apple\\_ref/doc/uid/TP30001066-CH214-TPXREF101](http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/drawingwith-quartz2d/dq_pdf/dq_pdf.html#/apple_ref/doc/uid/TP30001066-CH214-TPXREF101).

The complete implementation is shown in Listing 6-5.

**LISTING 6-5: Chapter6/PDFCreateThumbnail/YDViewController.m**

```
#import "YDViewController.h"
#import <QuickLook/QuickLook.h>
@interface YDViewController ()<QLPreviewControllerDataSource,
QLPreviewControllerDelegate,
QLPreviewItem>
{
    QLPreviewController* previewController;
}
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.view.backgroundColor=[UIColor blackColor];
    self.documents=[[NSArray alloc] initWithObjects:@"MobileHIG.pdf", nil];

    previewController = [[QLPreviewController alloc] init];
    previewController.dataSource=self;
    previewController.delegate=self;
    //Create a UIButton
    UIButton* pdfButton = [[UIButton alloc] initWithFrame:CGRectMake(100, 100, 70, 100)];
    pdfButton.backgroundColor=[UIColor clearColor];
    [pdfButton addTarget:self action:@selector(showDocument)
forControlEvents:UIControlEventTouchUpInside];
    //Set the image to thumbnail of page 4
    [pdfButton setImage:[self createThumbnailforPage:4]
forState:UIControlStateNormal];
    [self.view addSubview:pdfButton];
}

-(UIImage*)createThumbnailforPage:(int)pageno
{
    NSURL* pdfFileUrl = [NSURL fileURLWithPath:[[NSBundle mainBundle]
pathForResource:[self.documents objectAtIndex:0] ofType:nil]];

    CGPDFDocumentRef pdf =
    CGPDFDocumentCreateWithURL((__bridge CFURLRef)pdfFileUrl);

    CGRect aRect = CGRectMake(0, 0, 70, 100); // thumbnail size
    UIGraphicsBeginImageContext(aRect.size);

    UIGraphicsBeginImageContext(aRect.size);
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSaveGState(context);

```

*continues*

**LISTING 6-5 (continued)**

```
CGContextTranslateCTM(context, 0.0, aRect.size.height);
CGContextScaleCTM(context, 1.0, -1.0);
CGContextTranslateCTM(context, -(aRect.origin.x), -(aRect.origin.y));

CGContextSetGrayFillColor(context, 1.0, 1.0);
CGContextFillRect(context, aRect);

    //Grab the first PDF page
    CGPDFPageRef page = CGPDFDocumentGetPage(pdf, pageno);
    CGAffineTransform pdfTransform =
    CGPDFPageGetDrawingTransform(page, kCGPDFCropBox, aRect, 0, false);
    // And apply the transform.
    CGContextConcatCTM(context, pdfTransform);
    CGContextDrawPDFPage(context, page);

    // Create the new UIImage from the context
    UIImage *thumbnail = UIGraphicsGetImageFromCurrentImageContext();

    CGContextRestoreGState(context);
    UIGraphicsEndImageContext();
    CGPDFDocumentRelease(pdf);

    return thumbnail;
}

-(void)showDocument
{
    [self presentViewController:previewController animated:YES completion:nil];
}
#pragma mark delegates
-(NSInteger)numberOfPreviewItemsInPreviewController:(QLPreviewController *)controller
{
    return [self.documents count];
}
-(id<QLPreviewItem>)previewController:(QLPreviewController *)controller
    previewItemAtIndex:(NSInteger)index
{
    return [NSURL fileURLWithPath:[NSBundle mainBundle]
        pathForResource:[self.documents objectAtIndex:index] ofType:nil]];
}

-(void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

The result is shown in Figure 6-11.

When your application needs to create many PDF thumbnails to display in a UITableView, creating the thumbnail each time takes a lot of resources. In a scenario like that it's much better to create a separate class that is called on a background thread, which will create the thumbnails and store the created images so your UITableView will be much more responsive during loading and scrolling.

## Creating a PDF Document

You've now learned how to read and present PDF documents in your application. Wouldn't it be cool to generate a PDF document from within your application from data that is available in your application?

Start Xcode and create a new project based on the Single View Application Project template, and name it CreatePDFDocument using the options shown in Figure 6-12.

In this exercise you'll use the YDCar object from Chapter 2 and create a simple application that will generate a PDF document with my favorite cars.

Open your project and add a reference to the QuickLook framework. Copy the YDCar class from Chapter 2 into your project.

You also need the images for the different makes and models as used in Chapter 2.

Open the YDViewController.xib file with Interface Builder and create a user interface as shown in Figure 6-13. Use the Assistant Editor to create the method declarations and property as shown in Listing 6-6.

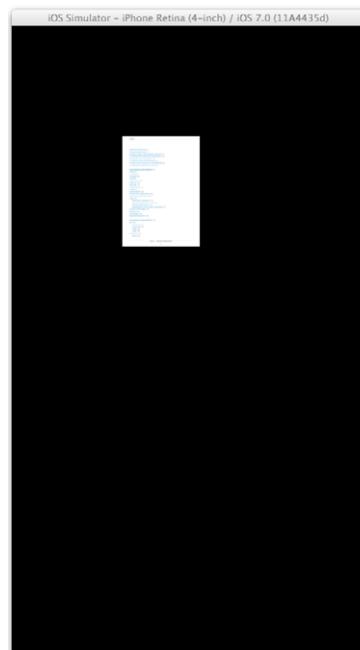


FIGURE 6-11

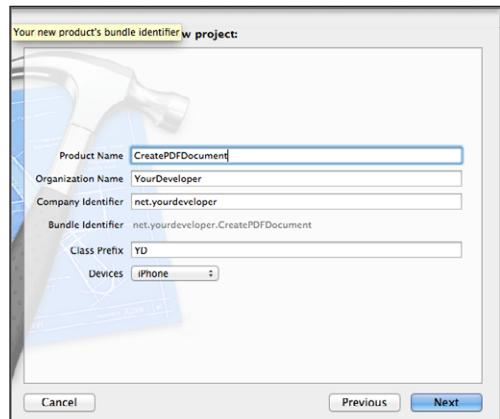


FIGURE 6-12

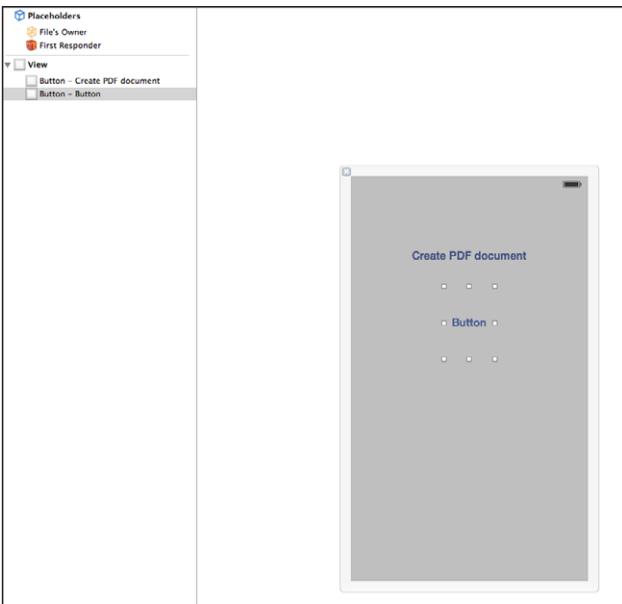


FIGURE 6-13

**LISTING 6-6:** Chapter6/CreatePDFDocument/YDViewController.h

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

@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIButton *previewButton;

- (IBAction)showDocument:(UIButton *)sender;
- (IBAction)createPDFDocument:(UIButton *)sender;

@property (nonatomic, strong) NSMutableArray* cars;
@end
```

In your `YDViewController.m` file you start by importing the QuickLook header file and subscribe to the `QLPreviewControllerDataSource`, `QLPreviewControllerDelegate`, and `QLPreviewItem` protocols as you've done before. In the `viewDidLoad` method you start by calling the `loadCars` method to initialize your `self.cars` array with data. Next, you create the `previewController` instance and set its delegate and datasource.

When the Create PDF Document button is tapped, the `createPDFDocument:` method will be called to generate a new PDF document and save it to the filesystem as `cars.pdf`.

The generation of the PDF document is using CoreGraphics functions for adding text, images, and other objects to the PDF context. To learn more about CoreGraphics, you can read the CoreGraphics framework reference at [http://developer.apple.com/library/ios/#documentation/CoreGraphics/Reference/CoreGraphics\\_Framework/\\_index.html](http://developer.apple.com/library/ios/#documentation/CoreGraphics/Reference/CoreGraphics_Framework/_index.html).

At the end of the `createPDFDocument:` method, after the PDF file has been generated, the `createThumbnailforPage:` method is called and the returned image is assigned to the `previewButton` button. When you tap the generated preview, the PDF document will show using QuickLook.

In the following three listings you see the implementation of the new introduced methods for this example. You can find the complete implementation of the `YDViewController.m` file in the download folder for this section.

The `createPDFDocument:` method is first calling the `setupPDFDocument:height:` method to create the file and open a `PDFContext`. The `UIGraphicsBeginPDFPageWithInfo` function creates a new page in the PDF context, and using a local helper method named `addText:withFrame:`, text is written to the PDF context with a frame. Next you loop over the `self.cars` array and access each of the cars in the array, and use the `addText:withFrame:` method to write the model of the car to the PDF context. Finally, you create a `UIImage` from the car `imageName` property and use the `addImage:atPoint:` helper method to write the car's image to the PDF context.

The `createPDFDocument:` method is shown in Listing 6-7.

#### LISTING 6-7: The `createPDFDocument:` method

```
- (IBAction)createPDFDocument:(UIButton *)sender
{
    [self setupPDFDocument:850 Height:2300];
    //Create a new Page
    UIGraphicsBeginPDFPageWithInfo(CGRectMake(0, 0,
PDFpageSize.width, PDFpageSize.height), nil);
    [self addText:@"My favorite cars!"
        withFrame:CGRectMake(20, 20, 400, 200) fontSize:48.0f];
    int y=200;
    for(int i=0;i<[self.cars count];i++)
    {
        YDCar* thisCar = [self.cars objectAtIndex:i];
        [self addText:[NSString stringWithFormat:
            @"%@ %@",thisCar.make,thisCar.model]
        withFrame:CGRectMake(20, y, 400, 50) fontSize:48.0f];
        y+=50;
        UIImage *anImage = [UIImage imageNamed:thisCar.imageName];
        CGRect imageRect = [self addImage:anImage
            atPoint:
            CGPointMake((PDFpageSize.width/2)-(anImage.size.width/2), y )];

        y+=anImage.size.height + 50.0f;
    }
    //finish the PDF Contents
    UIGraphicsEndPDFContext();
    //Set the image to thumbnail of page 1
    [self.previewButton setImage:[self createThumbnailforPage:1]
forState:UIControlStateNormal];
}
```

The `setupPDFDocument:height:` implementation is accepting the requested height of the PDF document and assigns this to the private variable `PDFpageSize`. It then calls the `UIGraphicsBeginPDFContextToFile` function to create the context and save it to the passed file.

The `setupPDFDocument:height:` method is shown in Listing 6-8.

#### LISTING 6-8: The `setupPDFDocument:height:` method

```
- (void)setupPDFDocument:(float)width height:(float)height{
    PDFpageSize = CGSizeMake(width, height);
    UIGraphicsBeginPDFContextToFile([self pdfFileName], CGRectMakeZero, nil);
}
```

The `addText:withFrame:` method is using CoreGraphics functions to calculate the size of the passed `NSString` and draw the passed text in a rectangle.

The `addText:withFrame:` method is shown in Listing 6-9.

#### LISTING 6-9: The `addText:withFrame:` method

```
//Helper function to add text to the Context
- (void)addText:(NSString*)text withFrame:(CGRect)frame fontSize:(float)fontSize {
    // UIFont *font = [UIFont systemFontOfSize:fontSize];
    CGSize textSize = [text sizeWithFont:[UIFont systemFontOfSize:fontSize]
constrainedToSize:CGSizeMake(PDFpageSize.width - 2*20-2*20,
    PDFpageSize.height - 2*20 - 2*20)
lineBreakMode:NSLineBreakByWordWrapping];
    float textWidth = frame.size.width;
    if (textWidth < textSize.width)
        textWidth = textSize.width;
    if (textWidth > PDFpageSize.width)
        textWidth = PDFpageSize.width - frame.origin.x;
    CGRect renderingRect =
        CGRectMake(frame.origin.x, frame.origin.y,
                   textWidth, textSize.height);
    [text drawInRect:renderingRect
        withFont:[UIFont systemFontOfSize:fontSize]
        lineBreakMode:NSLineBreakByWordWrapping
        alignment:NSTextAlignmentLeft];
}
```

The `addImage:atPoint:` method is using a CoreGraphics function to draw the image and is shown in Listing 6-10.

**LISTING 6-10:** The addImage:atPoint: method

```
//helper to addImage
- (void)addImage:(UIImage*)image atPoint:(CGPoint)point {
    CGRect imageFrame =
    CGRectMake(point.x, point.y, image.size.width, image.size.height);
    [image drawInRect:imageFrame];
}
```

The generated PDF document will appear in your application as shown in Figure 6-14.



FIGURE 6-14

## PLAYING AND RECORDING AUDIO

In this section you learn how to play and record audio in your application. You will learn which frameworks are involved and how to create applications that play and record audio.

## Introduction to the Frameworks

The iOS SDK contains several frameworks that are required for working with Audio on the iOS device.

### AVFoundation

- The AVFoundation framework provides classes for audio processing, such as:
  - **AVAudioPlayer:** This class is an audio player and provides the easiest solution to play audio files.
  - **AVAudioRecorder:** This class provides all the objects you need to record audio and save the recording.
  - **AVAudioSession:** This class provides methods for managing audio, routing changes, and mixing multiple audio files playing simultaneously.

### Audio Toolbox

The Audio Toolbox framework provides you with classes and objects to read and write file-based audio. The most common usage is for:

- **Audio Sessions:** Enable the device for managing audio sessions.
- **Hardware Services:** The API interface to work with the audio hardware.
- **System Services:** The API for playing simple small audio clips such as a notification.

### Media Player

The Media Player framework provides you a full, rich API for playing music, videos, and podcasts. It also enables you to access the iPod library on the device with all the details, such as albums, songs, artists, playlists, and related artwork.

## Playing an Audio File from the Bundle

If you want to play an audio file that is part of your application bundle, you create an instance of an `AVAudioPlayer`, set the `delegate`, and call the `prepareToPlay` and `play` methods of the `AVAudioPlayer` class.

Next you build an audio player application that will not only play an audio file from the bundle, but it will also provide a `UISlider` to set the volume, and a second `UISlider` to move to a certain point in the audio. The application will look similar to what is shown in Figure 6-15.

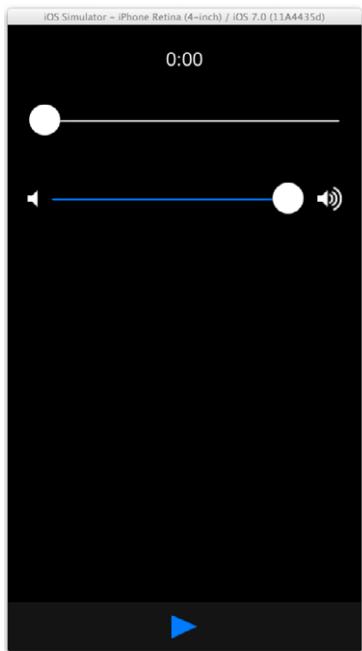


FIGURE 6-15

Start Xcode and create a new project by selecting the Single View Application Project template, and name it `CompleteAudioPlayer` using the options shown in Figure 6-16.

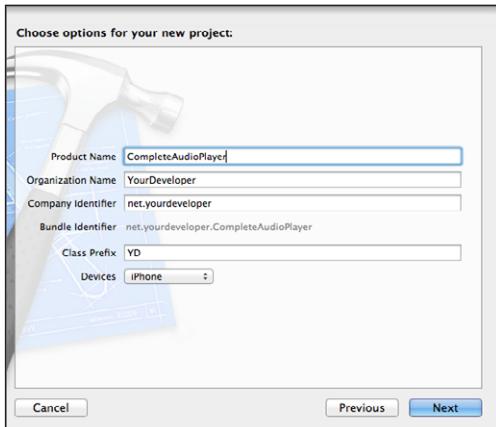


FIGURE 6-16

Add the AVFoundation framework to your project. In the download of this application project you will also find the following files:

- sample2ch.m4a
- sample.m4a
- volume\_down.png
- volume\_up.png
- pause.png
- play.png

Import these resources into your project.

Open the YDViewController.xib file with Interface Builder and design a user interface similar to the one shown in Figure 6-17.

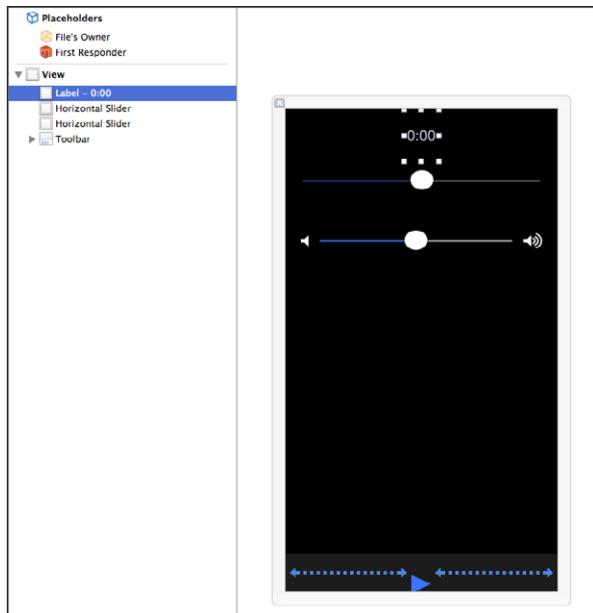


FIGURE 6-17

Use the Assistant Editor to create the properties and method declarations required for this example. In the YDViewController.h file, import the AVFoundation header file.

The complete YDViewController.h file is shown in Listing 6-11.

**LISTING 6-11:** Chapter6/CompleteAudioPlayer/YDViewController.h

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>
@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UILabel *trackStatus;
@property (weak, nonatomic) IBOutlet UISlider *trackSlider;
@property (weak, nonatomic) IBOutlet UISlider *volumeSlider;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *playButton;

@property (nonatomic, retain) AVAudioPlayer* player;

@property (nonatomic, strong) NSTimer* updateTimer;

@end
```

Open the `YDViewController.m` file and subscribe to the `AVAudioPlayer` delegate protocol. In the `viewDidLoad` method create an `NSURL` object to one of the imported audio files, and initialize the `AVAudioPlayer` instance. The `prepareToPlay` method of the `AVAudioPlayer` class is opening and reading the audio file and preparing the `AVAudioPlayer` to play the file.

The `playOrPause:` method is checking the current state of the `self.player` object and either calls the `play` or the `pause` method of the `AVAudioPlayer` class.

The `volumeSliderMoved:` method simply passed the value of the sender, which is a `UISlider`, to the `AVAudioPlayer`'s `volume` property.

The `trackSliderMoved:` method is setting the `currentTime` property of the `AVAudioPlayer` to the value of the `UISlider`. The `trackSlider` and `volumeSlider` values are updated in the `updateViewForPlayerInfo:` method.

The complete implementation is shown in Listing 6-12.

**LISTING 6-12:** Chapter6/CompleteAudioPlayer/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()<AVAudioPlayerDelegate>
- (IBAction)trackSliderMoved:(UISlider *)sender;
- (IBAction)volumeSliderMoved:(UISlider *)sender;
- (IBAction)playOrPause:(UIBarButtonItem *)sender;

@end

@implementation YDViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    NSURL *fileURL = [[NSURL alloc] initFileURLWithPath:
```

*continues*

**LISTING 6-12 (continued)**

```
[ [NSBundle mainBundle] pathForResource:@"sample2ch" ofType:@"m4a"]];  
    self.player = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL error:nil];  
    if (self.player)  
    {  
        [self updateViewForPlayerInfo:self.player];  
        [self updateViewForPlayerState:self.player];  
        [self.player prepareToPlay];  
        self.player.delegate = self;  
    }  
  
}  
- (IBAction)playOrPause:(UIBarButtonItem *)sender  
{  
    if (self.player)  
    {  
        if (self.player.playing)  
        {  
            [self.player pause];  
  
            [self.playButton setImage:[UIImage imageNamed:@"play.png"]];  
            [self updateViewForPlayerState:self.player];  
        }  
        else  
        {  
            [self.player play];  
            [self.playButton setImage:[UIImage imageNamed:@"pause"]];  
            [self updateViewForPlayerState:self.player];  
        }  
    }  
}  
  
- (IBAction)volumeSliderMoved:(UISlider *)sender  
{  
    self.player.volume = [sender value];  
}  
- (IBAction)trackSliderMoved:(UISlider *)sender  
{  
    self.player.currentTime = sender.value;  
    [self updateCurrentTimeForPlayer:self.player];  
}  
-(void)updateViewForPlayerInfo:(AVAudioPlayer*)p  
{  
    self.trackStatus.text = [NSString stringWithFormat:  
        @"%d:%02d", (int)p.duration / 60,  
        (int)p.duration % 60, nil];  
    self.trackSlider.maximumValue = p.duration;  
    self.volumeSlider.value = p.volume;  
}  
  
- (void)updateViewForPlayerState:(AVAudioPlayer *)p  
{
```

```
[self updateCurrentTimeForPlayer:p];
if (self.updateTimer!=nil)
    [self.updateTimer invalidate];

if (p.playing)
{
    [self.playButton setImage:[UIImage imageNamed:@"pause.png"]];
    self.updateTimer =
        [NSTimer scheduledTimerWithTimeInterval:.01 target:self
            selector:@selector(updatecurrentTime) userInfo:p repeats:YES];
}
else
{
    [self.playButton setImage:[UIImage imageNamed:@"play.png"]];
    self.updateTimer = nil;
}

}

-(void)updatecurrentTimeForPlayer:(AVAudioPlayer *)p
{
    self.trackStatus.text = [NSString stringWithFormat:@"%d:%02d",
    (int)p.currentTime / 60, (int)p.currentTime % 60, nil];
    self.trackSlider.value = p.currentTime;

}

-(void)updatecurrentTime
{
    [self updatecurrentTimeForPlayer:self.player];
}

-(void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player successfully:(BOOL)flag
{
    // [self.playButton setImage:[UIImage imageNamed:@"play.png"]];
    [self.updateTimer invalidate];
    self.updateTimer = nil;
}

-(void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

## Playing Audio from Your iTunes Library

The `MediaPlayer` framework is the framework you need to import into your project if you want to play audio from your iTunes library.

Start Xcode and create a new project using the Single View Application Project template, and name it MyTunesPlayer using the options shown in Figure 6-18.

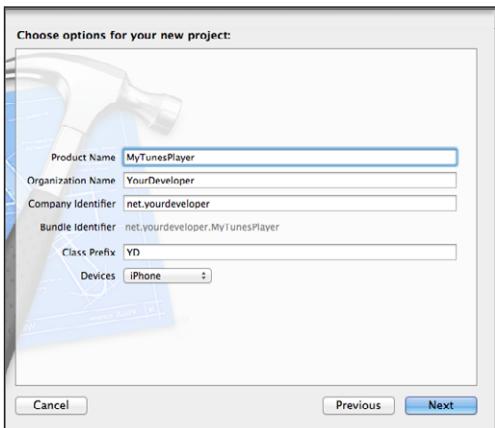


FIGURE 6-18

Add the MediaPlayer framework to your project.

Open the YDViewController.xib file with Interface Builder and put a UITableView object on the View object as shown in Figure 6-19.

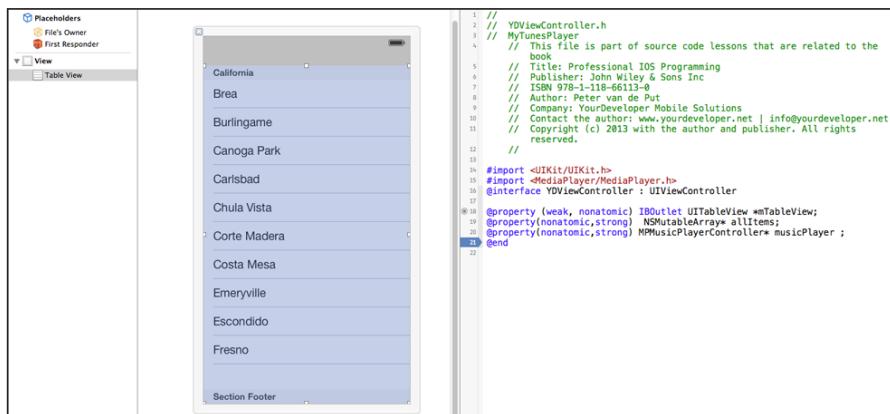


FIGURE 6-19

Use the Assistant Editor to create the weak property reference for the UITableView.

Open the YDViewController.h file and start with importing the MediaPlayer header file. Now create a strong property of type NSMutableArray named allItems to hold all the items from the iTunes library. Finally, create a strong property of type MPMusicPlayerController named musicPlayer, which is the object responsible for actually playing the selected song. The complete implementation of the YDViewController.h file is shown in Listing 6-13.

**LISTING 6-13:** Chapter6/MyTunesPlayer/YDViewController.h

```
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;
@property(nonatomic,strong) NSMutableArray* allItems;
@property(nonatomic,strong) MPMusicPlayerController* musicPlayer ;
@end
```

Now open the `YDViewController.m` file and start with subscribing to the `UITableViewDelegate` and `UITableViewDataSource` protocols.

In the `viewDidLoad` method you simply start with initializing the `self.musicPlayer` and call the `loadMedia` method. In the `loadMedia` method you start with creating an `MPMediaQuery` named `allSongQuery` by calling the static `songsQuery` method of the `MPMediaQuery` class. Now you can create the `self.allItems` array and initialize it by calling the `initWithArray:` method and pass the `allSongQuery` collections array.

In the `tableView:cellForRowAtIndexPath:` method you create a `UITableViewCell` and obtain an `MPMediaItem` from the `self.allItems` array for the selected row. To display the artwork for the song, you create an `MPMediaItemArtwork` object by calling the `valueForProperty:` `MPMediaItemPropertyArtwork` method on the `MPMediaItem` object. If the artwork exists, it is passed to the `imageView.image` property of the `UITableViewCell`.

You display the title of the song in the `textLabel` of the `UITableViewCell` by assigning the `valueForProperty:MPMediaItemPropertyAlbumTitle` value of the `MPMediaItem` to the `textLabel` `text` property.

In the `tableView:didSelectRowAtIndexPath:` method, the `musicPlayer` is stopped and its media queue is set to nil. The `MPMediaItem` is retrieved from the `allItems` array and an `MPMediaPropertyPredicate` object is created that is applied as a filter to the `songsQuery` by using the `addFilterPredicate`. The resulting `songsQuery` object is passed to the `musicPlayer` by calling the `setQueueWithQuery:` method.

The complete implementation is shown in Listing 6-14.

**LISTING 6-14:** Chapter6/MyTunesPlayer/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()<UITableViewDelegate,UITableViewDataSource>

@end

@implementation YDViewController

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

*continues*

**LISTING 6-14 (continued)**

```
self.musicPlayer=[MPMusicPlayerController applicationMusicPlayer];
[self loadMedia];
}

-(void)loadMedia
{
    //query all songs
    MPMediaQuery *allSongsQuery = [MPMediaQuery songsQuery];
    self.allItems =[NSMutableArray alloc]
    initWithArray:[allSongsQuery collections]];
    [self.mTableView reloadData];
}

#pragma mark tableView Delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

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

- (UITableViewCell *) tableView:(UITableView *) tableView
cellForRowAtIndexPath:(NSIndexPath *) indexPath
{
    UITableViewCell *cell = (UITableViewCell *)
    [tableView dequeueReusableCellWithIdentifier:@"MyCellIdentifier"];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"MyCellIdentifier"];
    }
    MPMediaItem *item =
    [[self.allItems objectAtIndex:indexPath.row] representativeItem];
    MPMediaItemArtwork *artwork =
    [item valueForProperty: MPMediaItemPropertyArtwork];
    if (artwork) {
        cell.imageView.image = [artwork imageWithSize: CGSizeMake (30, 30)];
    }
    cell.textLabel.text = [item valueForProperty:MPMediaItemPropertyAlbumTitle];

    return cell;
}
- (void) tableView:(UITableView *) tableView
didSelectRowAtIndexPath:(NSIndexPath *) indexPath
{
    [self.musicPlayer stop];
    [self.musicPlayer setQueueWithItemCollection:nil];
    MPMediaItem *item =
    [[self.allItems objectAtIndex:indexPath.row] representativeItem];
    MPMediaPropertyPredicate *myPredicate =
    [MPMediaPropertyPredicate predicateWithValue:
    [item valueForProperty:MPMediaItemPropertyAlbumPersistentID]
```

```

        forProperty:MPMediaItemPropertyAlbumPersistentID

        comparisonType:MPMediaPredicateComparisonContains];
MPMediaQuery *songsQuery = [MPMediaQuery songsQuery];
[songsQuery addFilterPredicate:myPredicate];
//set Query direct to Queue
[self.musicPlayer setQueueWithQuery:songsQuery];
[self.musicPlayer prepareToPlay];
[self.musicPlayer play];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

When you run this application in the simulator you won't get any results since there is no iTunes library on the simulator. Run it on your device instead.

## Playing Streaming Audio

If you want your application to play streaming audio, take a good look at the technical information on the Apple developer website following this direct link: <https://developer.apple.com/library/mac/#documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/Introduction/Introduction.html>.

It's not within the scope of this book to set up a complete server- and client-side infrastructure for working with live HTTP streams; however, in this section you learn the basic method for playing an HTTP stream on an iOS device.

Start Xcode and create a new project using the Single View Application Project template, and name it `StreamingAudioAndVideo` using the options shown in Figure 6-20.

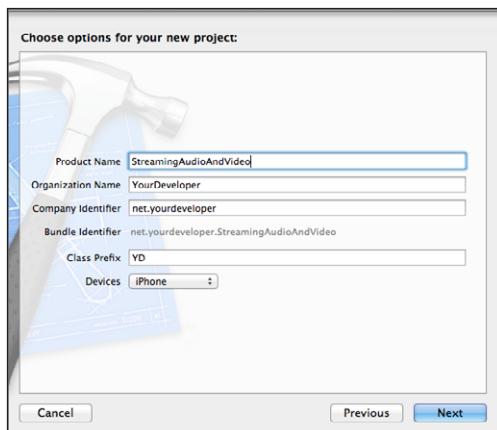


FIGURE 6-20

You must import the MediaPlayer framework into your project because the object used to play the audio stream is the MPMoviePlayerController. Apple has prepared a live stream server containing the HTML pages with the embedded information about the stream objects.

Open the YDViewController.xib file with Interface Builder and place a UIButton on the View, and use the Assistant Editor to create a method declaration as shown in Figure 6-21.

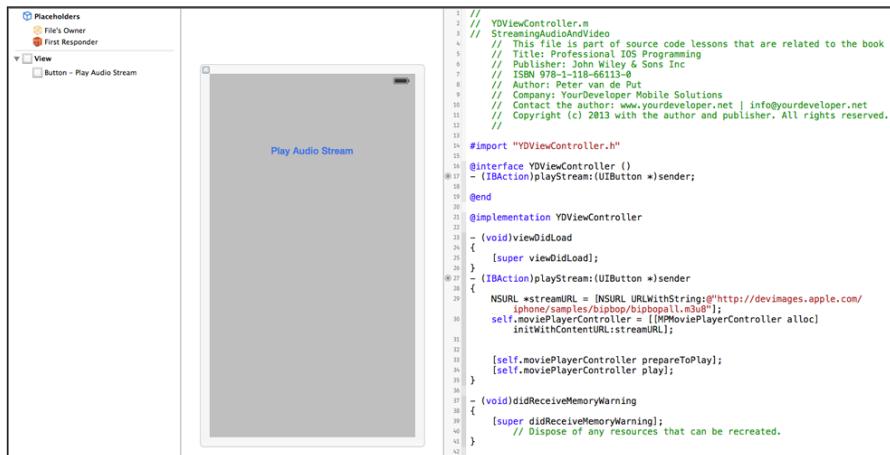


FIGURE 6-21

Open your YDViewController.h file and import the MediaPlayer header. Create a strong property of type MPMoviePlayerController named moviePlayerController as shown in Listing 6-15.

#### LISTING 6-15: Chapter6/StreamingAudioAndVideo/YDViewController.h

```

import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@interface YDViewController : UIViewController
@property(nonatomic,strong) MPMoviePlayerController* moviePlayerController;
@end

```

Open the YDViewController.m file and implement the playStream: method as shown in Listing 6-16.

You start by creating an NSURL object for the URL prepared by Apple, which is `http://devimages.apple.com/iphone/samples/bipbop/bipbopall.m3u8`. Next, you initialize the `self.moviePlayerController` by calling the `initWithContentURL:` method of the `MPMoviePlayerController` object, passing the just created `NSURL` object. Finally, you call the `prepareToPlay` and the `play` methods of the `MPMoviePlayerController` object.

**LISTING 6-16:** Chapter6/StreamingAudioAndVideo/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()  
- (IBAction)playStream:(UIButton *)sender;  
  
@end  
  
@implementation YDViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
}  
- (IBAction)playStream:(UIButton *)sender  
{  
    NSURL *streamURL = [NSURL URLWithString:  
@"http://devimages.apple.com/iphone/samples/bipbop/bipbopall.m3u8"];  
    self.moviePlayerController = [[MPMoviePlayerController alloc]  
initWithContentURL:streamURL];  
  
    [self.moviePlayerController prepareToPlay];  
    [self.moviePlayerController play];  
}  
  
- (void)didReceiveMemoryWarning  
{  
    [super didReceiveMemoryWarning];  
    // Dispose of any resources that can be recreated.  
}
```

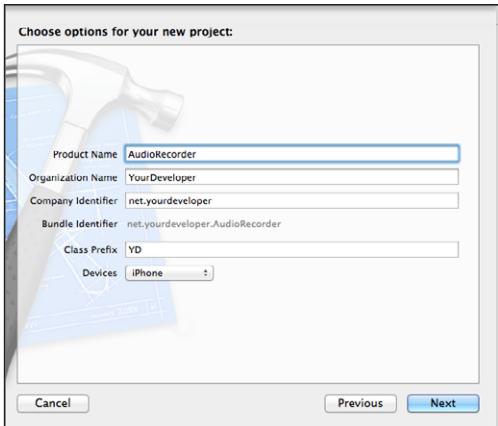
  
@end

## Recording Audio

The AVFoundation framework also contains a class named `AVAudioRecord` that you can use to record audio files.

In the next application you'll be able to record audio on an iOS device and play back the recording.

Start Xcode and create a new project based on the Single View Application Project template, and name it `AudioRecorder` using the options shown in Figure 6-22.



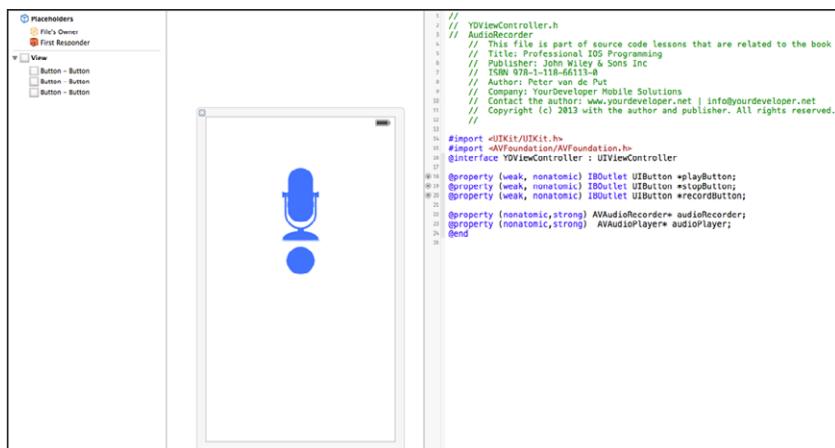
**FIGURE 6-22**

Import the AVFoundation framework into your project. For this example, the following images are used:

- > playbutton.png
  - > mic.png
  - > stopbutton.png
  - > recordbutton.png

All these images are available in the download section of this chapter.

Open the `YDViewController.xib` file and create a user interface similar to the one shown in Figure 6-23.



**FIGURE 6-23**

You see three `UIButton` objects on the View, two of which have been put on top of each other. Use the Assistant Editor to create the weak property references for the `UIButton` objects.

Open your `YDViewController.h` file and start with importing the `AVFoundation` header file. Now create a strong property declaration of type `AVAudioRecorder` named `audioRecorder` and one of type `AVAudioPlayer` named `audioPlayer` as shown in Listing 6-17.

#### LISTING 6-17: Chapter6/Recorder/YDViewController.h

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>
@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIButton *playButton;
@property (weak, nonatomic) IBOutlet UIButton *stopButton;
@property (weak, nonatomic) IBOutlet UIButton *recordButton;

@property (nonatomic, strong) AVAudioRecorder* audioRecorder;
@property (nonatomic, strong) AVAudioPlayer* audioPlayer;
@end
```

Creating an audio recording is really simple, as you will learn.

Open the `YDViewController.m` file and start by subscribing to the `AVAudioRecorderDelegate` and `AVAudioPlayerDelegate` protocols. Using the Assistant Editor, create the three declarations for the `startRecording:`, `startPlaying:`, and `stopRecording:` methods.

In the `viewDidLoad` method you set the hidden property for the `stopButton` and `playButton`. They will only be enabled if a recording has been made.

When the user taps the recording button, the `startRecording:` method is invoked and the `AVAudioRecord` object is created with an `NSURL` to a local file where the recording will be stored. An `NSDictionary` is created containing the settings for the recording and encoding and is passed to the `AVAudioRecorder` instance. The `self.audioRecorder` delegate property is set to `self` so the delegate callbacks can be implemented.

You implement the delegate methods to update the user interface once the recording is finished and enable the play button. Play back the recorded audio like you've done in the previous examples. The images used are part of the download project. The complete implementation is shown in Listing 6-18.

#### LISTING 6-18: Chapter6/Recorder/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()<AVAudioRecorderDelegate,AVAudioPlayerDelegate>
- (IBAction)startRecording:(UIButton *)sender;
- (IBAction)startPlaying:(UIButton *)sender;
- (IBAction)stopRecording:(UIButton *)sender;

@end
```

*continues*

**LISTING 6-18 (continued)**

```
@implementation YDViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    [self.stopButton setHidden:YES];
    [self.playButton setHidden:YES];

}
-(NSURL *)soundFileURL
{
    NSArray *dirPaths;
    NSString *docsDir;
    dirPaths =
    NSSearchPathForDirectoriesInDomains( NSDocumentDirectory, NSUserDomainMask, YES);
    docsDir = [dirPaths objectAtIndex:0];
    NSString *soundFilePath =
    [docsDir stringByAppendingPathComponent:@"recording.m4a"];
    return [NSURL fileURLWithPath:soundFilePath];
}
- (IBAction)startRecording:(UIButton *)sender
{
    [self.recordButton setHidden:YES];
    [self.stopButton setHidden:NO];
    [self.playButton setHidden:YES];
    //Create a dictionary for the recording settings
    NSMutableDictionary *recordSetting = [[NSMutableDictionary alloc] init];
    [recordSetting setValue:
    [NSNumber numberWithInt:kAudioFormatMPEG4AAC]
    forKey:AVFormatIDKey];//mpeg 4 aac
    [recordSetting setValue:
    [NSNumber numberWithFloat:44100.0]
    forKey:AVSampleRateKey];//sample rate
    [recordSetting setValue:
    [NSNumber numberWithInt: 2]
    forKey:AVNumberOfChannelsKey];//2 channels
    [recordSetting setValue:
    [NSNumber numberWithInt:16]
    forKey:AVLinearPCMBitDepthKey];//16 bits
    [recordSetting setValue:
    [NSNumber numberWithBool:NO]
    forKey:AVLinearPCMIsBigEndianKey];
    [recordSetting setValue:
    [NSNumber numberWithBool:NO]
    forKey:AVLinearPCMIsFloatKey];
    NSError *error = nil;

    self.audioRecorder = [[AVAudioRecorder alloc]
    initWithURL:[self soundFileURL]
    settings:recordSetting
    error:&error];
}
```

```

        self.audioRecorder.delegate=self;
        if (error)
        {
            NSLog(@"error: %@", [error localizedDescription]);
        } else {
            [self.audioRecorder prepareToRecord];
        }
        if (!self.audioRecorder.recording)
        {
            self.recordButton.enabled = NO;
            self.stopButton.enabled = YES;
            [self.audioRecorder record];
        }
    }
    - (IBAction)stopRecording:(UIButton *)sender
{
    [self.stopButton setHidden:YES];
    [self.recordButton setHidden:NO];
    self.recordButton.enabled = YES;
    self.stopButton.enabled = NO;
    [self.audioRecorder stop];
    self.audioRecorder=nil;
    [self.playButton setHidden:NO];
}
- (IBAction)startPlaying:(UIButton *)sender
{
    [self.playButton setUserInteractionEnabled:NO];
    NSURL *url = [self soundFileURL];
    NSError *error = nil;
    self.audioPlayer = [[AVAudioPlayer alloc]
                        initWithContentsOfURL:url
                        error:&error];
    if (error)
    {
        NSLog(@"Error in audioPlayer: %@", [error localizedDescription]);
    } else {
        self.audioPlayer.delegate = self;
        [self.audioPlayer prepareToPlay];
        [self.audioPlayer play];
    }
}
#pragma delegates
-(void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player successfully:(BOOL)flag
{
    self.audioPlayer = nil;

    [self.playButton setUserInteractionEnabled:YES];
}
-(void)audioRecorderDidFinishRecording:(AVAudioRecorder *)recorder
                                successfully:(BOOL)flag
{

```

*continues*

**LISTING 6-18 (continued)**

```
UIAlertView* alert = [[UIAlertView alloc]
initWithTitle:@"Thanks"
message:@"Your recording has finished" delegate:self
cancelButtonTitle:@"Ok" otherButtonTitles:nil, nil];
[alert show];

}
-(void)audioRecorderEncodeErrorDidOccur: (AVAudioRecorder *)recorder
error:(NSError *)error
{
    UIAlertView* alert = [[UIAlertView alloc]
initWithTitle:@"Error" message:@"An error has occurred"
delegate:self cancelButtonTitle:@"Ok"
otherButtonTitles:nil, nil];
[alert show];
}
-(void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
@end
```

## Supported Audio Formats

The following list shows which audio formats are supported:

- AAC (MPEG-4) extension (also .m4a)
- ALAC (Apple Lossless)
- HE\_AAC (MPEG-4 High Efficiency AAC)
- iLBC
- iMA4 (IMA/ADPCM)
- Linear PCM
- MP3 (MPEG-1 audio layer 3)

## PLAYING AND RECORDING VIDEO

Playing and recording video has a lot in common with playing and recording audio. Recording video can, of course, only be done with devices that have a camera and not by using the simulator.

## Playing a Video File from the Bundle

You'll use the MediaPlayer framework to play a video that is stored in your bundle.

Start Xcode and create a new project using the Single View Application Project template, and name it VideoPlayer using the options shown in Figure 6-24.

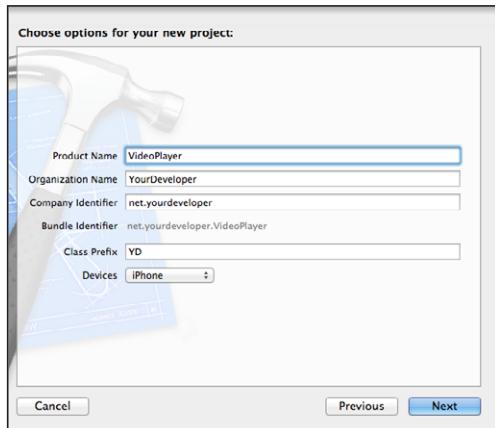


FIGURE 6-24

Import the MediaPlayer framework in your project. Open the YDViewController.xib file using Interface Builder and create a user interface as shown in Figure 6-25.



FIGURE 6-25

Open your ViewController.h file and import the MediaPlayer framework. Create a strong property of type MPMoviePlayerController named as shown in Listing 6-19.

**LISTING 6-19:** Chapter6/VideoPlayer/YDViewController.h

```
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@interface YDViewController : UIViewController

@property(nonatomic, strong) MPMoviePlayerViewController *moviePlayerView;
@property (weak, nonatomic) IBOutlet UIButton *moviePlayButton;
@end
```

Open your `YDViewController.m` file and in your `viewDidLoad` method create your `self.moviePlayerView` instance by passing the `NSURL` to the file in your bundle. Create a thumbnail from the video by calling the `thumbnailImageAtTime:timeOption:` method of the `MPMoviePlayerController`, passing the time in seconds. The `timeOption` parameter has the value `MPMovieTimeOptionNearestKeyFrame` to create a thumbnail image of the movie, by creating an image from the frame nearest the number of seconds passed.

Set the created `UIImage` thumbnail as the image for the `self.moviePlayButton`.

The `playVideo:` method calls the `presentMoviePlayerViewControllerAnimated:` method, providing your application with a user interface in which the movie is displayed. The complete listing is shown in Listing 6-20.

**LISTING 6-20:** Chapter6/VideoPlayer/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()
@property (weak, nonatomic) IBOutlet UIButton *moviePlayButton;
- (IBAction)playVideo:(UIButton *)sender;

@end

@implementation YDViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    self.view.backgroundColor = [UIColor whiteColor];
    self.moviePlayerView = [[MPMoviePlayerViewController alloc]
initWithContentURL:[NSURL fileURLWithPath:[[NSBundle mainBundle]
pathForResource:@"Movie" ofType:@"m4v"]]];
    [[NSNotificationCenter defaultCenter]
addObserver:self selector:@selector(thumbnailReady:)
name:MPMoviePlayerThumbnailImageRequestDidFinishNotification
object:nil];

NSMutableArray *timeArr = [[NSMutableArray alloc] init];
[timeArr addObject:[NSNumber numberWithFloat:1.f]];
[self.moviePlayerView.moviePlayer requestThumbnailImagesAtTimes:timeArr timeOption:MPMovieTimeOptionNearestKeyFrame];
```

```
}

- (void) thumbnailReady:(NSNotification*)notification
{
    NSDictionary *userInfo = [notification userInfo];
    UIImage *image =
    [userInfo objectForKey: @"MPMoviePlayerThumbnailImageKey"];
    [self.moviePlayButton setBackgroundImage:image forState:UIControlStateNormal];
}

- (IBAction)playVideo:(UIButton *)sender
{

    [self presentMoviePlayerViewControllerAnimated:self.moviePlayerView];
    self.moviePlayerView.moviePlayer.movieSourceType = MPMovieSourceTypeFile;
    [self.moviePlayerView.moviePlayer play];
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Your application will look as shown in Figure 6-26.

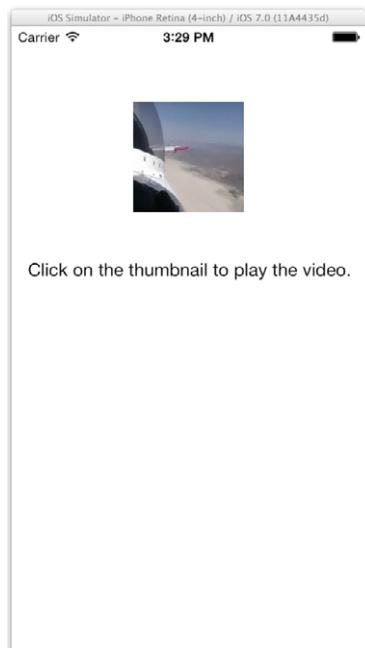


FIGURE 6-26

## Playing a Video from Your iTunes Library

Playing a video from your iTunes library (also referred to as the iPod library) is very similar to playing audio from the iTunes library.

Start Xcode and create a new project using the Single View Application Project template, and name it iTunesVideo using the options shown in Figure 6-27.

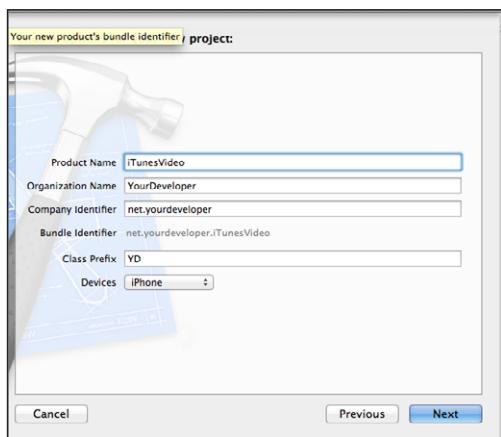


FIGURE 6-27

Import the MediaPlayer framework in your project.

Open the YDViewController.xib file with Interface Builder and place a UITableView on the View, and create a weak property reference for this UITableView as shown in Figure 6-28.

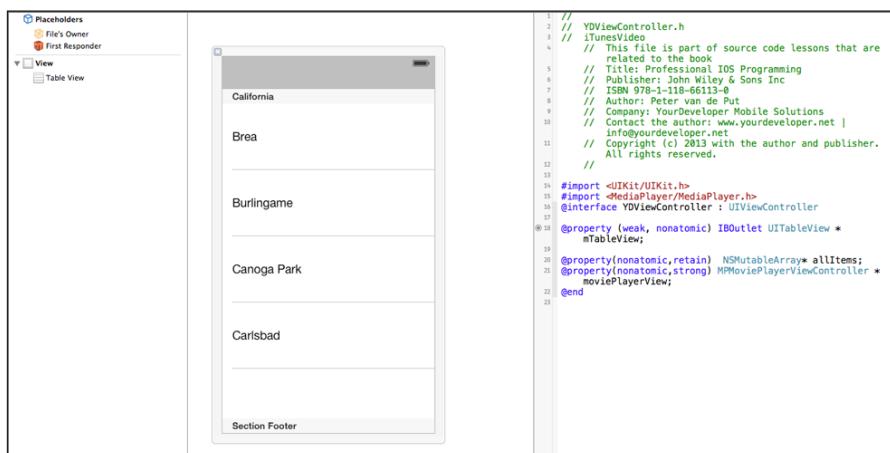


FIGURE 6-28

Open your ViewController.h file, import the MediaPlayer header file, and create a strong property of type MMMoviePlayerViewController as shown in Listing 6-21.

**LISTING 6-21:** Chapter6/iTunesVideo/YDViewController.h

```
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;

@property(nonatomic,retain) NSMutableArray* allItems;
@property(nonatomic,strong) MPMoviePlayerViewController *moviePlayerView;
@end
```

Open the `YDViewController.m` file and subscribe to the `UITableViewDelegate` and `UITableViewDataSource` protocols.

In the `viewDidLoad` method you initialize the `self.moviePlayerView` and call the `loadMedia` method. In the `loadMedia` method you create an `MPMediaQuery` by passing an `MPMediaPropertyPredicate` to retrieve only the `MPMediaItems` of type `video`. The result is stored in the `allItems` array and the `reloadData` method of the `mTableView` is called.

In the `tableView:didSelectRowAtIndexPath:` method you create an `MPMediaItem` for the selected row and create an `NSURL` object to the asset's media. You use the `moviePlayerView.moviePlayer` to play the movie. The complete implementation is shown in Listing 6-22.

**LISTING 6-22:** Chapter6/iTunesVideo/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()<UITableViewDelegate,UITableViewDataSource>

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    //create only once
    self.moviePlayerView = [[MPMoviePlayerViewController alloc]
    initWithContentURL:nil];
    [self loadMedia];
}

-(void)loadMedia
{
    NSNumber *videoTypeNum = [NSNumber numberWithInteger:MPMediaTypeAnyVideo];
    MPMediaPropertyPredicate *videoPredicate =
    [MPMediaPropertyPredicate predicateWithValue:videoTypeNum
    forProperty:MPMediaItemPropertyMediaType];
    MPMediaQuery *videoQuery = [[MPMediaQuery alloc] init];
    [videoQuery addFilterPredicate: videoPredicate];
    self.allItems = [[NSMutableArray alloc]
```

*continues*

**LISTING 6-18 (continued)**

```

        initWithArray:[videoQuery collections]];
    [self.tableView reloadData];

}

#pragma mark tableViewDelegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}
- (NSInteger)tableView:(UITableView *)table numberOfRowsInSection:(NSInteger)section {
    return [self.allItems count];
}

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

    UITableViewCell *cell = (UITableViewCell *)[tableView
dequeueReusableCellWithIdentifier:@"MyCellIdentifier"];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"MyCellIdentifier"];
    }
    cell.selectionStyle = UITableViewCellSelectionStyleNone;
    MPMediaItem *item =
[[self.allItems objectAtIndex:indexPath.row] representativeItem];
    MPMediaItemArtwork *artwork =
[item valueForProperty:MPMediaItemPropertyArtwork];
    if (artwork) {
        cell.imageView.image = [artwork imageWithSize:CGSizeMake (100,100)];
    }
    cell.textLabel.text = [item valueForProperty:MPMediaItemPropertyTitle];
    return cell;
}
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    MPMediaItem *item =
[[self.allItems objectAtIndex:indexPath.row] representativeItem];
    NSURL *assetURL=[item valueForProperty:MPMediaItemPropertyAssetURL];
    [self.moviePlayerView.moviePlayer setContentURL:assetURL];
    [self presentMoviePlayerViewControllerAnimated:self.moviePlayerView];
    self.moviePlayerView.moviePlayer.movieSourceType = MPMovieSourceTypeFile;
    [self.moviePlayerView.moviePlayer play];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

This application will not run in the simulator since there is no iTunes library available. Instead you run it on a device.

When you launch the application it will display a list of the videos found on your device. When you tap on a row, it will play the selected video.

## Playing a YouTube Video

To play a YouTube video in your iOS application you need to:

- Obtain a URL to the video
- Set up a `UIWebView` to display the video

Start Xcode and create a new project using the Single View Application Project template, and name it `YouTubePlayer` using the options shown in Figure 6-29.

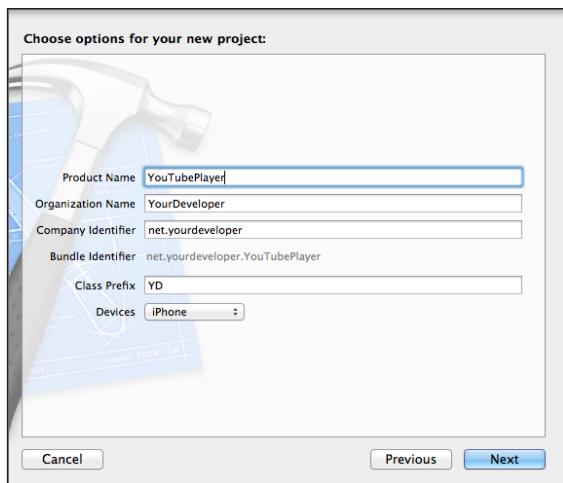


FIGURE 6-29

Open the `YDViewController.xib` file with Interface Builder and put a `UIWebView` on the View. Use the Assistant Editor to create the property reference as shown in Figure 6-30.

Open your `YDAppDelegate.m` file and in the `application:didFinishLaunchingWithOptions:` method, you need to implement the following code snippet:

```
[ [NSHTTPCookieStorage sharedHTTPCookieStorage]
    setCookieAcceptPolicy:NSHTTPCookieAcceptPolicyAlways]
```

This call is required because YouTube requires your `UIWebView` to accept cookies.

**FIGURE 6-30**

Open your `YDViewController.m` file and subscribe to the `UIWebViewDelegate` protocol. In the `viewDidLoad` method, create an HTML `NSString` containing the `iFrame` tag and URL to the video you would like to play, and then call the `loadHTMLString:baseURL:` method of the `UIWebView` class as shown in Listing 6-23.

**LISTING 6-23: Chapter6/YouTubePlayer/YDViewController.m**

```

#import "YDViewController.h"

@interface YDViewController ()<UIWebViewDelegate>

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    NSString *embedHTML = @"
<html><head>
<style type=\"text/css\">
body {
background-color: transparent;
color: white;
}
</style>
</head><body style=\"margin:0;\">
<iframe title=\"YouTube Video\" class=\"youtube-player\" type=\"text/html\" width=\"320\" height=\"460\" src=\"%@\" frameborder=\"0\" allowFullScreen ></iframe>";

```

```
NSString *urlToOpen = @"http://www.youtube.com/embed/u1zgFlCw8Aw?autoplay=1";
NSString *html = [NSString stringWithFormat:embedHTML, urlToOpen];

[self.youtubeViewer loadHTMLString:html baseURL:nil];

}

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

@end
```

## Recording Video

The iPhone has a camera and the iPad 2 and later models also have a camera built-in. You can use the camera to take pictures or to record videos. In this section you develop a simple application that will record a video file and save it on your device. Please note that this application will not run on the simulator because there is no camera available in the simulator.

Start Xcode and create a new project using the Single View Application Project template, and name it `VideoRecorder` using the options shown in Figure 6-31.

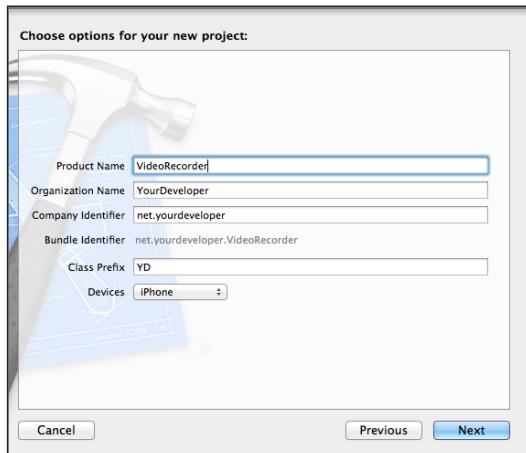


FIGURE 6-31

Import the `MobileCoreServices` framework to have access to the APIs you require in this application.

Open your `YDViewController.h` file and create a strong property of type `UIImagePickerController` named `camera` as shown in Listing 6-24.

**LISTING 6-24:** Chapter6/VideoRecorder/YDViewController.h

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

@interface YDViewController : UIViewController
@property(nonatomic,strong) UIImagePickerController* camera;
@end
```

Open your `YDViewController.m` file and import the `MobileCoreServices` header file.

The `recordVideo:` method first tests if a camera is available, and if so, it will initialize it and present the `UIImagePickerController` instance.

You need to implement the `imagePickerController:didFinishPickingMediaWithInfo:` delegate method that will dismiss the `UIImagePickerController` and save the recorded video to the Album.

Also, you need to implement the `video:didFinishSavingWithError:contextInfo:` delegate method to handle errors correctly. The complete implementation is shown in Listing 6-25.

**LISTING 6-25:** Chapter6/VideoRecorder/YDViewController.m

```
#import "YDViewController.h"
#import <MobileCoreServices/MobileCoreServices.h>
@interface YDViewController ()<UIImagePickerControllerDelegate>
- (IBAction)recordVideo:(UIButton *)sender;

@end

@implementation YDViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
}
- (IBAction)recordVideo:(UIButton *)sender
{
    //Test if there is a camera available
    if ([UIImagePickerController isSourceTypeAvailable:
         UIImagePickerControllerSourceTypeCamera])
    {
        self.camera = [[UIImagePickerController alloc] init];
        self.camera.sourceType = UIImagePickerControllerSourceTypeCamera;
        self.camera.mediaTypes = [[NSArray alloc]
initWithObjects:(NSString *)kUTTypeMovie, nil];
        self.camera.showsCameraControls=YES;
        self.camera.delegate = self;
        [self presentViewController:self.camera animated:YES completion:nil];
    }
    else
    {
        UIAlertView* alert = [[UIAlertView alloc]
```

```
initWithTitle:@"Error" message:@"No camera available"
delegate:nil cancelButtonTitle:@"Ok"
otherButtonTitles:nil, nil];
    [alert show];
}
}

#pragma mark UIImagePickerController delegates
-(void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    NSString *mediaType = [info objectForKey: UIImagePickerControllerMediaType];
    [self dismissViewControllerAnimated:YES completion:nil];
    // Handle a movie capture
    if (CFStringCompare (( CFStringRef) mediaType, kUTTypeMovie, 0) ==
kCFCompareEqualTo) {
        NSString *moviePath = (NSString *)[[info objectForKey: UIImagePickerControllerMedia
URL] path];

        if (UIVideoAtPathIsCompatibleWithSavedPhotosAlbum(moviePath)) {
            UISaveVideoAtPathToSavedPhotosAlbum(moviePath, self,
@selector(video:didFinishSavingWithError:contextInfo:), nil);
        }
    }
}

-(void)video:(NSString*)videoPath didFinishSavingWithError:(NSError*)error
contextInfo:(void*)contextInfo
{
    if (error) {
        UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Error" message:@"Video Saving Failed"
delegate:nil cancelButtonTitle:@"OK"
otherButtonTitles:nil];
        [alert show];
    } else {
        UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Video Saved" message:@"Saved to your Photo Album"
delegate:self cancelButtonTitle:@"OK"
otherButtonTitles:nil];
        [alert show];
    }
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

## SUMMARY

In this chapter you learned all the techniques to enrich your application with multimedia. You are now able to:

- Present PDF documents with a low memory footprint.
- Play audio from different source.
- Access the iTunes library.
- Record Audio.
- Play video files from your application bundle.
- Play YouTube videos.

In Part II you learn all you need to know about networking concepts and data processing. Communicating with web services, like SOAP and REST servers, processing the incoming data stream, parsing the data using either JSON or XML, and storing data are discussed. Data storage using the Core Data framework is presented in full detail, which will enable you to develop world-class applications.

## PART II

# Networking—Data Processing

---

- ▶ **CHAPTER 7:** Using Web Services and Parsing
- ▶ **CHAPTER 8:** Using FTP
- ▶ **CHAPTER 9:** Implementing Core Data



# 7

# Using Web Services and Parsing

## WHAT'S IN THIS CHAPTER?

---

- Calling RESTful web services and SOAP services
- Parsing the response in JSON and XML

In this chapter you learn all the relevant techniques and technologies to connect your iOS application to the outside world using web services.

## WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 7 download and individually named according to the names throughout the chapter.

## WHY WOULD YOU NEED TO USE A WEB SERVICE?

Very few applications work completely independent of the Internet to provide their functionality. Each application must contain data to present to the user interface. This data can be pictures, videos, documents, or just plain text.

If the data your application requires is static, like PDF documents, images, or even videos, you can choose to incorporate these files in your application bundle so it is part of the application and is available once the application is installed on an iOS device.

The App Store review guidelines state very clearly in Article 2.15 that if an application is larger than 50 MB it won't download over cellular networks but only over WIFI networks. For that reason it's better to not embed a large amount of data in your bundle but use a remote service, as will be explained in this chapter.

When your data is not static but changes frequently, embedding it as part of the bundle means that you need to build a new version with the new data embedded and go through the review process again. In addition, the new content is available to users only after they perform an upgrade of your application, which is a process you can't influence or control.

## UNDERSTANDING BASIC NETWORKING

The fundamentals of networking are described in the *Open Systems Interconnection* (OSI) model. OSI describes the architecture of networking by defining seven different layers, stacked on top of each other, in which each layer has its specific function. The following layers are defined:

- Layer 7: Application layer
- Layer 6: Presentation layer
- Layer 5: Session layer
- Layer 4: Transport layer
- Layer 3: Network layer
- Layer 2: Data link layer
- Layer 1: Physical layer

When implementing networking for your application, you will mainly focus on layers 6 and 7 by using protocols like HTTP and FTP and performing operations on those layers.

It's not within the scope of this book to completely describe the networking technology and techniques for the different layers in the OSI model, but this basic understanding is quite essential.

## Understanding Protocols

Once you have established a network connection, which is a connection between your application's device and a host, you must choose the protocol used to exchange data packets between the sender and the recipient.

The most common protocols used are HTTP, and FTP. The HTTP protocol is the protocol of the web. An HTTP server is nothing more than a web server that serves web pages to an HTTP client, which then interprets this data and presents it in a user interface. The most commonly known HTTP servers are Internet Information Server (IIS) from the Microsoft Server product family, and Apache, a web server that runs on a UNIX environment.

If you use the Internet, you use an HTTP client (your web browser is an HTTP client). On an iOS device, the Safari application is an HTTP client, and the `UIWebView` component is capable of interpreting HTTP responses and presenting them to the user interface.

In a standard configuration, HTTP servers and clients communicate using port 80 on the IP address.

The HTTPS protocol is derived from the HTTP protocol, and the additional S stands for Secure. On the server side this means that a different port is used—by default, port 443—for this secured connection. The protocol extension requires a signed certificate to be installed on the server side, and both the client and server encrypt the data sent back and forth to avoid interception by third parties.

The *File Transfer Protocol* (FTP) is used to exchange files between the server and client. In Chapter 8 you learn to implement an FTP client in your application.

## Understanding Operations

When your application is communicating with a web service using the HTTP protocol, different operations are available for your request. The most used are the HEAD, GET, and POST operations.

The HEAD operation sends a request to the web server to return the information about the server. The return is a message of variable length that indicates, at minimum, the HTTP version that's supported and a response code. You can use a HEAD operation to simply test whether a certain URL is available without requesting the content of the specified URL.

The GET operation is used to tell the server you are contacting that you want to “get” the data you are requesting, and the POST operation is used to send information to the web server, such as a web form with filled-in form fields.

Each of these methods is explained with examples in the following sections.

## Understanding Response Codes

When you send an HTTP request to a web service, you always receive a response code that is related to the request. Many different response codes are available, and on the server side the system administrator can control how and which responses are sent to the requesting client. Nevertheless, you need to use some standardized response codes to verify the response you are receiving before processing the response itself.

The most commonly used response codes are:

- **Code 200:** The request has succeeded.
- **Code 301:** The resource has moved permanently.
- **Code 400:** Bad request.
- **Code 401:** Unauthorized.
- **Code 403:** Forbidden.
- **Code 404:** Not found.
- **Code 408:** Request timed out.
- **Code 5xx:** All codes starting with 5 indicate a server error.

Before you process the response, it's important to test that the response code is 200, telling you the request has been executed by the server.

## INTRODUCTION TO WEB SERVICES

A *web service* was originally defined by the W3C council as a software system designed to support machine-to-machine interaction over a network. A web service should have an interface description that can be processed by a machine.

The world has evolved since this definition, and *web service* is now defined as all communications between the application you are developing and a hosted service, connectable via the Internet, that responds to HTTP requests.

With this definition in place, in the following sections you learn to interact with an HTTP service, a REST service, and a SOAP service.

## CALLING AN HTTP SERVICE

The most commonly known web service is the HTTP service that is used to serve a website. Each website you access is running on an HTTP server, and if you make a request to an HTTP service you'll get an HTTP response back that you can process.

Two main objects are involved in calling an HTTP service: the `NSURLRequest` object and the `NSURLConnection` object.

The `NSURLRequest` object represents a URL load request independent of protocol and URL scheme. The main constructor accepts an `NSURL` object containing the URL to load, and optionally the cache policy for this request.

The `NSURLConnection` object is used to perform loading of the `NSURLRequest`. The object can be invoked either synchronously or asynchronously, but the Apple developer guidelines very strongly discourage the use of synchronous connections, because your application thread will block until the response is received. This means that the complete user interface is blocked; if the connection cannot be established, your application thread only comes back alive after the timeout period has expired, which can be up to 900 seconds.

## Requesting a Website

In the first example you learn how to work with the `NSURLRequest` and `NSURLConnection` objects and the implementation of their delegates.

Start Xcode and create a new project using the Single View Application Project template, and name it `SimpleHTTPCall` using the options shown in Figure 7-1.

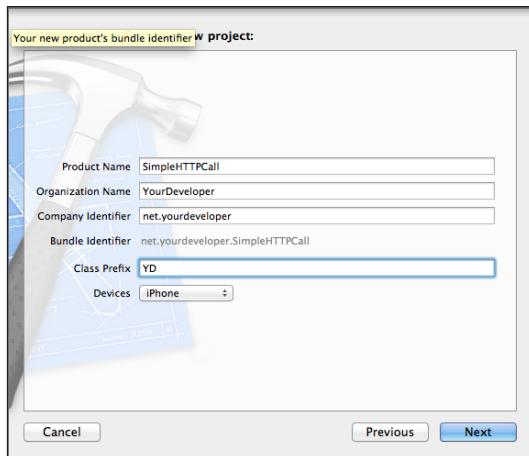


FIGURE 7-1

Open the `ViewController.xib` file using Interface Builder and create a user interface as shown in Figure 7-2. Use the Assistant Editor to create a weak property for the `UITextView` and a method declaration for the `UIButton` action.

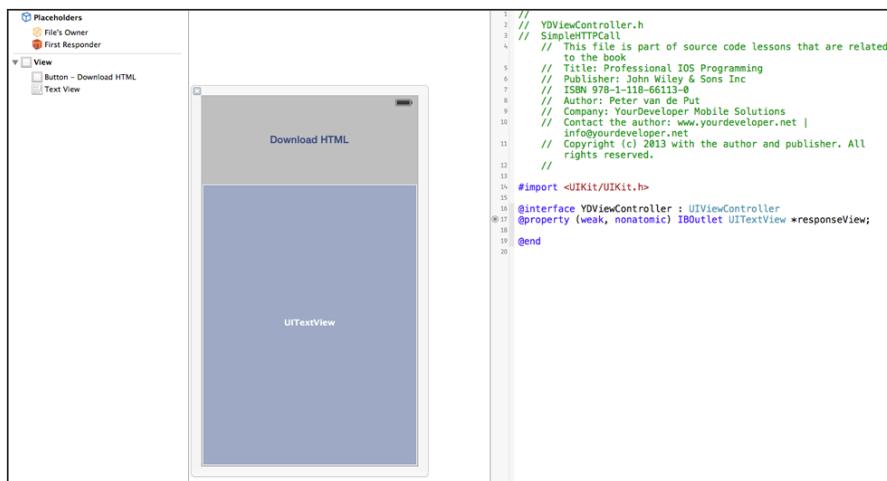


FIGURE 7-2

The `YDViewController.h` file is shown in Listing 7-1.

**LISTING 7-1:** Chapter7/SimpleHTTPCall/ViewController.h

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

@interface YDViewController : UIViewController
@property (weak, nonatomic) IBOutlet UITextView *responseView;

@end
```

Now open the `ViewController.m` file and start with the creation of three private instance variables. Create an `NSURLConnection` and an `NSMutableData` variable. In the `downloadHTML:` method, create an `NSURL` object from a string (in this case, the Apple website), set the HTTP method on the request, and initialize the `NSURLConnection` object. The delegate methods implemented will append the received data to the `webData` variable, and once the connection has finished loading the data, the response is displayed in the `responseView`. The complete code is shown in Listing 7-2.

**LISTING 7-2:** Chapter7/SimpleHTTPCall/ViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()
{
    NSURLConnection* connection;
    NSMutableData* webData;
}

- (IBAction)downloadHTML:(UIButton *)sender;

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)downloadHTML:(UIButton *)sender
{
    self.responseView.text=@"";

    //create a NSURL object with the string using the HTTP protocol
    NSURL *url = [NSURL URLWithString:@"http://www.apple.com"];
    //Create a NSURLRequest from the URL
    NSMutableURLRequest *theRequest = [NSMutableURLRequest requestWithURL:url];
    //it's not required to set the HTTP method since
```

```

        //if not set it will default to GET
        [theRequest setHTTPMethod:@"GET"];
        connection = [[NSURLConnection alloc]
                      initWithRequest:theRequest delegate:self];
        if( connection )
        {
            webData = [[NSMutableData alloc] init];
        }

    }
#pragma mark delegates
-(void) connection:(NSURLConnection *) connection
didReceiveResponse:(NSURLResponse *) response
{
    [webData setLength: 0];
}
-(void) connection:(NSURLConnection *) connection didReceiveData:(NSData *)data
{
    [webData appendData:data];
}
-(void) connection:(NSURLConnection *) connection didFailWithError:(NSError *)error
{
    UIAlertView *alert = [[UIAlertView alloc]
                          initWithTitle:@"Error" message:@"Can't make a connection."
                          delegate:nil cancelButtonTitle:@"Ok"
                          otherButtonTitles:nil, nil];
    [alert show];
}
-(void) connectionDidFinishLoading:(NSURLConnection *) connection
{
    NSString *responseString = [[NSString alloc] initWithBytes:
                                [webData mutableBytes] length:[webData length]
                                encoding:NSUTF8StringEncoding];
    self.responseView.text=responseString;
}

@end

```

## Downloading an Image from an HTTP URL

In some cases you don't want to request a complete website page, but only to download an image from a URL.

In this example you learn a different way of setting up an `NSURLConnection` and downloading the response directly to an `NSOutputStream`.

An `NSOutputStream` is a subclass of the `NSStream` object and provides a write-only stream. To use an `NSOutputStream` to write the receiving bytes that come in as a serial sequence, follow these steps:

1. Create and initialize an instance of `NSOutputStream`.
2. Schedule the stream object and open the stream.

3. Handle the events that the `NSOutputStream` reports to its delegate.
4. When all data is written, dispose of the object.

Start Xcode, create a new project using the Single View Application Project template, and name it `FotoDownloader` using the options shown in Figure 7-3.

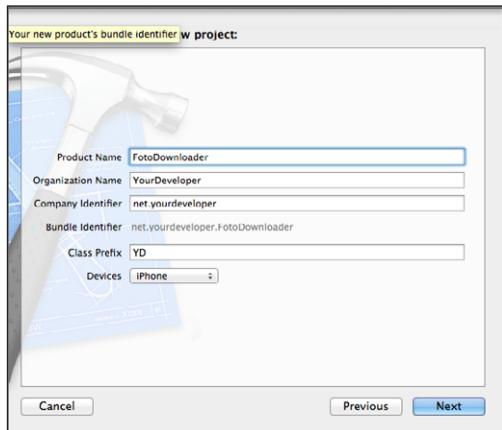


FIGURE 7-3

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor to create a user interface as shown in Figure 7-4.

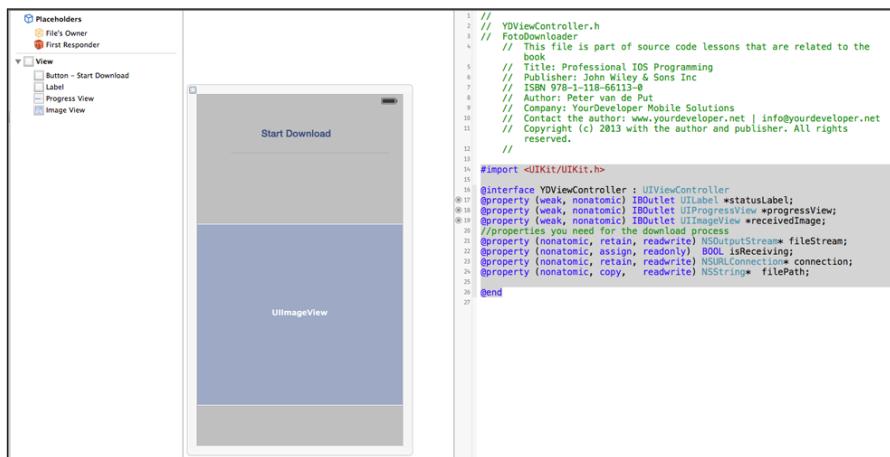


FIGURE 7-4

Open the `ViewController.h` file and add the properties you need for the download process as shown in Listing 7-3.

**LISTING 7-3:** Chapter7/FotoDownloader/ViewController.h

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

@interface YDViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *statusLabel;
@property (weak, nonatomic) IBOutlet UIProgressView *progressView;
@property (weak, nonatomic) IBOutlet UIImageView *receivedImage;
//properties you need for the download process
@property (nonatomic, retain, readonly) NSOutputStream* fileStream;
@property (nonatomic, assign, readonly) BOOL isReceiving;
@property (nonatomic, retain, readonly) NSURLConnection* connection;
@property (nonatomic, copy, readonly) NSString* filePath;

@end
```

Open your ViewController.m file and start by creating two instance variables of type NSInteger named fileSize and bytesDownloaded.

In the startDownload: method you create an NSURL to an image on the Internet. You create an NSOutputStream using the outputStreamToFileAtPath:append: method, passing a generated file-name. Next you create an NSURLRequest, and create an NSURLConnection using the connectionWithRequest: delegate: method of the NSURLConnection.

The delegate connection:didReceiveResponse: method is used to perform a test on the image types you allow to download, and the content length is taken from the HTTP header fields and stored in the fileSize variable.

The connection:didReceiveData:delegate: method is responsible for accepting the data from the stream and writing it directly to the NSOutputStream you are creating, and updating the bytesDownloaded variable that is used to update the UIProgressView. Once the download has finished, the downloaded file is hooked up with the UIImageView and displayed. The complete implementation is shown in Listing 7-4.

**LISTING 7-4:** Chapter7/FotoDownloader/ViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()
{
    NSInteger fileSize;
    NSInteger bytesDownloaded;
}
- (IBAction)startDownload:(UIButton *)sender;

@end

@implementation YDViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
```

*continues*

**LISTING 7-4 (continued)**

```

        // Do any additional setup after loading the view, typically from a nib.
    }
- (IBAction)startDownload:(UIButton *)sender
{
    BOOL             success;
    NSURL *          url;
    NSURLRequest *   request;
    url = [NSURL
URLWithString:@"http://developer.yourdeveloper.net/Images/YDLOGO.png"];
    success = (url != nil);
    // Open a stream for the file we're going to receive into.
    self.filePath = [self createFileName];
    //create the output stream
    self.fileStream = [NSOutputStream outputStreamToFileAtPath:self.filePath
append:NO];
    //open the stream
    [self.fileStream open];
    // create the request
    request = [NSURLRequest requestWithURL:url];
    //create the connection with the request
    self.connection = [NSURLConnection connectionWithRequest:request
delegate:self];
    //clear the image
    self.receivedImage.image = nil;
    self.progressView.progress =0;
}

- (void)stopReceiveWithStatus:(NSString *)statusString
// Shuts down the connection and displays the result
{
    if (self.connection != nil) {
        [self.connection cancel];
        self.connection = nil;
    }
    if (self.fileStream != nil) {
        [self.fileStream close];
        self.fileStream = nil;
    }
    self.statusLabel.text=statusString;
    self.receivedImage.image = [UIImage imageWithContentsOfFile:self.filePath];
    self.filePath = nil;
}
- (NSString*) createFileName {
    //create a file name based on timestamp
    NSDateFormatter *formatter =[[NSDateFormatter alloc] init];
    [formatter setDateFormat:@"ddmmyyyy-HHmmssSSS"];
    return[NSTemporaryDirectory() stringByAppendingPathComponent:
    [NSString stringWithFormat:@"%@.png",
    [formatter stringFromDate:[NSDate date]]]];
}
- (void)connection:(NSURLConnection *)conn

```

```

        didReceiveResponse:(NSURLResponse *)response
    {
        //create a static NSSet with mime types your download will support
        static NSSet * sSupportedImageTypes;
        NSHTTPURLResponse * httpResponse;
        if (sSupportedImageTypes == nil) {
            sSupportedImageTypes = [[NSSet alloc] initWithObjects:@"image/jpeg",
@"image/png", @"image/gif", nil];
        }
        httpResponse = (NSHTTPURLResponse *) response;
        //read the content length from the header field
        fileSize = [[[httpResponse allHeaderFields] valueForKey:@"Content-Length"]
integerValue];
        bytesDownloaded=0;
        //check the status code
        if (httpResponse.statusCode !=200) {
            NSLog(@"error: %@", [NSString stringWithFormat:@"HTTP error %zd", (ssize_t)
httpResponse.statusCode]);
        } else {
            NSString * fileMIMEType;
            fileMIMEType = [[httpResponse MIMEType] lowercaseString];
            if (fileMIMEType == nil) {
                [self stopReceiveWithStatus:@"No Content-Type!"];
            } else if ( ! [sSupportedImageTypes containsObject:fileMIMEType] ) {
                [self stopReceiveWithStatus:[NSString
stringWithFormat:@"Unsupported Content-Type (%@)", fileMIMEType]];
            } else {
                selfStatusLabel.text=@"Response OK";
            }
        }
    }

- (void)connection:(NSURLConnection *)conn didReceiveData:(NSData *)data
// delegate called by the NSURLConnection as data arrives.
// just write the data to the file.
{
    //you need some variable to keep track on where you are writing bytes
    // coming in over the data stream
    NSInteger dataLength;
    const uint8_t * dataBytes;
    NSInteger bytesWritten;
    NSInteger bytesWrittenSoFar;
    dataLength = [data length];
    dataBytes = [data bytes];
    bytesWrittenSoFar = 0;

    do {
        bytesWritten = [self.fileStream write:&dataBytes[bytesWrittenSoFar]
maxLength:dataLength - bytesWrittenSoFar];
        if (bytesWritten == -1) {
            [self stopReceiveWithStatus:@"File write error"];
            break;
        } else {
            bytesWrittenSoFar += bytesWritten;
        }
    }
}

```

*continues*

**LISTING 7-4 (continued)**

```
bytesDownloaded+=bytesWritten;
self.progressView.progress = ((float)bytesDownloaded / (float)fileSize) ;

    } while (bytesWrittenSoFar != dataLength);
}

- (void)connection:(NSURLConnection *)conn didFailWithError:(NSError *)error
{
    [self stopReceiveWithStatus:@"Connection failed"];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)conn
{
    [self stopReceiveWithStatus:@"download has finished"];
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
}

@end
```

The result is shown in Figure 7-5.

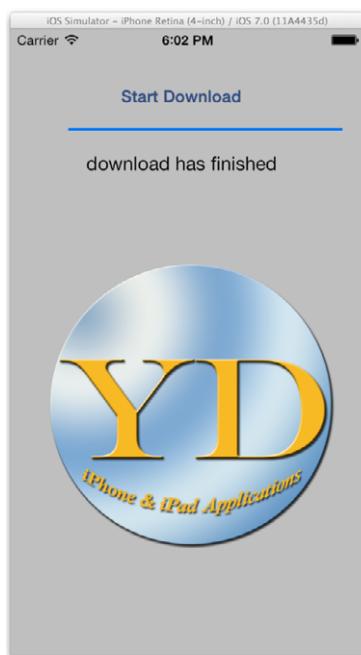


FIGURE 7-5

In these examples you used the `GET` operation for the HTTP protocol, which is the most commonly used operation. `POST` operations, however, are also possible, but much more complex when using the preceding techniques. For that reason, REST and SOAP services come into play. Later in this chapter you use these for not only retrieving data from a web service, but also for posting data to a web service.

## Requesting a Secure Website Using HTTPS

During the exchange of data using the HTTP protocol, all data sent to the server and all data coming in from the request are sent as plain text. In many cases this is not an issue, but if the website you are contacting contains “critical” information that needs to be secured, the web master will most likely run the website as a secure website using the HTTPS protocol. To run a website or web service with the HTTPS protocol, it is required that a certificate is installed on the web server to verify the authenticity of the website. Further, the website will, by default, be a server on port 443 instead of port 80.

The website `http://www.cacert.org` belongs to an organization that provides security certificates for websites, and you can use this website for your HTTPS development and testing purposes by calling the following URL: `https://www.cacert.org`.

Make a copy of the `FotoDownloader` project and rename the project `HTTPSPictureDownloader`.

This new application is aiming for the same—downloading a photo over an Internet connection. Instead of an HTTP protocol, it’s now using an HTTPS protocol. Copy all code from the `FotoDownloader` project and the URL for the downloaded image to the following URL: `https://www.cacert.org/images/cacert4.png`.

When you launch your application, you’ll notice that the status label informs you that the connection failed. This is because you are trying to download the image from the secured URL using the HTTPS protocol. Because the HTTPS protocol is a security protocol, you need to import the Security framework into your project.

Open your `YDViewController.m` file and add a private ivar of type `BOOL` named `shouldAllowSelfSignedCert`.

In the `startDownload:` method, change the URL to `https://www.cacert.org/images/cacert4.png`. In the next line, set the ivar `shouldAllowSelfSignedCert` to true, indicating you trust self-signed certificates. These are certificates not issued by an official organization. The complete `startDownload:` method should be as shown in the following code.

```
- (IBAction)startDownload:(UIButton *)sender
{
    BOOL           success;
    NSURL *        url;
    NSURLRequest * request;
    url = [NSURL URLWithString:@"https://www.cacert.org/images/cacert4.png"];
    shouldAllowSelfSignedCert = YES;
    success = (url != nil);
    // Open a stream for the file we're going to receive into.
    self.filePath = [self createFileName];
    //create the output stream
    self.fileStream = [NSOutputStream outputStreamToFileAtPath:self.filePath
append:NO];
    //open the stream
```

```

    [self.fileStream open];
    // create the request
    request = [NSURLRequest requestWithURL:url];
    //create the connection with the request
    self.connection = [NSURLConnection connectionWithRequest:request
        delegate:self];
    //clear the image
    self.receivedImage.image = nil;
    self.progressView.progress =0;

}

```

In your ViewController.m file, implement the following methods:

```

#pragma mark Challenge delegates

- (BOOL)connection:(NSURLConnection *)conn
    canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)protectionSpace
// This delegate method called by the NSURLConnection when something
//happens with the security-wise.
{
    if([[protectionSpace authenticationMethod]
        isEqualToString:NSURLAuthenticationMethodServerTrust]) {
        if(shouldAllowSelfSignedCert) {
            return YES; // Self-signed cert will be accepted
        } else {
            return NO; // Self-signed cert will be rejected
        }
    }
    // If no other authentication is required, return NO
    return NO;
}

static NSMutableDictionary * sSiteToCertificateMap;
static SecCertificateRef SecTrustGetLeafCertificate(SecTrustRef trust)
// Returns the leaf certificate from a SecTrust object (that is always the
// certificate at index 0).
{
    SecCertificateRef result;
    if (SecTrustGetCertificateCount(trust) > 0) {
        result = SecTrustGetCertificateAtIndex(trust, 0);
    } else {
        result = NULL;
    }
    return result;
}
- (void)connection:(NSURLConnection *)conn
    didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
// This delegate method called by the NSURLConnection when you accept a specific
// authentication challenge by returning YES from -//
    connection:canAuthenticateAgainstProtectionSpace:.

{
    NSURLProtectionSpace *protectionSpace = [challenge protectionSpace];
    SecTrustRef trust = [protectionSpace serverTrust];
    NSString * host;
    SecCertificateRef serverCert;
    OSStatus err;
}

```

```

BOOL                  trusted;
SecTrustResultType   trustResult;
CFRetain(trust);    // Make sure it's retained
//create the credential from the trust
host = [[challenge protectionSpace] host];
if ( [sSiteToCertificateMap objectForKey:host] == nil ) {
    //get the certificate
    serverCert = SecTrustGetLeafCertificate(trust);
    if (serverCert != NULL) {
        //add to array
        [sSiteToCertificateMap setObject:(__bridge id)serverCert forKey:host];
    }
}
//evaluat the certificate
err = SecTrustEvaluate(trust, &trustResult);
trusted = (err == noErr) && ((trustResult == kSecTrustResultProceed) ||
(trustResult == kSecTrustResultUnspecified));
if (!trusted)
{
    NSLog(@"%@", @"Not trusted");
    //check if you trust the domain.
    if ([[challenge protectionSpace host
isEqualToString:@"www.cacert.org"] && shouldAllowSelfSignedCert)
    {
        NSURLCredential *credential =
        [NSURLCredential
        credentialForTrust:challenge.protectionSpace.serverTrust];
        [challenge.sender useCredential:credential
        forAuthenticationChallenge:challenge];
        //continue with the credential from the trust
        [challenge.sender continueWithoutCredentialForAuthenticationChallenge:
challenge];
    }
    else
    {
        NSLog(@"%@", @"trusted so accept the certificate");
        NSURLCredential *credential =
        [NSURLCredential
        credentialForTrust:challenge.protectionSpace.serverTrust];
        [challenge.sender useCredential:credential
        forAuthenticationChallenge:challenge];
        //continue with the credential from the trust
        [challenge.sender
        continueWithoutCredentialForAuthenticationChallenge:challenge];
    }
}
}

```

The connection: `canAuthenticateAgainstProtectionSpace:` method returns a BOOL value. If the `authenticationMethod` property of the `NSURLPrototectionSpace` class returns `NSURLAuthenticationMethodServerTrust` indicating the `NSURLConnection` needs to authenticate against a server certificate, the return value of this method will be true.

The connection: `didReceiveAuthenticationChallenge:` method is called when you accept the authentication challenge, returning YES and an authentication challenge.

It tries to read a certificate from the `protectionSpace` and checks if the certificate is trusted or if you allow self-signed certificates. When the evaluation is true, meaning you accept the connection, the download will take place as in the previous example.

You can find the complete implementation of the `YDViewController.m` file on the website in the download code for this chapter.

## Using Blocks

When using an `NSURLConnection`, you can start the processing synchronously and asynchronously as mentioned before. Apple strongly suggests using only asynchronous `NSURLConnections` to avoid blocking of the thread. If you call an `NSURLConnection` synchronously and the server is not responding, your application would be waiting for a timeout to appear. In a different iOS version, the defaults for these timeouts vary, and despite the information available explaining how to configure the timeout period, experience will tell you it's not a configuration you can rely on.

Most of the time you want to send a request to an `NSURLConnection` and process the result without blocking the application and/or user interface thread. Of course, you can program your code as you've done before by capturing the response in the `connectionDidFinishLoading:` method of the `NSURLConnection` class, and processing it from there, but you can also use blocks to create a `URLRequest` with a completion handler.

Start Xcode, create a new project using the Single View Application Project template, and name it `WebsiteUsingBlocks` using the options shown in Figure 7-6.

Blocks are an advanced concept which you should already be familiar with. It is not within the scope of this book to explain blocks in detail but more information can be found here  
[https://developer.apple.com/library/ios/DOCUMENTATION/Cocoa/Conceptual/Blocks/Articles/00\\_Introduction.html](https://developer.apple.com/library/ios/DOCUMENTATION/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html)

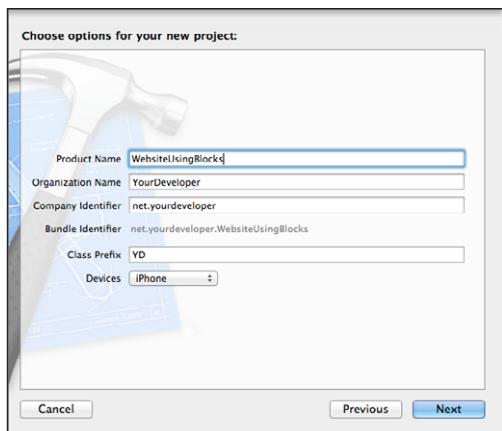


FIGURE 7-6

Start by creating an Objective-C class named `URLRequest` that inherits from `NSObject`. You need to create two methods: the `initWithRequest:` method used to initialize the object and the `startWithCompletion:` method that contains a block you want to implement. The declaration of the header file is shown in Listing 7-5.

**LISTING 7-5:** Chapter7/WebsiteUsingBlocks/URLRequest.h

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

@interface URLRequest : NSObject

- (id)initWithRequest:(NSURLRequest*)req;
- (void)startWithCompletion:(void (^)(URLRequest* request, NSData* data,
    BOOL success))compl;
@end
```

In the URLRequest implementation file, implement the code as shown in Listing 7-6. The initWithRequest: method simply assigns the passed request to the internal request variable, and the startWithCompletion: method creates the completion block handler, starts the connections with the passed request, and manages the completion block. The startWithCompletion: method has a special feature that tests whether or not ARC is used and applies the logic accordingly. Also, the dealloc method has been made conditionally so the URLRequest class will work both with and without ARC without causing memory leaks.

**LISTING 7-6:** Chapter7/WebsiteUsingBlocks/URLRequest.m

```
#import "URLRequest.h"
@interface URLRequest ()
{
    NSURLRequest *request;
    NSURLConnection *connection;
    NSMutableData *webData;
    int httpStatusCode;
    void (^completion)(URLRequest* request, NSData* data, BOOL success);
}
@end

@implementation URLRequest

- (id)initWithRequest:(NSURLRequest*)req
{
    self = [super init];
    if (self != nil)
    {
        request = req;
    }
    return self;
}

- (void)startWithCompletion:(void (^)(URLRequest* request, NSData* data, BOOL
    success))compl
{
    completion = [compl copy];
    connection = [[NSURLConnection alloc] initWithRequest:request delegate:self];
    if (connection)
```

*continues*

**LISTING 7-6 (continued)**

```

{
    #if ! __has_feature(objc_arc)
        webData = [[NSMutableData data] retain];
    #else
        webData = [[NSMutableData alloc] init];
    #endif

}

else
{
    completion(self, nil, NO);
}

}

#endif

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    completion(self, webData, httpStatusCode == 200);
}

- (void)connection:(NSURLConnection *)connection didReceiveResponse:
    (NSURLResponse *)response
{
    NSHTTPURLResponse* httpResponse = (NSHTTPURLResponse*)response;
    httpStatusCode = [httpResponse statusCode];
    [webData setLength: 0];
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    [webData appendData:data];
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    completion(self, webData, NO);
}

@end

```

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor to create a user interface with a `UIImageView` and create the weak property reference as shown in Listing 7-7.

**LISTING 7-7:** Chapter7/WebsiteUsingBlocks/YDViewController.h

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

@interface YDViewController : UIViewController

@property (nonatomic,weak) IBOutlet UIImageView* logoImage;
@end
```

In the `YDViewController.m` file, import the `URLRequest.h` file and create a request to retrieve an image from a URL in the `viewDidLoad` method using the newly created `URLRequest` class, as shown in Listing 7-8.

**LISTING 7-8:** Chapter7/WebsiteUsingBlocks/YDViewController.m

```
#import "YDViewController.h"
#import "URLRequest.h"
@interface YDViewController ()
```

```
@end
```

```
@implementation YDViewController
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSURL *myURL = [NSURL URLWithString:
                     @"http://developer.yourdeveloper.net/Images/YDLOGO.png"];
    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:myURL
cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
           timeoutInterval:60];
    URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
    [urlRequest startWithCompletion:^(URLRequest *request,
                                     NSData *data, BOOL success) {
        if (success)
        {
            self.logoImage.image = [UIImage imageWithData:data];
        }
        else
        {
            NSLog(@"error %@", [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding]);
        }
    }];
}
```

```
- (void)didReceiveMemoryWarning
{
```

*continues*

**LISTING 7-8 (continued)**

```
[super didReceiveMemoryWarning];
// Dispose of any resources that can be recreated.
}

@end
```

## CALLING A REST SERVICE

*Representational State Transfer* (REST) services were introduced several years ago to achieve communication between a client application, like an iOS application, and a back-end application using understandable operations for the developer. A REST service, sometimes referred to as a CRUD service, allows the client application to send standardized requests to the web application to perform operations on data objects. CRUD is an abbreviation for Create, Read, Update, and Delete operations, which typically map to SQL operations as Insert, Select, Update, and Delete.

The main objective of a RESTful service is to achieve a generality of the interface so the service can be used by any type of client developed in any kind of programming language.

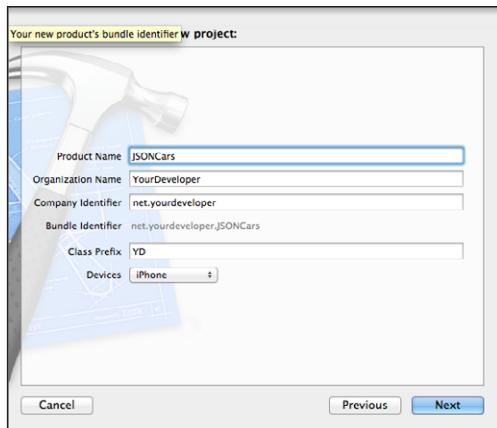
Many iOS applications require a back-end system in which the content is managed by an administrator or user. If your iOS application needs to communicate with your back-end system, you can extend the back-end system with a RESTful service that enables you to simplify the integration of data objects in your applications in a secure way. Another methodology is using SOAP requests, which are explained later in this chapter under the heading “Making SOAP Requests.” Note, however, that developers are losing interest in using SOAP.

To support the code samples in this chapter, I’ve developed a simple back-end system using Microsoft MVC 4 technology that contains some public data sets that are available using the RESTful service technology. The main URL for this back-end system is <http://developer.yourdeveloper.net>, and you can access it without restrictions. One of the advantages of using the MVC 4 technology is that the service can return its data set in either XML or JSON format, which gives you the option to use the data format and parser you prefer to work with. JSON, (JavaScript Object Notation is a lightweight data-interchange format, which is easy to read for humans and it is easy for programming languages to parse and generate. More information about JSON can be found on <http://json.org>.

One of the *Application Programming Interfaces* (APIs) that I’ve implemented returns a data set containing all the car brands that are available. For each brand, you have a property with the brand and one with a URL to the brand logo.

## Constructing Your Request

Start Xcode, create a new project using the Single View Application Project template, and name it `JSONCars` using the options shown in Figure 7-7.



**FIGURE 7-7**

In this project you call a RESTful service and process the JSON response to load a UITableView with a brand name for all car brands in the world.

Open your YDViewController.xib file using Interface Builder and create a user interface with a simple UITableView. Use the Assistant Editor to create a weak property reference to the UITableView named mTableView. Next, create a strong property of type NSMutableArray named cars to store the received cars information from the JSON response.

Make sure your YDViewController.h file looks like Listing 7-9.

**LISTING 7-9: Chapter7/JSONCars/YDViewController.h**

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

@interface YDViewController : UIViewController
@property(nonatomic,weak) IBOutlet UITableView* mTableView;
@property(nonatomic,strong) NSMutableArray* cars;
@end
```

You'll be using the `URLRequest` class you created earlier in the `YDViewController.m` file, so before you import this class, copy it from your previous project into this project.

In the `viewDidLoad` method the RESTful service is called using the `URLRequest` class you created earlier. In the `if (success)` part of the implementation, the `self.cars` array is initialized with the values from the `JSONObjectWithData:options:error:` method of the `NSJSONSerialization` class. In the `tableView:cellForRowAtIndexPath:` method, an object is retrieved from the `self.cars` array and created as an `NSDictionary` named `car`. The `objectForKey:` method of the `NSDictionary` class is used to access the object and display it on the UITableView.

The complete implementation of `YDViewController.m` file is shown in Listing 7-10.

**LISTING 7-10: Chapter7/JSONCars/YDViewController.m**

```
#import "YDViewController.h"
#import "NSURLRequest.h"

@interface YDViewController ()<UITableViewDataSource, UITableViewDelegate>

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    NSURL *myURL = [NSURL
URLWithString:@"http://developer.yourdeveloper.net/api/car"];
    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:myURL

cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
           timeoutInterval:60];
    //import to the Content-Type to application/json to
    //receive JSON format response
    [request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
    URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
    [urlRequest startWithCompletion:^(URLRequest *request,
    NSData *data, BOOL success) {
        if (success)
        {
            NSError* error=nil;
            self.cars = [NSJSONSerialization JSONObjectWithData:data
options:kNilOptions error:&error];

            [self.tableView reloadData];
        }
        else
        {
            NSLog(@"error %@", [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding]);
        }
    }];
}

#pragma UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section {
    return [self.cars count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
```

```
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = (UITableViewCell *)[tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    NSDictionary* car = [self.cars objectAtIndex:indexPath.row];
    cell.selectionStyle = UITableViewCellStyleNone;
    cell.textLabel.text=[car objectForKey:@"brand"];
    return cell;
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Figure 7-8 shows the result when you launch the application.

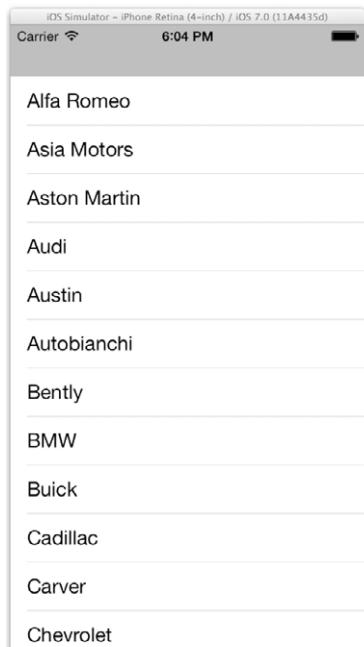


FIGURE 7-8

## Processing the Response

The response you received was a JSON response because you set the Content-Type to application/json. JSON stands for *JavaScript Object Notation* and is a commonly used syntax to exchange information between systems. The structural characters used in the JSON notation are brackets, colons, and commas ([{}], :, and ,).

### Processing the JSON Response

Since iOS version 5, Cocoa has a built-in JSON library. To work with JSON you need to understand that the data coming in from a stream needs to be serialized before it can be used for processing. The most convenient object to store a full JSON result is an NSArray. The response you receive in this example has the following structure:

```
{
    {
        brand = "Alfa Romeo";
        carID = 1;
        logoURL =
        "http://developer.yourdeveloper.net/ImageData/cars/alfa-romeo-logo-small.gif";
    },
    {
        brand = "Asia Motors";
        carID = 2;
        logoURL =
        "http://developer.yourdeveloper.net/ImageData/cars/asia-motors-logo-small.gif";
    }
}
```

When you receive this data and serialize it into an NSArray with the following function:

```
NSError* error=nil;
Self.cars = [NSJSONSerialization JSONObjectWithData:data
options:kNilOptions error:&error];
```

your self.cars array, which is defined as an NSMutableArray, will contain an object for each individual data object. The data object here contains three elements: brand, carID, and logoURL. You can access the details from your array by retrieving the required objectAtIndex and storing it in an NSDictionary like you do in the tableView:cellForRowAtIndexPath: method:

```
NSDictionary* car = [self.cars objectAtIndex:indexPath.row];
```

You now have an NSDictionary named car from which you can access the individual properties using the [car objectForKey:name] method, where name is equal to the name you received in the JSON output. So in this case, [car objectForKey@"logoURL"] will return a value like http://developer.yourdeveloper.net/ImageData/cars/alfa-romeo-logo-small.gif.

## Processing an XML Response

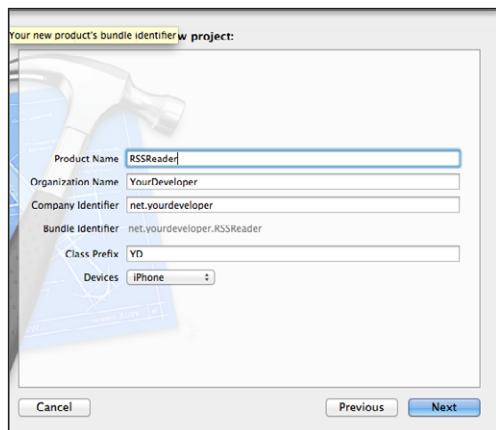
Another format you can use to exchange data between two systems is XML. XML stands for *eXtensible Markup Language* and is a structure that is also widely used. Which flavor, JSON or XML, you prefer to work with is up to you. Often, if your clients have a back-end system you need to integrate in your application, you have to respect their choice and adapt to it.

When I started developing iPhone applications I preferred to work with XML formats because they seemed to be more organized and structured, and the rules for valid XML were more restricted. Last year I worked on several projects where the client had a back-end system providing XML data streams that weren't consistent, and I was forced to write a lot of additional checks to process these XML streams. At that moment I decided to switch over to JSON where possible.

In comparison to XML parsing, JSON parsing is much easier to implement with much less code.

You can parse XML in many different ways. In the next example you learn the standard Cocoa way, but you can make use of external libraries as well.

To demonstrate how to parse an XML response, start Xcode, create a new project using the Single View Application Project template, and name it `RSSReader` using the options shown in Figure 7-9.



**FIGURE 7-9**

In this example you create an `RSSReader` that displays the RSS feed of the *New York Times* in a `UITableView`.

Open the `YDViewController.xib` file using Interface Builder and create a simple user interface with a `UITableView` object. Use the Assistant Editor to create the weak property reference.

Open the `YDViewController.h` file and create the `parseXMLFileAtURL:` method declaration. Create a strong property of type `NSMutableArray` named `feeds` that will be used to store the results from the parsing, as shown in Listing 7-11.

The `NSXMLParser` object is the standard Cocoa XML parser you'll be using in this project.

**LISTING 7-11:** Chapter7/RSSReader/YDViewController.h

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

@interface YDViewController : UIViewController
- (void)parseXMLFileAtURL:(NSString *)URL;
@property(nonatomic,weak) IBOutlet UITableView* mTableView;
@property(nonatomic,strong) NSMutableArray* feeds;
@end
```

XML parsing is done by the `NSXMLParser` object and its delegate methods. The `NSXMLParser` is a SAX parser. There is no native DOM XML parser available in the iOS SDK. The implementation of the `parseXMLFileAtURL:` method is using the earlier created `URLRequest` to retrieve the data from the RSS feed's address. You start the parsing by calling the `parse` method of the `NSXMLParser` class. In the implementation I've added inline comments to explain what each of the delegate methods does in the process of parsing. The complete implementation of the `YDViewController.m` file is shown in Listing 7-12.

**LISTING 7-12:** Chapter7/RSSReader/YDViewController.m

```
#import "YDViewController.h"
#import "NSURLRequest.h"
@interface YDViewController ()<UITableViewDataSource, UITableViewDelegate, NSXMLParserDelegate>
{
    NSXMLParser *rssParser;
    NSMutableArray *articles;
    NSMutableDictionary *item;
    NSString *currentElement;
    NSMutableString *ElementValue;
    BOOL errorParsing;
}
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self parseXMLFileAtURL:
     @"http://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml"];
}

#pragma mark XML Parsing

/*
This method is called with a urlstring of the feed that needs to be parsed
*/
- (void)parseXMLFileAtURL:(NSString *)URL
{
```

```

NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:
                               [NSURL URLWithString:URL]];
// NSURL *myURL = [NSURL URLWithString:URL];
URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
[urlRequest startWithCompletion:^(URLRequest *request, NSData *data, BOOL success) {
    if (success)
    {
        //create the article array
        articles = [[NSMutableArray alloc] init];
        errorParsing=NO;
        //create the NSXMLParser and initialize it with the data received
        rssParser = [[NSXMLParser alloc] initWithData:data];
        //set the delegate
        [rssParser setDelegate:self];
        // You may need to turn some of these on depending on the type of XML
        // file you are parsing
        [rssParser setShouldProcessNamespaces:NO];
        [rssParser setShouldReportNamespacePrefixes:NO];
        [rssParser setShouldResolveExternalEntities:NO];
        [rssParser parse];
    }
    else
    {
        NSLog(@"%@", [[NSString alloc] initWithData:data
                                         encoding:NSUTF8StringEncoding]);
    }
}];

}

//this method is called when the parser has found a valid start element
- (void)parserDidStartDocument:(NSXMLParser *)parser
{
}

/*
This method is called if an error occurs
*/
- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError
{
    NSString *errorString = [NSString stringWithFormat:
                            @"Error code %i", [parseError code]];
    NSLog(@"Error parsing XML: %@", errorString);
    errorParsing=YES;
}
/*
The XML parser contains three methods, one that runs at the beginning of an
individual element,
one that runs during the middle of parsing the element,
and one that runs at the end of the element.

```

For this example, we'll be parsing an RSS feed that breaks down elements into groups under the heading of "items".

At the start of the processing, we are checking for the element name "item" and *continues*

**LISTING 7-12 (continued)**

```

allocating our item dictionary when a new group is detected.
Otherwise, we initialize our variable for the value.
*/
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qName
    attributes:(NSDictionary *)attributeDict
{
    currentElement = [elementName copy];
    ElementValue = [[NSMutableString alloc] init];
    if ([elementName isEqualToString:@"item"]) {
        item = [[NSMutableDictionary alloc] init];
    }
}
/*
When the parser find characters, they are added to your variable "ElementValue".
*/
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    [ElementValue appendString:string];
}
/*
If the endelement for the item is found the item is copied and added to the
articles array
Also the tableview is reloaded here
*/
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
{
    if ([elementName isEqualToString:@"item"]) {
        [articles addObject:[item copy]];
        [self.mTableView reloadData];
    } else {
        [item setObject:ElementValue forKey:elementName];
    }
}

#pragma mark UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [articles count];
[AU/ED: Please change the tab(s) in the above line to spaces.]
}
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

```

```
UITableViewCell *cell = (UITableViewCell *)[tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CellIdentifier];
}
//get the item from the articles array as a dictionary
NSDictionary* feedItem = [articles objectAtIndex:indexPath.row];
cell.selectionStyle = UITableViewCellSelectionStyleNone;
cell.textLabel.text=[feedItem objectForKey:@"title"];
return cell;
}

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

@end
```

As you can see, parsing the XML response takes much more coding than parsing a JSON response, where a single line did the job. In the parsing methods you need to code the logic related to the elements that will be available in the XML data.

Running the application produces the result as shown in Figure 7-10.

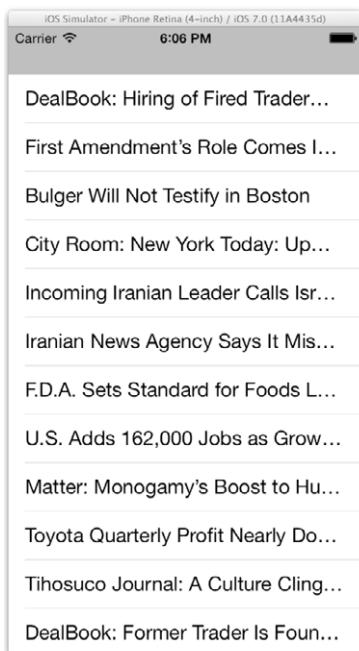


FIGURE 7-10

## Posting to a RESTful Service

When your application needs to integrate with a back-end system using a RESTful service, you not only need to retrieve data from the server, but you also may need to post data to the service. To demonstrate how you can make a post to a RESTful web service, I've created a guestbook API that you can use to send a message to, and also retrieve messages that have been posted by, other users.

Start Xcode, create a new project using the Tabbed Application Project template, and name it Guestbook using the options shown in Figure 7-11.

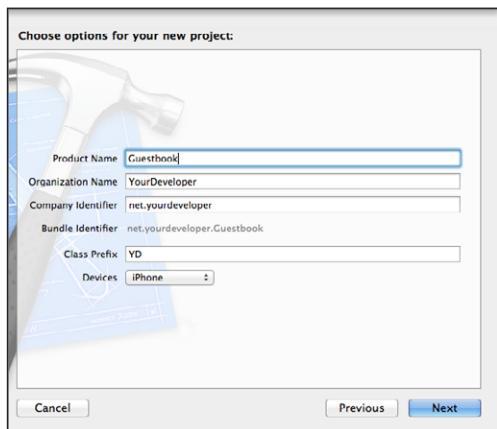


FIGURE 7-11

The generated `YDFirstViewController` is used to display in a `UITableView` the guestbook entries that have been made by sending a `GET` request to the URL. The `YDSecondViewController` is used to post a message to the RESTful API by sending a JSON dictionary to the server to create a new entry.

Copy the `URLRequest` class files from the previous project to your project because you'll be using this class to send the requests using blocks.

Open the `YDFirstViewController.xib` file with Interface Builder and place a `UITableView` on the `View` object to create a user interface as shown in Figure 7-12.

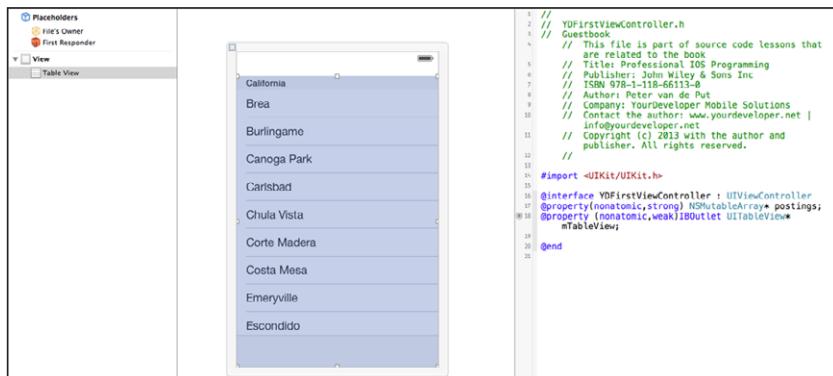


FIGURE 7-12

Open the `YDFirstViewController.h` file and create a strong property of type `NSMutableArray` named `postings` as shown in Listing 7-13.

#### LISTING 7-13: Chapter7/Guestbook/YDFirstViewController.h

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

@interface YDFirstViewController : UIViewController
@property(nonatomic,strong) NSMutableArray* postings;
@property (nonatomic,weak) IBOutlet UITableView* mTableView;
@end
```

Open the `YDFirstViewController.m` file and import the `URLRequest` header. Subscribe to the `UITableViewDataSource` and `UITableViewDelegate` protocols. In the `viewDidLoad` method, call the `loadGuestbook` method that is responsible for creating the GET request and store the incoming data into the `postings` array. The complete implementation is shown in Listing 7-14. You've used all the techniques that were used in this example earlier in this chapter.

#### LISTING 7-14: Chapter7/Guestbook/YDFirstViewController.m

```
#import "YDFirstViewController.h"
#import "URLRequest.h"
@interface YDFirstViewController ()<UITableViewDataSource, UITableViewDelegate>

@end

@implementation YDFirstViewController

- (id)initWithNibName:(NSString *)NibNameOrNil bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNilOrNil bundle:nibBundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"Table", @"Table");
        self.tabBarItem.image = [UIImage imageNamed:@"first"];
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self loadGuestBook];
}
- (void)loadGuestBook
{
    if (self.postings!=nil)
        self.postings=nil;
    NSURL *myURL = [NSURL URLWithString:
    @"http://developer.yourdeveloper.net/api/guestbook"];
```

*continues*

**LISTING 7-14 (continued)**

```

NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:myURL
    cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
    timeoutInterval:60];
//import to the Content-Type to application/json to receive JSON format
// response
[request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
[urlRequest startWithCompletion:^(URLRequest *request, NSData *data, BOOL success) {
    if (success)
    {
        NSError* error=nil;
        self.postings = [NSJSONSerialization JSONObjectWithData:data
            options:kNilOptions error:&error];

        [self.mTableView reloadData];
    }
    else
    {
        NSLog(@"error %@", [[NSString alloc] initWithData:data
            encoding:NSUTF8StringEncoding]);
    }
}];
}
#pragma mark UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [self.postings count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = (UITableViewCell *)[tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    NSDictionary* posting = [self.postings objectAtIndex:indexPath.row];
    cell.selectionStyle = UITableViewCellStyleNone;
    cell.textLabel.text = [NSString stringWithFormat:@"%@ - %@",
        [posting objectForKey:@"guestbookID"],
        [posting objectForKey:@"name"]];
    return cell;
}
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
    (NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}

```

```
NSDictionary* posting = [self.postings objectAtIndex:indexPath.row];
UIAlertView* alert = [[UIAlertView alloc]
    initWithTitle:[posting objectForKey:@"name"]
    message:[posting objectForKey:@"message"]
    delegate:self
    cancelButtonTitle:@"Ok"
    otherButtonTitles:nil, nil];
[alert show];
}
-(void)viewDidAppear:(BOOL)animated
{
    [self loadGuestBook];
}

-(void) didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Open the `YDSecondViewController.xib` file using Interface Builder and the Assistant Editor and create a user interface similar to the one shown in Figure 7-13.

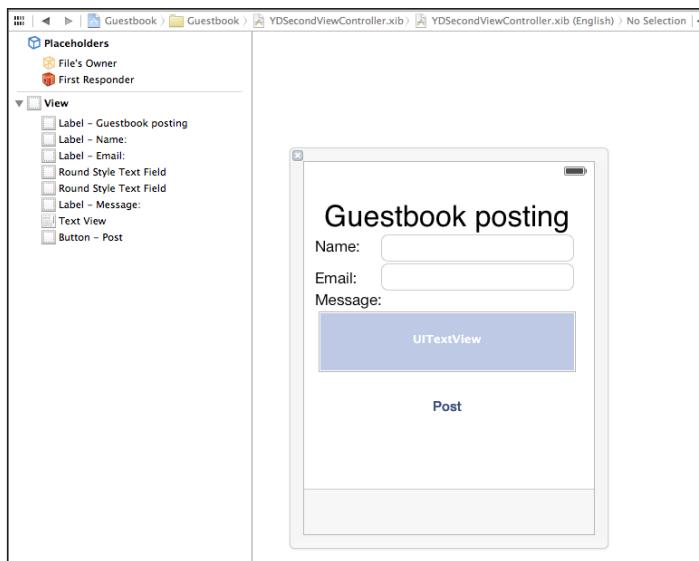


FIGURE 7-13

The complete implementation is shown in Listing 7-15.

**LISTING 7-15:** Chapter7/Guestbook/YDSecondViewController.h

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

@interface YDSecondViewController : UIViewController

@property(nonatomic,weak) IBOutlet UITextField* nameField;
@property(nonatomic,weak) IBOutlet UITextField* emailField;
@property(nonatomic,weak) IBOutlet UITextView* messageField;

@end
```

Open the `YDSecondViewController.m` file and implement the code as shown in Listing 7-16.

**LISTING 7-16:** Chapter7/Guestbook/YDSecondViewController.m

```
#import "YDSecondViewController.h"
#import "NSURLRequest.h"
@interface YDSecondViewController ()
-(IBAction)postMessage:(id)sender;
@end

@implementation YDSecondViewController

- (id)initWithNibName:(NSString *)NibNameOrNil bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:bundleOrNil];
    if (self) {
        self.title = NSLocalizedString(@"Write", @"Write");
        self.tabBarItem.image = [UIImage imageNamed:@"second"];
    }
    return self;
}
- (void)viewDidLoad
{
    [super viewDidLoad];
}
-(IBAction)postMessage:(id)sender
{
    [self.nameField resignFirstResponder];
    [self.emailField resignFirstResponder];
    [self.messageField resignFirstResponder];
    if (([self.nameField.text length]==0) ||
        ([self.emailField.text length]==0) ||
        ([self.messageField.text length]==0))
    {
        UIAlertView* alert = [[UIAlertView alloc]
                               initWithTitle:@"Error"
                               message:@"fill all fields"
                               delegate:self
                               cancelButtonTitle:@"Ok"
                               otherButtonTitles:nil,
                               nil];
    }
}
```

```

        nil];
    [alert show];
}
else{
    //fields are filled so make a post
    NSString *urlToPost = @"http://developer.yourdeveloper.net/api/Guestbook";
    //create a NSDictionary with the fields you need to pass
    NSMutableDictionary *jsonDict = [NSMutableDictionary dictionaryWithDictionary];
    [jsonDict setObject:self.nameField.text forKey:@"name"];
    [jsonDict setObject:self.emailField.text forKey:@"email"];
    [jsonDict setObject:self.messageField.text forKey:@"message"];
    //serialize it as a JSON string
    NSError *error;
    NSData *jsonData = [NSJSONSerialization
                         dataWithJSONObject:jsonDict
                         options:NSJSONWritingPrettyPrinted
                         error:&error];

    if (!jsonData) {
        NSLog(@"Got an error: %@", error);
    } else {
        NSString *jsonString = [[NSString alloc]
                               initWithData:jsonData
                               encoding:NSUTF8StringEncoding];
    }
}

NSURL *myURL = [NSURL URLWithString:urlToPost];
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:myURL
                                                       cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
                                                       timeoutInterval:60];
//http
[request setHTTPMethod:@"POST"];
NSString *msgLength = [NSString stringWithFormat:@"%d",
                      [jsonString length]];
[request addValue: msgLength forHTTPHeaderField:@"Content-Length"];
[request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
[request setHTTPBody: [jsonString dataUsingEncoding:NSUTF8StringEncoding]];

URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
[urlRequest startWithCompletion:^(URLRequest *request, NSData *data,
                                BOOL success) {
    if (success)
    {
        NSError* error=nil;
        NSString *responseString = [NSJSONSerialization
                                    JSONObjectWithData:data
                                    options:kNilOptions
                                    error:&error];
    }
    else
    {
        NSString *responseString= [[NSString alloc]

```

*continues*

**LISTING 7-16** (continued)

```

initWithData:data
encoding:NSUTF8StringEncoding];

}

self.nameField.text=@";
self.emailField.text=@";
self.messageField.text=@";
UIalertView* alert = [[UIAlertView alloc]
    initWithTitle:@"Thanks"
    message:@"for writing in our guestbook"
    delegate:self
    cancelButtonTitle:@"Ok"
    otherButtonTitles:nil, nil];
[alert show];
};

}

}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

All the magic happens in the `postMessage` method, which starts with checking if all fields have been filled and then creates an `NSMutableDictionary` with objects for name, e-mail, and message. This `NSMutableDictionary` is then serialized as JSON data, which is converted to an `NSString` object.

The HTTP method of the `NSMutableURLRequest` is set to POST because you are posting data to the API. Also, the content-length header field is set and the created JSON string is set to the `HTTPbody` property of the `NSMutableURLRequest`. When the request is sent, the server can accept the passed information based on the HTTP header fields and the JSON string passed in the `HTTPbody`, and creates a new entry. If you now switch back to the `YDFFirstViewController` you will see your new post at the bottom of the `UITableView`.

## MAKING SOAP REQUESTS

SOAP, originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services. For more information visit <http://en.wikipedia.org/wiki/SOAP>.

In the SOAP examples in this chapter you can use the special SOAP service I've created for this purpose. This SOAP service is available via the URL <http://developer.yourdeveloper.net/soapservice.asmx>.

When you visit this URL with a web browser, you will see the metadata for this service as shown in Figure 7-14.

The following operations are supported. For a formal definition, please review the [Service Description](#).

- [GetProtectedData](#)  
Returns some protected data
- [HelloWorld](#)
- [MyIPAddress](#)
- [addTwoNumbers](#)

This web service is using <http://tempuri.org/> as its default namespace.

FIGURE 7-14

Four operations are available for this web service. The one you'll be using to start with is the `MyIPAddress` operation, a service that responds with the public IP address of the connection you are using to make this SOAP request.

If you click the hyperlink for the operation you want to perform you will see a page that explains the structure of this SOAP operation. Each SOAP operation provides you with this information to help you understand how to call this SOAP service. Figure 7-15 shows the structure of the request for the `MyIPAddress` that you need to send to the server.

Click [here](#) for a complete list of operations.

### MyIPAddress

**Test**  
The test form is only available for requests from the local machine.

#### SOAP 1.1

The following is a sample SOAP 1.1 request and response. The placeholders shown need to be replaced with actual values.

```

POST /soapservice.asmx HTTP/1.1
Host: developer.youddevt.net
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/MyIPAddress"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
</soap:Body>
</soap:Envelope>

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<MyIPAddressResponse xmlns="http://tempuri.org/">
<MyIPAddressResult>string</MyIPAddressResult>
</MyIPAddressResponse>
</soap:Body>
</soap:Envelope>

```

#### SOAP 1.2

The following is a sample SOAP 1.2 request and response. The placeholders shown need to be replaced with actual values.

```

POST /soapservice.asmx HTTP/1.1
Host: developer.youddevt.net
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2005/05/soap-envelope">
<soap12:Body>
<MyIPAddressResponse xmlns="http://tempuri.org/">
<MyIPAddressResult>string</MyIPAddressResult>
</MyIPAddressResponse>
</soap12:Body>
</soap12:Envelope>

```

HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8

```

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2005/05/soap-envelope">
<soap12:Body>
<MyIPAddressResponse xmlns="http://tempuri.org/">
<MyIPAddressResult>string</MyIPAddressResult>
</MyIPAddressResponse>
</soap12:Body>
</soap12:Envelope>

```

FIGURE 7-15

Start Xcode, create a new project using the Single View Application Project template, and name it SoapRequest using the options shown in Figure 7-16.

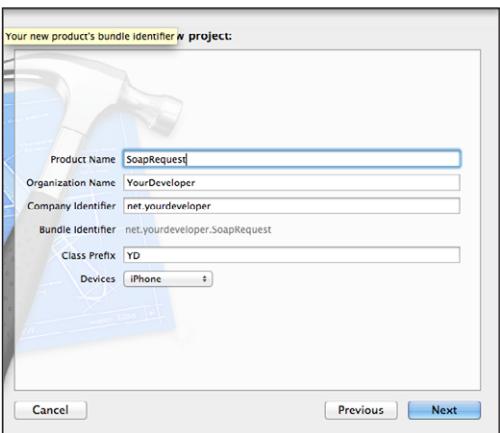


FIGURE 7-16

Start by copying the `URLRequest` class from the previous project to this project so you can again make use of this powerful class.

## Preparing Your Request

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor and create a user interface that looks like the one shown in Figure 7-17.

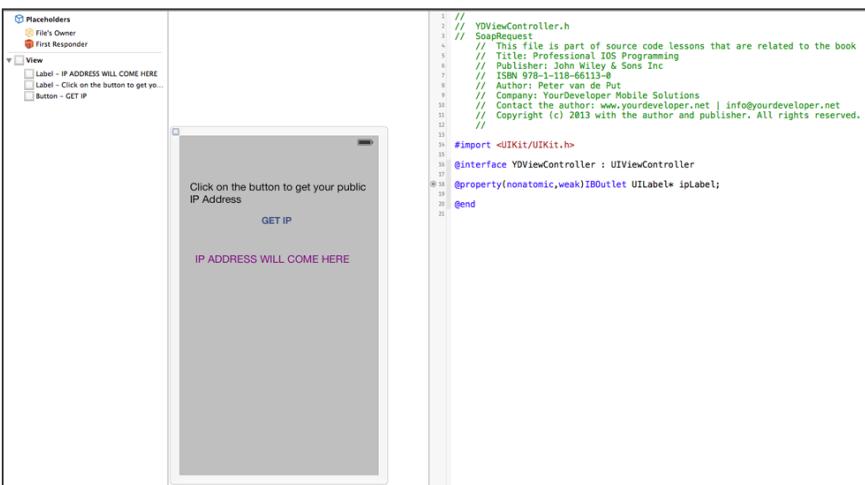


FIGURE 7-17

The complete YDViewController.h file is shown in Listing 7-17.

**LISTING 7-17: Chapter7/SOAPRequest/YDViewController.h**

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

@interface YDViewController : UIViewController
@property(nonatomic,weak) IBOutlet UILabel* ipLabel;
@end
```

Open your YDViewController.m file and write the getIPAddress method that will do most of the work. Start by constructing the SOAP message you need to send to the server. When calling a SOAP service, it's important to specify the SOAPAction in the HTTP header you found when you clicked on the hyperlink of the operation you want to perform. Set the HTTP method to POST and pass the request to the URLRequest object. In the success routine, the received data is passed to the soapParser that will parse the result and set the found IP address to the ipLabel you've created. The construction of the SOAP message is the only difference with the other application you created in this chapter. For that reason you'll find the construction part of the soapMessage in Listing 7-18. The implementation of the NSXMLParser is similar to the RSSReader application. You can find the complete implementation in the project download for this chapter.

**LISTING 7-18: Chapter 7/Construct the soapMessage in the getIPAddress: method**

```
//Construct the soapMessage
NSString *soapMsg = @"<?xml version=\"1.0\" encoding=\"utf-8\"?>
<soap:Envelope
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">
<soap:Body>
<MyIPAddress xmlns=\"http://tempuri.org/\" />
</soap:Body>
</soap:Envelope>";

//Create the NSURL
NSURL *url = [NSURL
URLWithString:@"http://developer.yourdeveloper.net/soapservice.asmx"];
NSMutableURLRequest *theRequest = [NSMutableURLRequest requestWithURL:url];

NSString *msgLength = [NSString stringWithFormat:@"%d", [soapMsg length]];

[theRequest addValue: @"text/xml; charset=utf-8"
forHTTPHeaderField:@"Content-Type"];

//Set the SOAP Action
[theRequest addValue: @"http://tempuri.org/MyIPAddress"
forHTTPHeaderField:@"SOAPAction"];
```

*continues*

**LISTING 7-18 (continued)**

```
[theRequest addValue: msgLength forHTTPHeaderField:@"Content-Length"] ;  
  
//make sure it's a POST  
[theRequest setHTTPMethod:@"POST"] ;  
  
[theRequest setHTTPBody: [soapMsg dataUsingEncoding:NSUTF8StringEncoding]] ;
```

When you launch the application and press the button labeled GET IP, the SOAP message is sent to the SOAP service and the response is parsed. Your user interface is updated as shown in Figure 7-18.

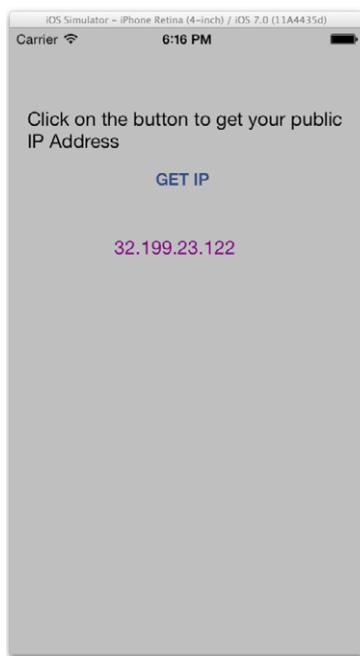


FIGURE 7-18

## Passing Values to an Operation

In many cases you want to pass values to a SOAP operation. To demonstrate how to pass parameters to a SOAP operation, I've created the `addTwoNumbers` operation that simply adds the two passed numbers and returns the result of this addition.

Start Xcode, create a new project using the Single View Application Project template, and name it `SoapCalculator` using the options shown in Figure 7-19.

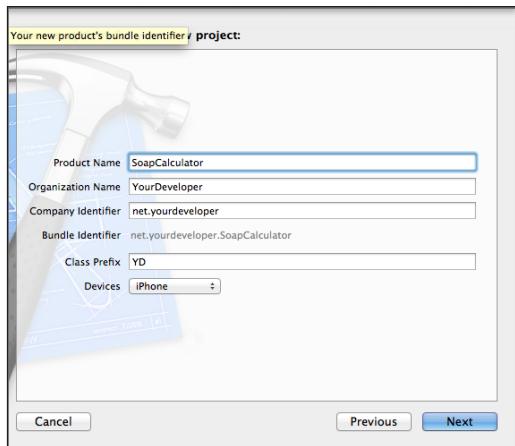


FIGURE 7-19

Start with copying the `URLRequest` class to your project.

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor to create a user interface that looks like the one shown in Figure 7-20.

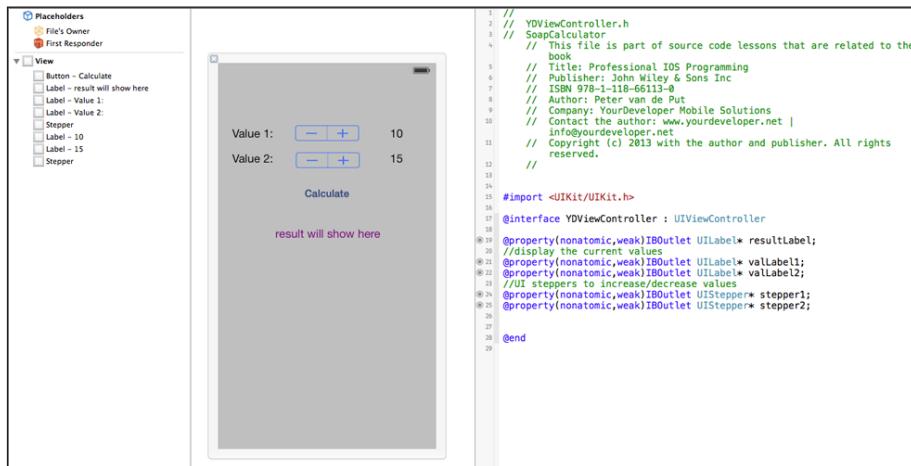


FIGURE 7-20

Open the `YDViewController.h` file to verify the properties are created by the Assistant Editor as shown in Listing 7-19.

**LISTING 7-19:** Chapter7/SoapCalculator/YDViewController.h

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

@interface YDViewController : UIViewController

@property(nonatomic,weak) IBOutlet UILabel* resultLabel;
//display the current values
@property(nonatomic,weak) IBOutlet UILabel* valLabel1;
@property(nonatomic,weak) IBOutlet UILabel* valLabel2;
//UI steppers to increase/decrease values
@property(nonatomic,weak) IBOutlet UIStepper* stepper1;
@property(nonatomic,weak) IBOutlet UIStepper* stepper2;

@end
```

The implementation of the `YDViewController` contains some actions for updating the value label using the `UIStepper` object. This is a simple way to avoid false data entries; for this SOAP operation, you can only pass integer values.

The construction of the `soapMsg` `NSString` object is now done with a `stringWithFormat:` method of the `NSString` class, so the values of the `valLabel1` and `valLabel2` can be passed. Of course, you are now looking for a different element in your `soapParser` so you can copy and paste most of the code, but you need to make changes for this specific operation. The complete implementation is shown in Listing 7-20.

**LISTING 7-20:** Chapter7/SoapCalculator/YDViewController.m

```
#import "YDViewController.h"
#import "NSURLRequest.h"
@interface YDViewController ()<NSXMLParserDelegate>
{
    NSXMLParser *soapParser;
    NSString *currentElement;
    NSMutableString *ElementValue;
    BOOL errorParsing;
    NSMutableDictionary *item;
}
-(IBAction)calculate:(id)sender;
-(IBAction)increaseValue1:(id)sender;
-(IBAction)increaseValue2:(id)sender;
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}
-(IBAction)increaseValue1:(id)sender
{
```

```

        self.valLabel1.text = [NSString stringWithFormat:@"%@", self stepper1.value];
    }
    -(IBAction)increaseValue2:(id)sender
    {
        self.valLabel2.text = [NSString stringWithFormat:@"%@", self stepper2.value];
    }
    -(IBAction)calculate:(id)sender
    {

        //Construct the soapMessasge

        NSString *soapMsg =[NSString stringWithFormat:@"<?xml version=\"1.0\""
                           encoding=\"utf-8\"?>"
                           "<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\""
                           "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\""
                           "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">"
                           "<soap:Body>"
                           "<addTwoNumbers xmlns=\"http://tempuri.org/\">"
                           "<num1>%@</num1>"
                           "<num2>%@</num2>"
                           "</addTwoNumbers>"
                           "</soap:Body>"
                           "</soap:Envelope>",self.valLabel1.text ,self.valLabel2.text ];
        NSLog(@"soap: %@",soapMsg);
        //Create the NSURL
        NSURL *url = [NSURL
URLWithString:@"http://developer.yourdeveloper.net/soapservice.asmx"];
       NSMutableURLRequest *theRequest = [NSMutableURLRequest requestWithURL:url];
        NSString *msgLength = [NSString stringWithFormat:@"%@", [soapMsg length]];
        [theRequest addValue: @"text/xml; charset=utf-8"
forHTTPHeaderField:@"Content-Type"];
        //Set the SOAP Action
        [theRequest addValue: @"http://tempuri.org/addTwoNumbers"
forHTTPHeaderField:@"SOAPAction"];
        [theRequest addValue: msgLength forHTTPHeaderField:@"Content-Length"];
        //make sure it's a POST
        [theRequest setHTTPMethod:@"POST"];
        [theRequest setHTTPBody: [soapMsg dataUsingEncoding:NSUTF8StringEncoding]];

        URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:theRequest];
        [urlRequest startWithCompletion:^(URLRequest *request,
        NSData *data, BOOL success) {
            if (success)
            {
                errorParsing=NO;
                //create the NSXMLParser and initialize it with the data received
                soapParser = [[NSXMLParser alloc] initWithData:data];
                //set the delegate
                [soapParser setDelegate:self];
                // You may need to turn some of these on depending on the type of XML
                //file you are parsing
                [soapParser setShouldProcessNamespaces:NO];
                [soapParser setShouldReportNamespacePrefixes:NO];
                [soapParser setShouldResolveExternalEntities:NO];
                [soapParser parse];
            }
        }];
    }
}

```

*continues*

**LISTING 7-20 (continued)**

```

        }
    else
    {
        NSLog(@"%@", [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding]);
    }
};

}

#pragma marks delegates
- (void)parserDidStartDocument:(NSXMLParser *)parser{
    NSLog(@"data found and parsing started");
}
- (void)parserDidEndDocument:(NSXMLParser *)parser
{
}
/*
This method is called if an error occurs
*/
- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError {
    NSString *errorString = [NSString stringWithFormat:@"Error code %i",
[parseError code]];
    NSLog(@"Error parsing XML: %@", errorString);
    errorParsing=YES;
}
/*
The XML parser contains three methods, one that runs at the beginning of an
individual element,
one that runs during the middle of parsing the element,
and one that runs at the end of the element.

For this example, we'll be parsing an RSS feed that breaks down elements into
groups under the heading of "items".
At the start of the processing, we are checking for the element name "item" and
allocating our item dictionary when a new group is detected.
Otherwise, we initialize our variable for the value.
*/
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)
qName attributes:(NSDictionary *)attributeDict{
    currentElement = [elementName copy];
    ElementValue = [[NSMutableString alloc] init];
    if ([elementName isEqualToString:@"addTwoNumbersResult"]){
        item = [[NSMutableDictionary alloc] init];
    }
}
/*
When the parser finds characters, they are added to your variable "ElementValue".
*/
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string{
    [ElementValue appendString:string];
}

```

```
/*
If the endelement for the item is found the item is copied and added to the
articles array
Also the tableview is reloaded here
*/
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName{
    if ([elementName isEqualToString:@"addTwoNumbersResult"]) {
        self.resultLabel.text =
        [NSString stringWithFormat:@"Result: %@", ElementValue];
    } else {
        [item setObject:ElementValue forKey:elementName];
    }
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
@end
```

When you launch the application you can use the `UIStepper` object to increase and decrease the values you would like to pass. Once you tap the Calculate button, the SOAP operation is invoked and the result of the addition is displayed in the result label as shown in Figure 7-21.

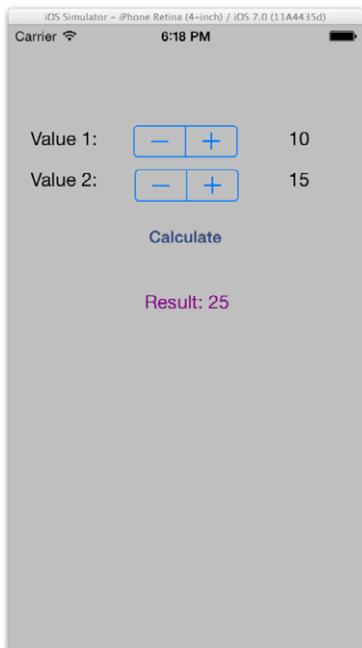


FIGURE 7-21

## Understanding Secure SOAP Requests

As you've seen, when a SOAP service is made available on a public IP address, each developer can study how the operation should be called and what the response would be. If you are working with non-critical information, this might be acceptable; however, if the SOAP service is linked to a company database, you don't want everyone to have access to the operations. To support this, you can program a SOAP operation in such a way that authentication is required. To demonstrate this, I've created the operation `GetProtectedData` that requires an authentication of the SOAP request.

Start Xcode, create a new project using the Single View Application Project template, and name it `SecureSOAP` using the options shown in Figure 7-22.

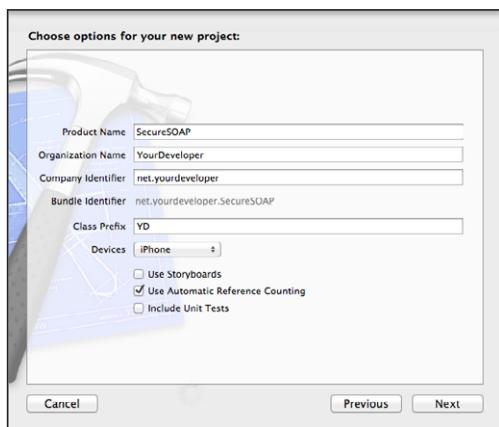


FIGURE 7-22

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor and create a user interface that looks like the one shown in Figure 7-23.



FIGURE 7-23

Copy the `URLRequest` class to your project and open the `YDViewController.h` file. Verify the Assistant Editor has created the properties as shown in Listing 7-21.

**LISTING 7-21:** Chapter7/SecureSOAP/YDViewController.h

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

@interface YDViewController : UIViewController

@property(nonatomic,weak) IBOutlet UILabel* resultLabel;
@property(nonatomic,weak) IBOutlet UITextField* nameField;
@property(nonatomic,weak) IBOutlet UITextField* passwordField;

@end
```

In the construction of the `soapMsg`, you need to add the `soap:Header` node that includes an `AuthSoap` node containing a username and a password. The SOAP operation expects this SOAP authentication information in its header and uses this to validate the passed username and password against the database.

**NOTE** *To test this program, enter the value test for both the username and the password.*

Again, the XML parser needs a small adjustment because the returning node you are looking for is `GetProtectedDataResult`, and it will return a secret message or the message “authentication failed.”

Because all the rest of the coding is similar to the application you developed earlier in this chapter, I will only show the construction of the SOAP message coding in Listing 7-22.

**LISTING 7-22:** Chapter 7/Construction of the SOAP message

```
//Construct the soapMessasge
NSString *soapMsg =[NSString stringWithFormat:@"<?xml version=\"1.0\""
encoding="utf-8"?>"
"<soap:Envelope"
"xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\""
"xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\""
"xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">"
"  <soap:Header>"
"    <AuthSoap xmlns=\"http://tempuri.org/\">"
"      <UserName>%@</UserName>"
"      <Password>%@</Password>"
"    </AuthSoap>"
"  </soap:Header>"
"  <soap:Body>"
"    <GetProtectedData xmlns=\"http://tempuri.org/\" />"
"  </soap:Body>"
"  </soap:Envelope>",self.nameField.text,
self.passwordField.text];
//Create the NSURL
```

*continues*

**LISTING 7-22 (continued)**

```

NSURL *url = [NSURL
URLWithString:@"http://developer.yourdeveloper.net/soapservice.asmx"];
NSMutableURLRequest *theRequest = [NSMutableURLRequest requestWithURL:url];
NSString *msgLength = [NSString stringWithFormat:@"%d", [soapMsg length]];
[theRequest addValue: @"text/xml; charset=utf-8"
forHTTPHeaderField:@"Content-Type"];
//Set the SOAP Action
[theRequest addValue: @"http://tempuri.org/GetProtectedData"
forHTTPHeaderField:@"SOAPAction"];
[theRequest addValue: msgLength forHTTPHeaderField:@"Content-Length"];
//make sure it's a POST
[theRequest setHTTPMethod:@"POST"];
[theRequest setHTTPBody: [soapMsg dataUsingEncoding:NSUTF8StringEncoding]];

```

When you launch the application and enter **test** as the username and password and you tap the Login button, the soap message will be created and sent to the SOAP server. The response will be parsed and shown in the user interface.

You can find the complete implementation of the `YDViewController.m` file in the project download for this chapter.

## MORE PARSING

In this chapter you've learned how to parse JSON and XML results using the standard APIs that come with Apple's framework. The implementation of the standard `NSXMLParser` can become complicated if the structure of the received XML document is complex. Also, the `NSXMLParser` is not a very fast parser, so when parsing large XML documents, it will probably not be as fast as you want.

Fortunately, many developers share their knowledge and techniques in the open source circuit. Two of the most commonly used XML parsers are `TBXML` and `KissXML`. You can download the full source packages via <https://github.com/71squared/TBXML> and <https://github.com/robbiehanson/KissXML>, respectively.

In most of my projects for which I need to parse XML results, I prefer to use the `TBXML` parser because it allows me to write “better readable” code and the parsing is about 20 times faster than with the standard `NSXML` parser.

## What about Comma-Separated Value Files?

A *comma-separated value file*, also known as CSV file, is probably the most widely known format for exchanging data between systems. Although many things are wrong with the format, especially if large chunks of text or text with special characters are involved (which happens a lot in many foreign languages), most computer systems are capable of producing a CSV export file.

Parsing a CSV file is pretty straightforward because it is one big string with data. A separator, which can be different for each file, separates each field, and your parsing needs to take this into account.

Field values might be quoted to be a literal; this again is depending on how the CSV is generated. This means a field value like 5 might be quoted as “5”.

Start Xcode, create a new project using the Single View Application Project template, and name it YDCSVParsing using the options shown in Figure 7-24.

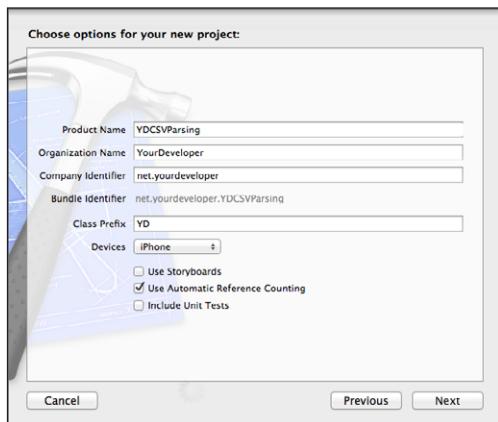


FIGURE 7-24

For this example you want to display some data in a UITableView so to save some time, delete all YDViewController files (.h, .m, and .xib). Create a new file that subclasses UITableViewController and name it YDviewController. Check the With XIB for User Interface box so Xcode will generate the class and the Interface Builder file.

You can take several approaches depending on your preferred method of accessing the data that will be read from the CSV file. In this example, I've chosen to implement a Category on an NSArray.

In Xcode, choose File → New → File, and in the pop-up screen select Objective-C Category as shown in Figure 7-25.

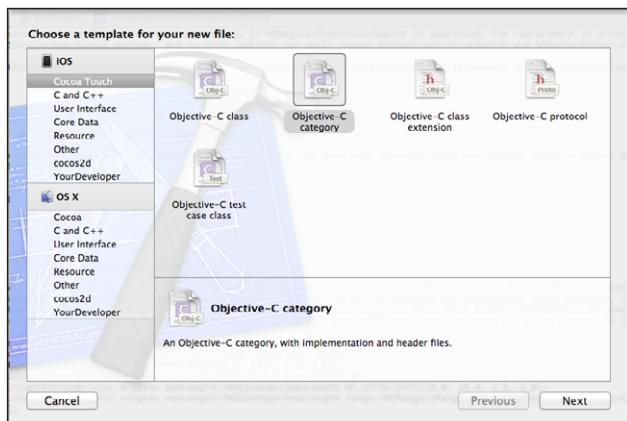


FIGURE 7-25

In the next screen, enter CSV in the Category field and select NSArray in the Category On dropdown, as shown in Figure 7-26.

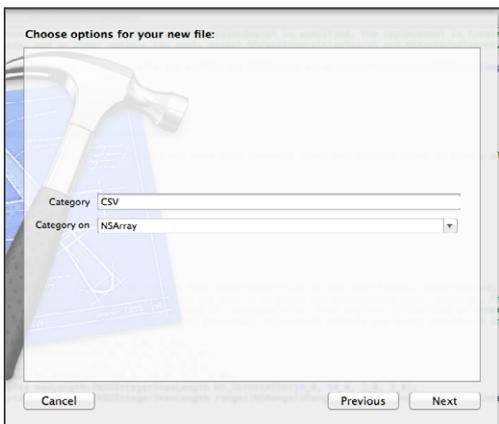


FIGURE 7-26

This results in an `NSArray+CSV.h` and an `NSArray+CSV.m` file in your project structure.

In the header file for the `NSArray+CSV` category, create two static methods: one that accepts a string containing the CSV data as one string and another one that accepts the path to a CSV file, as shown in Listing 7-23.

#### LISTING 7-23: Chapter7/YDCSVParsing/NSArray+CSV.h

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

@interface NSArray (CSV)
+ (NSArray *)arrayWithCSVString:(NSString *)string;
+ (NSArray *)arrayWithCSVFile:(NSString *)path;
@end
```

The implementation of the category starts with some definitions. You need to identify the line and field separators, a `BOOL` value used to identify whether field quotes need to be removed during processing, and another `BOOL` that identifies whether the first line, by default containing the field names, should be omitted in the result.

The `arrayWithCSVFile:` method accepts a string for the path to the CSV file and returns the array by passing on the content string to the `arrayWithCSVString:` method. This method is responsible for the actual parsing; it creates an array named `lines` containing the raw unparsed lines as single string values. Next, the array named `fieldnames` is created based on the assumption that the top line in the CSV file always contains the field names. In a loop, each line is processed and parsed and an `NSDictionary` is created for each cell of data, one entry with the field name and one entry with the field value. The complete implementation is shown in Listing 7-24, but you can modify it to meet your specific requirements.

**LISTING 7-24:** Chapter7/YDCSVParsing/NSArray+CSV.m

```

#import "NSArray+CSV.h"
#define kLineSeparator      @"\n"
#define kFieldSeparator     @","
#define kRemoveFieldQuotes  YES
#define kFieldQuote          @""@"
#define kSkipFirstLine       YES

@implementation NSArray (CSV)

+ (NSArray *)arrayWithCSVFile:(NSString *)path
{
    NSString *contentString = [NSString stringWithContentsOfFile:path
encoding:NSUTF8StringEncoding error:nil];
    return [self arrayWithCSVString:contentString];
}
+ (NSArray *)arrayWithCSVString:(NSString *)string
{
    NSArray* lines = [string componentsSeparatedByString:kLineSeparator];
    //Create an Array to store the fieldnames, we assume this is the first line
    //of the file
    NSArray* fieldNames = [[lines objectAtIndex:0]
componentsSeparatedByString:kFieldSeparator];
    //cleanLines will contain an object for each line in the CSV
    NSMutableArray* cleanLines =[[NSMutableArray alloc] init];
    //loop over each line
    for (int i=0;i<[lines count];i++)
    {
        //create an array to hold the fields
        NSMutableArray *fields = [NSMutableArray array];
        //These are the parsed rawfield contents
        NSArray* rawFields= [[lines objectAtIndex:i]
componentsSeparatedByString:kFieldSeparator];
        //for each field
        for(int j=0;j<[rawFields count];j++)
            {//create a NSDictionary to store fieldvalue and fieldname
                NSMutableDictionary *field = [NSMutableDictionary dictionary];
                NSString *fieldName;
                NSString *fieldVal;
                //for the field name always remove quotes
                fieldName=[[fieldNames objectAtIndex:j]
stringByReplacingOccurrencesOfString:
kFieldQuote withString:@""];
                if (kRemoveFieldQuotes)
                    {//if option is set replace kFieldQuote symbol with empty string
                    fieldVal=[[rawFields objectAtIndex:j]
stringByReplacingOccurrencesOfString:kFieldQuote withString:@""];
                }
                else
                {
                    fieldVal=[rawFields objectAtIndex:j];
                }
            }
    }
}

```

*continues*

**LISTING 7-24 (continued)**

```

        }
        //set the values
        [field setValue:fieldVal forKey:@"fieldvalue"];
        [field setValue:fieldName forKey:@"fieldname"];
            //add to the fields array
        [fields addObject:field];
    }
    //add object to cleanLines Array
    [cleanLines addObject:fields];

}
//the first line containing file headers will be removed from the two arrays
so the resulting NSDictionary contains only data

if (kSkipFirstLine)
    [cleanLines removeObjectAtIndex:0];

return cleanLines;
}

@end

```

To display the result of your category implementation, open the `YDViewController` and create a property of type `NSArray` named `countries`, as shown in Listing 7-25.

**LISTING 7-25: Chapter7/YDCSVParsing/YDViewController.h**

```

#import <UIKit/UIKit.h>

@interface YDViewController : UITableViewController

@property(nonatomic,strong) NSArray* countries;
@end

```

The implementation of the `YDViewController` is simply obtaining the file path to a CSV file in the bundle. A small CSV file is included in the source code sample you can download. You can create the `countries` array by calling the category method `arrayWithCSVString:`. In the `tableView:numberOfRowsInSection:` delegate method you return the `[self.countries count]`, and in the `tableView:cellForRowAtIndexPath` you access the object from the `NSArray` the `self.countries` array using the `objectAtIndex:indexPath.row` statement. To obtain a field you can create an `NSDictionary` using the `[dataRow objectAtIndex:]` method, where you need to check if the index exists to avoid a crash. This `NSDictionary` will contain two keys: one named `fieldname` and one named `fieldvalue`. The complete implementation is shown in Listing 7-26.

**LISTING 7-26:** Chapter7/YDCSVParsing/YDViewController.m

```

#import "YDViewController.h"
#import "NSArray+CSV.h"
@interface YDViewController : NSObject

@end

@implementation YDViewController
@synthesize countries=_countries;
- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
NSError* error = nil;
NSString *myCountries = [NSString stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding error:&error];
NSString *myCountries = [NSString stringWithContentsOfFile:filePath];
self.countries = [NSArray arrayWithCSVString:myCountries];

}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

#pragma mark - Table view data source

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

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

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

```

*continues*

**LISTING 7-26 (continued)**

```

{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier];
    }
    //This call will return the datarow that represents the line from the
original CSV
    NSArray* datarow = [self.countries objectAtIndex:indexPath.row];
    //obtain the datafield dictionary for a field at index
    // in this example the sixth field is returned
    NSDictionary* datafield = [datarow objectAtIndex:6];
    // Configure the cell...
    //access the datafield dictionary with key fieldvalue or fieldname
    cell.textLabel.text=[datafield objectForKey:@"fieldvalue"];
    return cell;
}

@end

```

Based on this simple CSV parsing category you can create your own customized versions that meet specific requirements or return a different structure of data.

## Transforming XML to an NSDictionary

As you've seen during the XML parsing, there is always some code involved, and you need to write specific XML parsing code for each project. When the XML is pretty straightforward you can use the following `YDXMLParser` that accepts an XML string and returns an `NSDictionary`, which is created automatically. The header file is shown in Listing 7-27.

**LISTING 7-27: Chapter7/XMLToDict/YDXMLReader.h**

```

#import <Foundation/Foundation.h>

@interface YDXMLReader : NSObject
{
    NSMutableArray *stack;
    NSMutableString *currentText;
}

+ (NSDictionary *)dictionaryFromXMLData:(NSData *)data;
+ (NSDictionary *)dictionaryFromXMLString:(NSString *)string;

@end

```

The implementation is shown in Listing 7-28. I've included a lot of comments in the code to explain what is happening.

**LISTING 7-28: Chapter7/XMLToDict/YDXMLReader.m**

```
#import "YDXMLReader.h"
//if your XML contains a node names textvalue you can change this const to
a different value
NSString *const kTextNodeKey = @"textvalue";

@interface YDXMLReader (Internal)<NSXMLParserDelegate>

- (id)init;
- (NSDictionary *)objectWithData:(NSData *)data;

@end

@implementation YDXMLReader

+ (NSDictionary *)dictionaryFromXMLData:(NSData *)data
{
    YDXMLReader *reader = [[YDXMLReader alloc] init];
    NSDictionary *rootDictionary = [reader objectWithData:data];
    return rootDictionary;
}

+ (NSDictionary *)dictionaryFromXMLString:(NSString *)string
{
    NSData *data = [string dataUsingEncoding:NSUTF8StringEncoding];
    return [YDXMLReader dictionaryFromXMLData:data];
}

- (id)init
{
    if (self = [super init])
    {

    }
    return self;
}

- (NSDictionary *)objectWithData:(NSData *)data
{
    stack=nil;
    currentText =nil;
    stack = [[NSMutableArray alloc] init];
    currentText = [[NSMutableString alloc] init];
    // Initialize the stack with a fresh dictionary
```

*continues*

**LISTING 7-28 (continued)**

```

[stack addObject:[NSMutableDictionary dictionaryWithDictionary]];
// Create and Initialize the parser object
NSXMLParser *parser = [[NSXMLParser alloc] initWithData:data];
    //set delegate to self
parser.delegate = self;
BOOL success = [parser parse];
// Return the stack's root dictionary on success
if (success)
{
    return [stack objectAtIndex:0];
}
//else return nil
return nil;
}

#pragma mark NSXMLParserDelegate methods

- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
attributes:(NSDictionary *)attributeDict
{
    // Get last dict from stack
NSMutableDictionary *parentDict = [stack lastObject];
    // Create a child dictionary for the new element, and initilaize
NSMutableDictionary *childDict = [NSMutableDictionary dictionaryWithDictionary];
[childDict addEntriesFromDictionary:attributeDict];
    // if there is already an item create a new array
id existingValue = [parentDict objectForKey:elementName];
if (existingValue)
{
    NSMutableArray *array = nil;
    if ([existingValue isKindOfClass:[NSMutableArray class]])
    {
        //if the array exists and it is indeed an array use it
        array = (NSMutableArray *) existingValue;
    }
    else
    {
        // Create an new array
        array = [NSMutableArray array];
        [array addObject:existingValue];
        // Replace the child dictionary with an array of children dictionaries
        [parentDict setObject:array forKey:elementName];
    }
    // Add the new child dictionary to the array
    [array addObject:childDict];
}
else
{
    // Nothing so update the dictionary
}
}

```

```

        [parentDict setObject:childDict forKey:elementName];
    }

    // Update the stack by adding the childDict
    [stack addObject:childDict];
}

- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
{
    // get last object from stack
    NSMutableDictionary *dictInProgress = [stack lastObject];
    // Set the text property
    if ([currentText length] > 0)
    {
        [dictInProgress setObject:currentText forKey:kTextNodeKey];
        //Reset the mutable string
        currentText = [[NSMutableString alloc] init];
    }

    // update the stack
    [stack removeLastObject];
}
- (void)parser:(NSXMLParser *)parser foundCDATA:(NSData *)CDATABlock
{
    //In case a node contains a CDATA element we need to append the
    Current String with the CDATABlock
    [currentText appendString:[NSString alloc] initWithData:CDATABlock
encoding:NSUTF8StringEncoding]];
}
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    [currentText appendString:string];
}

- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError
{
    NSLog(@"parseError: %@",parseError);
}

@end

```

To use this in your application, you can implement the code as shown in the following snippet:

```

//Create some testXML string
NSString *testXML = @"<items><item id=\"0001\" type=\"iMac\"><name>
<! [CDATA[iMac 21 inch]]></name><price>1995</price><options><option
id=\"1001\">Mouse</option><option id=\"1002\">Wireless keyboard</option><option
id=\"1003\">Touchpad</option></options></item></items>";
//Create a Dictionary
NSDictionary *xmlDictionary = [YDXMLReader dictionaryFromXMLString:testXML ];

```

## SUMMARY

In this chapter you learned how to make network calls to websites, RESTful services, and SOAP services, and how to parse the results they send you.

In the next chapter you learn how to implement the FTP protocol in your application, which is a more profound solution for transferring larger files like video and large PDF documents.

# 8

## Using FTP

### WHAT'S IN THIS CHAPTER?

---

- Understanding the File Transfer Protocol
- Developing a simple FTP client
- Implementing Network Streams

### WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 8 download and individually named according to the names throughout the chapter.

In Chapter 7 you learned about networking, how to download files from a web server, and make post requests to RESTful and SOAP services. In some cases you might have to deal with large files such as videos that either need to be downloaded into your application or uploaded to a server for processing.

With images and PDF documents you can encode the binary data to a base64 string and post that to a server, but this is not a very efficient and fast process. Creating the base64 string can take quite some time, and then you still have to transfer the data.

The *File Transfer Protocol* (FTP) is a protocol that is used to communicate between an FTP server and an FTP client.

## DEVELOPING AN FTP CLIENT

When developing an FTP client for an iOS application you must understand some basic elements of the FTP protocol. The client makes a connection to the server using an Internet connection on a pre-defined port. By default, the configuration of an FTP system uses the following ports:

- Port 20 is used when the server is initiating the connection to a client.
- Port 21 is used when the client is initiating a connection to the server.

The configuration of the FTP server can be different, so it's important that you agree with the system manager of the FTP server as to which port will be used.

In addition to agreeing on a port number to set up a network stream, the FTP protocol also has pre-defined commands that are exchanged so the client can tell the server what it wants and vice versa.

The most commonly used commands are:

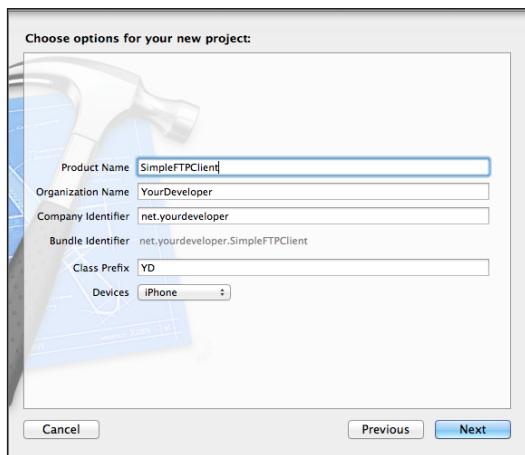
- **OPEN:** Open a connection
- **CLOSE:** Close the connection
- **GET:** Copy a file from the server to the client (download)
- **MGET:** Copy multiple files from the server to the client (download)
- **PUT:** Copy a file from the client to the server (upload)
- **MPUT:** Copy multiple files from the client to the server (upload)
- **DELETE:** Delete a file in the current remote directory
- **CD:** Change directory on the server (initiated by the client)
- **LCD:** Change directory on the client
- **MKDIR or MKD:** Create a directory on the server (initiated by the client)

If you have never worked with an FTP client before, you can download the FileZilla project from the project's website URL at <http://filezilla-project.org/>. FileZilla is an open source FTP server and client that can be configured and used to analyze the specific commands that are exchanged between the two.

Depending on the requirements you have in your application, you have basically two options to consider. If you simply need to upload or download a file from an FTP server, you can use a higher-level API provided by the CFNetwork framework. This solution will have limited capabilities and is definitely not a complete FTP client. On the other hand, if you need to have full control over the FTP process, you can write your own FTP client class that will accept all of the raw FTP commands and responses.

## Writing a Simple FTP Client

Start Xcode and create a new project using the Single View Application Project template, and name it `SimpleFTPCClient` using the options shown in Figure 8-1.



**FIGURE 8-1**

Start with adding the CFNetwork framework to your project.

Create a new class named `FTPManager` that inherits from `NSObject`. Open the `FTPManager.h` file and start by including the CFNetwork framework by using `#includes <CFNetwork/CFNetwork.h>`.

Next, define the `FTPManagerDelegate` protocol with the methods that will provide feedback to the delegate.

Create an initializer that accepts the server, username, and password for the FTP connection. The implementation is assuming you always need a username and a password to set up the FTP connection, because it wouldn't make sense to open a public FTP server for your application's purpose.

Next, declare four methods for the supported operations that relate to the FTP commands, as shown in Table 8-1.

**TABLE 8-1:** Relation between Methods and FTP Commands

METHOD	FTP COMMAND
<code>downloadRemoteFile</code>	GET
<code>uploadFileWithFilePath</code>	PUT
<code>createRemoteDirectory</code>	MKD
<code>listRemoteDirectory</code>	LIST

Finally, create a public property for the delegate of the class.

The complete code is shown in Listing 8-1.

**LISTING 8-1:** Chapter8/SimpleFTPCClient/FTPManager.h

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

#include <CFNetwork/CFNetwork.h>
enum {
    kSendBufferSize = 32768
};

@protocol FTPManagerDelegate <NSObject>
- (void)ftpUploadFinishedWithSuccess:(BOOL)success;
- (void)ftpDownloadFinishedWithSuccess:(BOOL)success;
- (void)directoryListingFinishedWithSuccess:(NSArray *)arr;
- (void)ftpError:(NSString *)err;
@end

@interface FTPManager : NSObject<NSStreamDelegate>
- (id)initWithServer:(NSString *)server user:(NSString *)username
               password:(NSString *)pass;
- (void)downloadRemoteFile:(NSString *)filename localFileName:
               (NSString *)localname;
- (void)uploadFileWithPath:(NSString *)filePath;
- (void)createRemoteDirectory:(NSString *)dirname;
- (void)listRemoteDirectory;
@property (nonatomic, assign) id<FTPManagerDelegate> delegate;
@end
```

Open the `FTPManager.m` file and start by writing the private interface just after the import statement for the header file.

Because you are using streams for the network communication, you use two different types of streams: the `NSOutputStream` and the `NSInputStream`.

At this point it's important to understand that the FTP `PUT` and `MKD` commands are commands you send to the server, and therefore require an `NSOutputStream`. The `LIST` and `GET` commands are sent over the open socket and will respond with the data you requested, which you need to read, so you need an `NSInputStream`.

Two `BOOL` properties are defined (`isReceiving` and `isSending`) to keep track of the status of the streams, and to avoid mixing of data between stream operations.

You can find the complete implementation of the `FTPManager` in the download of this chapter.

Here, the `FTPManager.m` implementation is broken down and explained step by step. The first step is to initialize the `FTPManager` object using a custom initializer in which you can pass the connection and authentication properties, as shown in the Listing 8-2.

**LISTING 8-2:** The initWithServer method

```
- (id)initWithServer:(NSString *)server user:(NSString *)username
              password:(NSString *)pass
{
    if ((self = [super init]))
    {
        self.ftpServer = server;
        self.ftpUsername=username;
        self.ftpPassword=pass;
    }
    return self;
}
```

A potential problem with stream processing is blocking. A thread that is writing to or reading from a stream might have to wait indefinitely until there is space available on the stream to write to the stream, or until bytes are available on the stream to read. To overcome this problem, you need a method that will accept an `NSStream` as an argument and schedule it in the current `NSRunLoop` so that the delegate receives messages reporting stream-related events only when blocking is unlikely to take place. For this purpose, write a simple helper method as shown in Listing 8-3.

The `NSRunLoop` class declares the programmatic interface to objects that manage input sources. An `NSRunLoop` object processes input for sources such as mouse and keyboard events from the window system, `NSPort` objects, and `NSConnection` objects. An `NSRunLoop` object also processes `NSTimer` events.

Your application cannot either create or explicitly manage `NSRunLoop` objects. Each `NSThread` object, including the application's main thread, has an `NSRunLoop` object automatically created for it as needed. If you need to access the current thread's run loop, you do so with the class method `currentRunLoop`.

**LISTING 8-3:** The scheduleInCurrentThread method

```
- (void)scheduleInCurrentThread:(NSStream*)aStream
{
    [aStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
              forMode:NSRunLoopCommonModes];
}
```

The `smartURLForString:` method accepts a string and transforms it to a valid `NSURL` object. If you pass in an IP address like `127.0.0.1`, it will return `ftp://127.0.0.1` as a `NSURL` object.

The `smartURLForString:` is shown in Listing 8-4.

**LISTING 8-4:** The smartURLForString method

```
- (NSURL *)smartURLForString:(NSString *)str
{
    NSURL *      result;
    NSString *   trimmedStr;
```

*continues*

**LISTING 8-4** (continued)

```

NSRange      schemeMarkerRange;
NSString *  scheme;
result = nil;
trimmedStr = [str stringByTrimmingCharactersInSet:
              [NSCharacterSet whitespaceCharacterSet]];
if ( (trimmedStr != nil) && ([trimmedStr length] != 0) ) {
    schemeMarkerRange = [trimmedStr rangeOfString:@"://"];
    if (schemeMarkerRange.location == NSNotFound) {
        result = [NSURL URLWithString:[NSString stringWithFormat:
                                         @"ftp://%@", trimmedStr]];
    } else {
        scheme = [trimmedStr substringWithRange:
                   NSMakeRange(0, schemeMarkerRange.location)];
        if ( ([scheme compare:@"http" options:
               NSCaseInsensitiveSearch] == NSOrderedSame) ) {
            result = [NSURL URLWithString:trimmedStr];
        } else {
            //unsupported url schema
        }
    }
}
return result;
}

```

The `isReceiving` method is checking if the `dataStream` is initialized, and the `isSending` method is checking if the `commandStream` is initialized. These methods are used in the network communication to avoid running a command while another one is still in progress.

The `isReceiving` and `isSending` methods are shown in Listing 8-5.

**LISTING 8-5:** The `isReceiving` and `isSending` methods

```

- (BOOL)isReceiving
{
    return (_dataStream != nil);
}
- (BOOL)isSending
{
    return (_commandStream != nil);
}

```

The `closeAll` method is called at the end of each operation. If a stream can't be opened and it will close the `filestream` if it's open and remove the `commandStream` and `dataStream` from the current RunLoop and close them.

The `closeAll` method is shown in Listing 8-6.

**LISTING 8-6:** The closeAll method

```

-(void)closeAll
{
    if (_commandStream != nil) {
        [_commandStream removeFromRunLoop:
            [NSRunLoop currentRunLoop] forMode:NSUTFDefaultRunLoopMode];
        _commandStream.delegate = nil;
        [_commandStream close];
        _commandStream = nil;
    }
    if (_uploadStream != nil) {
        [_uploadStream close];
        _uploadStream = nil;
    }
    if (_downloadfileStream != nil) {
        [_downloadfileStream close];
        _downloadfileStream = nil;
    }
    if (_dataStream != nil) {
        [_dataStream removeFromRunLoop:
            [NSRunLoop currentRunLoop] forMode:NSUTFDefaultRunLoopMode];
        _dataStream.delegate = nil;
        [_dataStream close];
        _dataStream = nil;
    }
    _currentOperation = @"";
}

```

## Downloading a Remote File

To download a file from the FTP server you call the `downloadRemoteFile:localFileName:` method and pass the filename, which is the remote filename on the server; for example, `picture1.png` and a local filename. The downloaded file will be written with the passed local filename in the temporary directory under your application's root.

The method starts with creating the `smartURLforString` with the FTP server address followed by a `/` and the remote filename, resulting in a `NSURL` object like `ftp://127.0.0.1/picture1.png`. If the `isReceiving` method is returning `YES`, the delegate method `ftpError` is called with an error message; otherwise the `downloadStream` is initialized with the path created. The `downloadStream` is opened so data can be read from the stream, and the `currentOperation` property is set to `GET`. The stream delegate that is responsible for reading and writing data for the different streams needs to operate differently for a `GET` command (reading data from the stream and writing to a file) as with the `LIST` command, which is streaming the directory listing that needs to be parsed. For that reason you have a `currentOperation` property.

**NOTE** Instead of the 127.0.0.1 address you should use the IP address of your FTP server.

You create the `commandStream` using a `CFBridgeRelease` for the `CFWriteStreamCreateWithFTPURL` function, passing the earlier created URL.

Because you are using authentication you need to set the `username` and `password` values passed during the initialization of the `FTPManager` to the `commandStream`. Finally, set the `delegate` to `self` and call the method to schedule the `commandstream` in the current thread and open the stream.

When the `commandstream` is opened, the connection is established and the `username` and `password` are used to authenticate the stream. The server response is captured in the `stream:handleEvent:` method, which will be explained later because the other methods you are learning right now are using the same method for its result processing.

The `downloadRemoteFile:` method is shown in Listing 8-7.

#### LISTING 8-7: The downloadRemoteFile method

```
- (void)downloadRemoteFile:(NSString *)filename localFileName:(NSString *)localname
{
    BOOL                     success;
    NSURL *                  url;
    url = [self smartURLForString:[NSString stringWithFormat:@"%@/%@",
                                 _ftpServer, filename]];
    success = (url != nil);
    if (!success) {
        [self.delegate ftpError:@"invalid url for downloadRemoteFile method"];
    } else {
        if (self.isReceiving) {
            [self.delegate ftpError:@"receiving in progress"];
            return ;
        }
        NSString *path = [NSTemporaryDirectory()
                          stringByAppendingPathComponent:localname];
        _downloadfileStream = [NSOutputStream outputStreamToFileAtPath:
                               path append:NO];
        [_downloadfileStream open];
        _currentOperation = @"GET";
        _dataStream= CFBridgeingRelease(
            CFReadStreamCreateWithFTPURL(NULL,
                                         (_bridge CFURLRef) url));
        [_dataStream setProperty:_ftpUserName
                      forKey:(id)kCFStreamPropertyFTPUUserName];
        [_dataStream setProperty:_ftpPassword
                      forKey:(id)kCFStreamPropertyFTPPassword];

        _dataStream.delegate = self;
        [self performSelector:@selector(scheduleInCurrentThread:)
                      onThread:[[self class] networkThread]
                      withObject:_dataStream
                      waitUntilDone:YES];
        [_dataStream open];
    }
}
```

## Creating a Remote Directory

You can use the `createRemoteDirectory:` method to create a directory on the FTP server. Of course, the credentials you provide to set up the connection need to have the access rights to create a directory. The implementation is similar to the download example you just learned. The URL that is created is created using the `CFURLCreateCopyAppendingPathComponent`, where the requested directory name to be created is appended to the URL resulting in a URL like `ftp://127.0.0.1/newdirname`. Because you are sending a command to the FTP server, the `commandStream` is initialized instead of the `dataStream`, the credentials are passed, and the stream is opened. Also, here the `stream handleEvent:` method is responsible for processing the response. The `createRemoteDirectory` method is shown in Listing 8-8.

**LISTING 8-8:** The `createRemoteDirectory` method

```
- (void)createRemoteDirectory:(NSString *)dirname
{
    BOOL                     success;
    NSURL *                  url;

    url = [self smartURLForString:_ftpServer];
    success = (url != nil);
    if (!success) {
        url = CFBridgingRelease(
            CFURLCreateCopyAppendingPathComponent(NULL,
            (__bridge CFURLRef) url,
            (__bridge CFStringRef) dirname, true)
        );
        success = (url != nil);
    }
    if (!success) {
        [self.delegate ftpError:@"invalid url for createRemoteDirectory method"];
    } else {
        if (self.isSending) {
            [self.delegate ftpError:@"sending in progress"];
            return;
        }
        _commandStream = CFBridgingRelease(
            CFWriteStreamCreateWithFTPURL(NULL,
            (__bridge CFURLRef) url)
        );
        //set credentials
        [_commandStream SetProperty:_ftpUsername
        forKey:(id)kCFStreamPropertyFTPUUserName];
        [_commandStream SetProperty:_ftpPassword
        forKey:(id)kCFStreamPropertyFTPPassword];
        _commandStream.delegate = self;
        [self performSelector:@selector(scheduleInCurrentThread:)
        onThread:[[self class] networkThread]
        withObject:_commandStream
        waitUntilDone:YES];
        [_commandStream open];
    }
}
```

## Listing a Remote Directory

The `listRemoteDirectory` is the equivalent of the FTP `LIST` command, which lists the contents of a directory on the FTP server. Although several RFCs are applicable to the FTP process, the result of the FTP `LIST` command is not 100 percent standardized as you would like, so it can be parsed easily into an `NSArray`.

After checking that you're not already receiving data on the `dataStream`, you set the `listData` property and initialize the `listEntries` property. You set the `currentOperation` property to `LIST` to differentiate between the `GET` and the `LIST` as explained before, and initialize the `dataStream` with the `CFReadStreamCreateWithFTPURL`, passing the created URL.

Again, after setting the credentials on the `dataStream` it is scheduled to run and open.

To be able to parse the results of the directory listing command, three helper methods are implemented that will parse the incoming data stream and finally write the results to the `listEntries` property array. After all data has been processed, the `directoryListingFinishedWithSuccess` is called, which is returning an `NSArray` with the files found on the FTP server. The `listRemoteDirectory` method is shown in Listing 8-9.

**LISTING 8-9:** The `listRemoteDirectory` method

```
- (void)listRemoteDirectory
{
    BOOL             success;
    NSURL *          url;
    url = [self smartURLForString:_ftpServer];

    success = (url != nil);
    if ( ! success) {
        [self.delegate ftpError:@"invalid url for listRemoteDirectory method"];
    } else {
        if (self.isReceiving) {
            [self.delegate ftpError:@"receiving in progress"];
            return ;
        }
        self.listData = [NSMutableData data];
        if (self.listEntries)
            self.listEntries=nil;
        self.listEntries=[[NSMutableArray alloc] init];
        _currentOperation = @"LIST";
        self.dataStream = CFBridgingRelease(
            CFReadStreamCreateWithFTPURL(NULL,
                (__bridge CFURLRef) url));
        //set credentials
        [self.dataStream setProperty: self.ftpUsername
            forKey:(id)kCFStreamPropertyFTPUUserName];
        [self.dataStream setProperty: self.ftpPassword
            forKey:(id)kCFStreamPropertyFTPPassword];
        self.dataStream.delegate = self;
        [self performSelector:@selector(scheduleInCurrentThread:)];
    }
}
```

```
        onThread: [[self class] networkThread]
        withObject: _dataStream
        waitUntilDone:YES];
    [self.dataStream open];
}
```

The following methods are taken from an Apple example and have been cleaned up. Because the response from an FTP server is not 100 percent standardized, you can use a sample parsing routine from an Apple example program that works on most server implementations, and will present you an `NSArray` with the returned file information. The parsing helper methods are shown in Listing 8-10.

**LISTING 8-10:** Parsing helper methods

```
pragma listing helpers

- (void)addListEntries:(NSArray *)newEntries
{
    [ self.listEntries addObjectFromArray:newEntries];
    [self closeAll];
    [self.delegate directoryListingFinishedWithSuccess: self.listEntries];
}
//this function is taken over from Apple samples
- (NSDictionary *)entryByReencodingNameInEntry:(NSDictionary *)entry
                                         encoding:(NSStringEncoding)newEncoding
{
    NSDictionary * result;
    NSString * name;
    NSData * nameData;
    NSString * newName;
    newName = nil;
    // Try to get the name, convert it back to MacRoman, and then reconvert it
    // with the preferred encoding.
    name = [entry objectForKey:(id) kCFFTPResourceName];
    if (name != nil) {
        nameData = [name dataUsingEncoding:NSUTF8StringEncoding];
        if (nameData != nil) {
            newName = [[NSString alloc]
                       initWithData:nameData encoding:newEncoding];
        }
    }
    if (newName == nil) {
        result = (NSDictionary *) entry;
    } else {
        NSMutableDictionary * newEntry;
        newEntry = [entry mutableCopy];
        [newEntry setObject:newName forKey:(id) kCFFTPResourceName];
        result = newEntry;
    }
    return result;
}
```

*continues*

**LISTING 8-10 (continued)**

```

//also this function is taken over from Apple samples
- (void)parseListData
{
    NSMutableArray *      newEntries;
    NSUInteger            offset;
    newEntries = [NSMutableArray array];
    offset = 0;
    do {
        CFIndex           bytesConsumed;
        CFDictionaryRef   thisEntry;
        thisEntry = NULL;
        bytesConsumed = CFFTPCreateParsedResourceListing(NULL, &((const uint8_t *) self.listData.bytes)[offset],
                                                       (CFIndex) ([self.listData length] - offset), &thisEntry);
        if (bytesConsumed > 0) {
            if (thisEntry != NULL) {
                NSDictionary * entryToAdd;
                entryToAdd = [self entryByReencodingNameInEntry:
                             (_bridge NSDictionary *) thisEntry
                             encoding:NSUTF8StringEncoding];
                [newEntries addObject:entryToAdd];
            }
            // We consume the bytes regardless of whether we get an entry.
            offset += (NSUInteger) bytesConsumed;
        }
        if (thisEntry != NULL) {
            CFRelease(thisEntry);
        }
        if (bytesConsumed == 0) {
            // We haven't yet got enough data to parse an entry.
            //Wait for more data to arrive
            break;
        } else if (bytesConsumed < 0) {
            // We totally failed to parse the listing. Fail.
            break;
        }
    } while (YES);

    if ([newEntries count] != 0) {
        [self addListEntries:newEntries];
    }
    if (offset != 0) {
        [self.listData replaceBytesInRange:NSMakeRange(0, offset)
                                     withBytes:NULL length:0];
    }
}

```

The resulting array will provide information similar to the following:

```
{
    kCFFTPResourceGroup = ftp;
    kCFFTPResourceLink = "";
```

```

kCFFTPResourceModDate = "2013-03-22 14:34:00 +0000";
kCFFTPResourceMode = 420;
kCFFTPResourceName = "1.jpg";
kCFFTPResourceOwner = ftp;
kCFFTPResourceSize = 5855;
kCFFTPResourceType = 8;

```

Each entry in the `NSArray` contains an `NSDictionary` with the following elements, as explained in Table 8-2.

**TABLE 8-2:** File Elements

ELEMENT	EXPLANATION
<code>kCFFTPResourceGroup</code>	Most of the time this returns <code>ftp</code>
<code>kCFFTPResourceLink</code>	Most of the time this returns <code>void</code>
<code>kCFFTPResourceModDate</code>	Returns the last modification date of the file
<code>kCFFTPResourceMode</code>	<code>420 = file</code> <code>493 = directory</code>
<code>kCFFTPResourceName</code>	Returns the local filename
<code>kCFFTPResourceOwner</code>	Returns the owner of the file; many servers will return <code>ftp</code> as the owner
<code>kCFFTPResourceSize</code>	Returns the size in bytes
<code>kCFFTPResourceType</code>	<code>4 = directory</code> <code>8 = file</code>

## Uploading a File

To upload a file to the FTP server, you can use the `uploadFileWithFilePath:` method by passing the `filePath` of the file you want to upload.

You create the URL using the `CFURLCreateCopyAppendingPathComponent` by appending the `smartURLForString:` with the `lastPathComponent` of the passed `filePath`.

The `uploadStream` is created with the `filePath` and opened so the file can be read using the `NSInputStream`. Next, the `commandStream` is created with the `CFWriteStreamCreateWithFTPURL` function using the earlier created URL, and the credentials are set on the `commandStream`.

The `commandStream` is scheduled in the `NSRunLoop` and opened where, again, the `stream:handleEvent:` method is responsible for processing the response. The `uploadFileWithFilePath` method is shown in Listing 8-11.

**LISTING 8-11:** The uploadFileWithFilePath method

```

- (void)uploadFileWithFilePath:(NSString *)filePath
{
    BOOL                     success;
    NSURL *                  url;
    url = [self smartURLForString:_ftpServer];
    success = (url != nil);
    if (success) {
        url = CFBridgeRelease(
            CFURLCreateCopyAppendingPathComponent(NULL,
                (CFURLRef) url,
                (CFStringRef) [filePath lastPathComponent], false));
        success = (url != nil);
    }
    if (!success) {
        [self.delegate ftpError:@"invalid url for uploadFileWithFilePath method"];
    } else {
        if (self.isSending) {
            [self.delegate ftpError:@"sending in progress"];
            return;
        }
        self.uploadStream = [NSInputStream
            inputStreamWithFileAtPath:filePath];
        [self.uploadStream open];
        self.commandStream = CFBridgeRelease(CFWriteStreamCreateWithFTPURL(NULL,
            (_bridge CFURLRef) url));
        //set credentials
        [self.commandStream setProperty:_ftpUsername
            forKey:(id)kCFStreamPropertyFTPPUserName];
        [self.commandStream setProperty:_ftpPassword
            forKey:(id)kCFStreamPropertyFTPPassword];
        self.commandStream.delegate = self;
        [self performSelector:@selector(scheduleInCurrentThread:)
            onThread:[[[self class] networkThread]
            withObject:self.commandStream
            waitUntilDone:YES];
        [self.commandStream open];
    }
}

```

As mentioned before, the `stream:handleEvent:` method is the method where all the magic happens with the stream communication. The implementation first uses a switch statement to differentiate processing for the various `NSStreamEvents` that can happen.

## Reading from an `NSStream`

As you can see in the implementation shown in Listing 8-12, if the `eventCode` is an `NSStreamEventHasBytesAvailable`, meaning there are bytes available for reading, you create a buffer and read the bytes from the stream into the buffer. Once all the bytes have been read from the stream, depending on the current operation, you call the `parseListData` method if you were performing a `listRemoteDirectory` to parse the results, or you write to file to storage if you were downloading a file.

## Writing to an NSStream

When you want to write to an `NSStream`, you need to check if the event is `NSStreamEventHasSpaceAvailable`, simply being called when the `NSStream` has space available to be written to. Because the stream is scheduled in the current run loop, the thread won't block if there is not space available to write. Now you can simply write bytes from a buffer to the stream.

**LISTING 8-12:** The `stream:handleEvent:` method

```
- (void)stream:(NSStream *)aStream handleEvent:(NSStreamEvent)eventCode
{
#pragma unused(aStream)
    switch (eventCode) {
        case NSStreamEventOpenCompleted: {
            NSLog(@"stream openend");
        } break;
        case NSStreamEventHasBytesAvailable: {
            NSInteger bytesRead;
            uint8_t buffer[32768];
            // Pull some data off the network.
            if ([_currentOperation isEqualToString:@"LIST"]){
                bytesRead = [_dataStream read:buffer maxLength:sizeof(buffer)];
                if (bytesRead < 0) {
                    [self.delegate ftpError:@"can't read data stream"];
                    [self closeAll];
                } else if (bytesRead == 0) {

                } else {
                    [self.listData appendBytes:buffer
                        length:(NSUInteger) bytesRead];
                    [self parseListData];
                }
            }
            else if ([_currentOperation isEqualToString:@"GET"]){
                bytesRead = [_dataStream read:buffer maxLength:sizeof(buffer)];
                if (bytesRead == -1) {
                    [self.delegate ftpError:@"can't read data stream"];
                    [self closeAll];
                } else if (bytesRead == 0) {

                } else {
                    NSInteger bytesWritten;
                    NSInteger bytesWrittenSoFar;
                    bytesWrittenSoFar = 0;
                    do {
                        bytesWritten = [_downloadfileStream
                            write:&buffer[bytesWrittenSoFar]
                            maxLength:(NSUInteger)
                            (bytesRead - bytesWrittenSoFar)];
                        if (bytesWritten == -1) {
                            [self.delegate ftpDownloadFinishedWithSuccess:NO];
                            [self closeAll];
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

*continues*

**LISTING 8-12** (*continued*)

```

        } else {
            bytesWrittenSoFar += bytesWritten;
        }
    } while (bytesWrittenSoFar != bytesRead);
    [self closeAll];
    [self.delegate ftpDownloadFinishedWithSuccess:YES];
}
}

} break;

case NSStreamEventHasSpaceAvailable: {
    // If we don't have any data buffered, go read the next chunk of data.
    if (self.bufferOffset == self.bufferLimit) {
        NSInteger bytesRead;
        bytesRead = [_uploadStream read:self.buffer
            maxLength:kSendBufferSize];
        if (bytesRead == -1) {
            [self.delegate ftpError:@"can't read fileupload stream"];
            [self closeAll];
        } else if (bytesRead == 0) {

            } else {
                self.bufferOffset = 0;
                self.bufferLimit = bytesRead;
            }
        }
    // If we're not out of data completely, send the next chunk.
    if (self.bufferOffset != self.bufferLimit) {
        NSInteger bytesWritten;
        bytesWritten = [_commandStream
            write:&self.buffer[self.bufferOffset]
            maxLength:self.bufferLimit - self.bufferOffset];
        // assert(bytesWritten != 0);
        if (bytesWritten == -1) {
            [self.delegate ftpError:@"can't read data stream"];
            [self closeAll];
        } else {
            self.bufferOffset += bytesWritten;
        }
    }
} break;
case NSStreamEventErrorOccurred: {
    [self.delegate ftpError:@"stream open error"];
    [self closeAll];
} break;
case NSStreamEventEndEncountered: {
    // ignore
} break;
default: {
    assert(NO);
} break;
}
}

```

To use the `FTPManager` in your application, change the `YDViewController.h` file and import the `FTPManager.h` file in the header file and subscribe to the `FTPManagerDelegate` protocol. You also declare a property so your `YDViewController.h` file will look as shown in Listing 8-13.

**LISTING 8-13:** Chapter8/SimpleFTPClient/YDViewController.h

```
#import <UIKit/UIKit.h>
#import "FTPManager.h"
@interface YDViewController : UIViewController<FTPManagerDelegate>
{
    FTPManager* ftpmanager;
}
- (IBAction)uploadFile:(id)sender;
@end
```

To use the `FTPManager` you initialize the `FTPManager` with the IP address of the FTP server and provide a valid username and password in the `viewDidLoad` method.

You implement the delegate methods you require, create the user interface as you require, call the appropriate functions on the `FTPManager` instance, and off you go.

As you already know, you shouldn't run network operations on the main thread, and therefore it's best to create the `FTPManager` on a `dispatch_queue` with `DISPATCH_QUEUE_PRIORITY_BACKGROUND` so your user interface will not be blocked.

A sample implementation is shown in Listing 8-14.

**LISTING 8-14:** Chapter8/SimpleFTPClient/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController : UIViewController<FTPManagerDelegate>
{
    FTPManager* ftpmanager;
}
- (void)viewDidLoad
{
    [super viewDidLoad];

    dispatch_queue_t defQueue = dispatch_get_global_queue
        (DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0);
    dispatch_async(defQueue, ^{
        ftpmanager=[[FTPManager alloc] initWithServer:@"YOUR_SERVERNAME"
                                               user:@"YOUR_USERNAME"
                                             password:@"YOUR_PASSWORD!"];
        ftpmanager.delegate=self;
    });
}
```

*continues*

**LISTING 8-14 (continued)**

```

        });
    }
    -(IBAction)uploadFile:(id)sender
    {
        [ftpmanager listRemoteDirectory];
    }
    -(void)ftpDownloadFinishedWithSuccess:(BOOL)success
    {
        if (!success)
        {
            //handle your error
        }
    }
    -(void)ftpError:(NSString *)err
    {
        //handle your error
    }
    -(void)directoryListingFinishedWithSuccess:(NSArray *)arr
    {
        //use the array the way you need it
    }
    -(void)ftpUploadFinishedWithSuccess:(BOOL)success
    {
        if (!success)
        {
            //handle your error
        }
    }
    -(void)didReceiveMemoryWarning
    {
        [super didReceiveMemoryWarning];
    }
}
@end

```

You can add the created `FTPManager` class to the application framework you made in Chapter 1 so it's available when you develop a new application.

## Writing a Complex FTP Client

With the `FTPManager` you created you can perform basic operations like uploading and downloading a file, creating a directory on the remote server, and retrieving a directory listing, but that's all. If you need more control over the FTP process and/or to perform native operations, you need a different approach by implementing the FTP protocol on a lower level. In this example you learn how to write a complex FTP client that gives you full control over the FTP operations and processing the responses.

Start Xcode and create a new project using the Single View Application Project template, and name it ComplexFTPClient using the options shown in Figure 8-2.

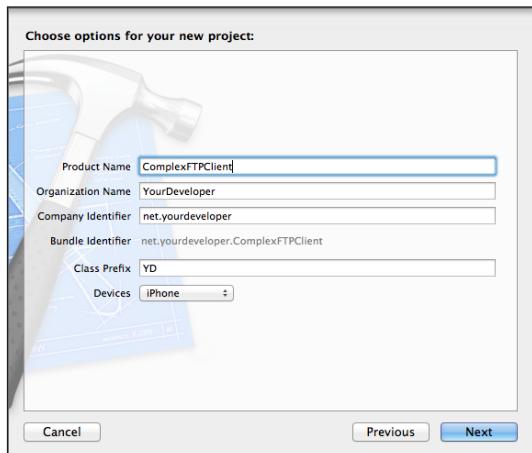


FIGURE 8-2

As a first step, again you add the CFNetwork framework to your project.

Create a new class that inherits from `NSObject` and name it `YDFTPClient`. Create the `YDFTPClientDelegate` protocol and the public methods as shown in Listing 8-15.

#### LISTING 8-15: Chapter8/ComplexFTPClient/YDFTPClient.h

```
#import <Foundation/Foundation.h>
@protocol YDFTPClientDelegate <NSObject>

-(void)loginFailed;
-(void)loggedOn;
-(void)serverResponseReceived:(NSString *)lastResponseCode
    message:(NSString *)lastResponseMessage;
-(void)ftpError:(NSString *)err;
@end

@interface YDFTPClient : NSObject<NSStreamDelegate>
@property (nonatomic, strong) id<YDFTPClientDelegate> delegate;
@property (readonly) UInt64 numberOfBytesSent;
@property (readonly) UInt64 numberOfBytesReceived;
- (id)initClient;
-(void)sendRAWCommand:(NSString *)command;
-(void)connect;
-(void)disconnect;

@end
```

The `YDFTPCClient.m` file consists of the private interface definition with its variables and the implementation of the different methods. Key logic in the `YDFTPCClient` is streams like in the previous project. You can find the full implementation of the `YDFTPCClient.m` in the downloads for this chapter. This is not a 100 percent implementation of all FTP commands and responses, but a skeleton class you can extend by implementing the commands you require.

Start with creating the properties in the private interface of the `YDFTPCClient` as shown in Listing 8-16.

#### LISTING 8-16: Private Interface of the YDFTPCClient class

```
@interface YDFTPCClient()
{
    UInt64 numberOfBytesSent;
    UInt64 numberOfBytesReceived;
    int uploadbytesreadSoFar;
}
@property (readonly, assign) NSString* dataIPAddress;
@property (readonly, assign) UInt16 dataPort;

@property (nonatomic, assign, readonly) uint8_t * buffer;
@property (nonatomic, assign, readwrite) size_t bufferOffset;
@property (nonatomic, assign, readwrite) size_t bufferLimit;

@property (nonatomic,assign) int lastResponseInt;
@property (nonatomic,assign) NSString* lastResponseCode;
@property (nonatomic,assign) NSString* lastCommandSent;
@property (nonatomic,assign) NSString* lastResponseMessage;

@property (nonatomic,retain, strong) NSInputStream *inputStream;
@property (nonatomic, retain,strong) NSOutputStream *outputStream;

@property (nonatomic, retain,strong) NSInputStream *dataInStream;
@property (nonatomic, retain,strong) NSOutputStream *dataOutStream;

@property (nonatomic,assign) BOOL isConnected;
@property (nonatomic,assign) BOOL loggedOn;
@property (nonatomic,assign) BOOL isDataStreamConfigured;
@property (nonatomic,assign) BOOL isDataStreamAvailable;

@end
```

So let's break down the implementation step by step. The `initClient` method is your custom initializer method that will initialize all the local variables and properties. The `initClient` method is shown in Listing 8-17.

#### LISTING 8-17: The initClient method

```
-(id)initClient
{
    if ((self = [super init]))
```

```

    {
        self.isConnected=NO;
        self.dataIPAddress=0;
        self.dataPort=0;
        self.isConnected=NO;
        self.isDataStreamAvailable=NO;
        self.lastCommandSent=@"";
        self.lastResponseCode=@"";
        self.lastResponseMessage=@"";
    }
    return self;
}

```

You define a connect and a disconnect method. The connect method simply calls the initNetworkCommunication: method where the disconnect method is calling the logoff: method as shown in Listing 8-18.

#### LISTING 8-18: The connect and disconnect methods

```

- (void)connect
{
    if (!self.isConnected)
        [self initNetworkCommunication];
}
-(void)disconnect
{
    if (self.isConnected)
        [self logoff];
}

```

The scheduleInCurrentThread: method is exactly the same as the one you used in the previous example. The initNetworkCommunication: method, which is being called by the connect: method, is responsible for creating the inputStream and outputStream, scheduling the streams on the current NSRunLoop, and opening them, as you can see in Listing 8-19.

#### LISTING 8-19: The initNetworkCommunication method

```

- (void) initNetworkCommunication {
    CFReadStreamRef readStream;
    CFWriteStreamRef writeStream;
    CFStreamCreatePairWithSocketToHost(NULL,
        (_bridge CFStringRef)kFTPServer,
        kFTPPort, &readStream, &writeStream);
    self.inputStream = (_bridge_transfer NSInputStream *)readStream;
    self.outputStream = (_bridge_transfer NSOutputStream *)writeStream;
    [self.inputStream setDelegate:self];
    [self.outputStream setDelegate:self];
    [self performSelector:@selector(scheduleInCurrentThread:)];
}

```

*continues*

**LISTING 8-19 (continued)**

```

        onThread:[ [self class] networkThread]
        withObject:self.inputStream
        waitUntilDone:YES];
[self performSelector:@selector(scheduleInCurrentThread:)
    onThread:[ [self class] networkThread]
    withObject:self.outputStream
    waitUntilDone:YES];
[self.inputStream open];
[self.outputStream open];
self.isConnected=YES;
self.isDataStreamConfigured=NO;
}

```

The `stream:handleEvent:` method is also in this implementation and is the key method that contains the control logic for reading and writing to the stream objects, as you can see in Listing 8-20. In the `NSStreamEventHasBytesAvailable` event, you read the bytes from the stream in the buffer, increment the `numberOfBytesReceived` property value you can use for network statistics, and send the read output to the `messageReceived:` method that will process it.

**LISTING 8-20: The `stream: handleEvent:` method**

```

- (void)stream:(NSStream *)theStream handleEvent:(NSStreamEvent)streamEvent {

    switch (streamEvent) {
        case NSStreamEventOpenCompleted:
            break;
        case NSStreamEventNone:
            break;
        case NSStreamEventHasBytesAvailable:
            if (theStream == self.inputStream) {
                uint8_t buffer[1024];
                int len;
                while ([self.inputStream hasBytesAvailable]) {
                    len = [self.inputStream read:buffer
                        maxLength:sizeof(buffer)];
                    numberOfBytesReceived+=len;
                    if (len > 0) {
                        NSString *output = [[NSString alloc]
                            initWithBytes:buffer
                            length:len
                            encoding:NSUTF8StringEncoding];
                        if (output) {
                            [self messageReceived:output];
                        }
                    }
                }
            }
            else if (theStream == self.dataInStream) {
                uint8_t buffer[8192];//8kB block
                int len;

```

```

        while ([self.dataInStream hasBytesAvailable]) {
            len = [self.dataInStream read:buffer
                maxLength:sizeof(buffer)];
            numberOfBytesReceived+=len;
            if (len > 0) {
                NSString *output = [[NSString alloc]
                    initWithBytes:buffer
                    length:len
                    encoding:NSUTF8StringEncoding];
                if (output) {
                    [self messageReceived:output];
                }
            }
        }
    }
    break;
case NSStreamEventHasSpaceAvailable:
    if (theStream == self.dataOutStream) {
        //write your custom code for upload and download
    }
    break;
case NSStreamEventErrorOccurred:
    [self.delegate ftpError:@"Network stream error occurred"];
    break;
case NSStreamEventEndEncountered:
    break;
}
}

```

The `messageReceived` method is a simple method that will keep track of your last response from the server, both as an integer as well as a code and a message, so you can refer the response code to the RFC protocol. The `messageReceived:` method is shown in Listing 8-21.

#### LISTING 8-21: The `messageReceived:` method

```

- (void) messageReceived:(NSString *)message {
    self.lastResponseCode = [message substringToIndex:3];
    self.lastResponseMessage=message;
    int response = [_lastResponseCode intValue];
    self.lastResponseInt=response;
    [self.delegate serverResponseReceived:
        self.lastResponseCode message:_lastResponseMessage];
    switch (response) {
        case 150:
            //connection accepted
            break;
        case 200:

            [self sendCommand:@"PASV"];
        case 220: //server welcome message so wait for username

```

*continues*

**LISTING 8-21** (*continued*)

```

        [self setUsername];
        break;
    case 226:
        //transfer OK
        break;
    case 227:
        [self acceptDataStreamConfiguration:message];
        break;
    case 230: //server logged in
        self.loggedOn=YES;
        [self sendCommand:@":PASV"];
        [self.delegate loggedOn];
        break;

    case 331: //server waiting for password
        [self setPassword];

        break;
    case 530: //Login or passwod incorrect
        [self.delegate loginFailed];
        self.loggedOn=NO;
        break;
    default:
        break;
    }
}
}

```

Notice in the code that most server implementations use a passive mode connection for security purposes. For that reason, when sending the `PASV` command to the FTP server, the response will contain the IP address and port number of the data socket that will be set up for the data exchange between the client and the server.

The `acceptDataStreamConfiguration` method is responsible for parsing the result using a regular expression. The first four groups of numbers are presenting the IP address and the last two are used to create the port number. So if the response is `a.b.c.d.x.y`, the port number is calculated by using the following formula:  $(x * 256) + y$ . The `acceptDataStreamConfiguration` method is shown in Listing 8-22.

**LISTING 8-22:** `acceptDataStreamConfiguration` method

```

- (void)acceptDataStreamConfiguration:(NSString*)serverResponse
{
    NSString *pattern=
    @"([-\\d]+),([-\\d]+),([-\\d]+),([-\\d]+),([-\\d]+),([-\\d]+)";
    NSError *error = nil;
    NSRegularExpression *regex = [NSRegularExpression
        regularExpressionWithPattern:pattern
        options:0
        error:&error];
}

```

```

NSTextCheckingResult *match = [regex firstMatchInString:
                             serverResponse
                             options:0
                             range:NSMakeRange(0,
                                           [serverResponse length])];

self.dataIPAddress = [NSString stringWithFormat:@"%@", 
                     [serverResponse substringWithRange:[match rangeAtIndex:1]],
                     [serverResponse substringWithRange:[match rangeAtIndex:2]],
                     [serverResponse substringWithRange:[match rangeAtIndex:3]],
                     [serverResponse substringWithRange:[match rangeAtIndex:4]]];
self.dataPort = ([[serverResponse substringWithRange:
                  [match rangeAtIndex:5]] intValue] * 256) +
[[serverResponse substringWithRange:[match rangeAtIndex:6]] intValue];
self.isDataStreamConfigured=YES;
[self openDataStream];

}

}

```

Finally, you need some life-cycle management code in your implementation as shown in Listing 8-23. The `openDataStream` creates an `inputstream` and an `outputstream` and uses the `scheduleInCurrentThread:` method to schedule them on the current `RunLoop`. The `closeDataStream` is properly closing and removing the data streams.

#### LISTING 8-23: `openDataStream`, `closeDataStream`, and `logoff` methods

```

-(void)openDataStream
{
    if (self.isDataStreamConfigured && !self.isDataStreamAvailable) {
        CFReadStreamRef readStream;
        CFWriteStreamRef writeStream;
        CFStreamCreatePairWithSocketToHost(NULL,
                                         (_bridge CFStringRef)self.dataIPAddress,
                                         self.dataPort, &readStream, &writeStream);
        self.dataInStream = (_bridge_transfer NSInputStream *)readStream;
        self.dataOutStream = (_bridge_transfer NSOutputStream *)writeStream;
        [self.dataInStream setDelegate:self];
        [self.dataOutStream setDelegate:self];
        [self performSelector:@selector(scheduleInCurrentThread:)
                      onThread:[[[self class] networkThread]
                      withObject:self.dataInStream
                      waitUntilDone:YES];
        [self performSelector:@selector(scheduleInCurrentThread:)
                      onThread:[[[self class] networkThread]
                      withObject:self.dataOutStream
                      waitUntilDone:YES];

        [self.dataInStream open];
        [self.dataOutStream open];
        self.isDataStreamAvailable=YES;
    }
}

```

*continues*

**LISTING 8-23 (continued)**

```
        }
    }
- (void)closeDataStream
{
    if (self.dataInStream.streamStatus != NSStreamStatusClosed)
    {
        [self.dataInStream removeFromRunLoop:
            [NSRunLoop currentRunLoop]
            forMode:NSUTFDefaultRunLoopMode];
        self.dataInStream.delegate = nil;
        [self.dataInStream close];
    }
    if (self.dataOutStream.streamStatus != NSStreamStatusClosed)
    {
        [self.dataOutStream removeFromRunLoop:
            [NSRunLoop currentRunLoop]
            forMode:NSUTFDefaultRunLoopMode];
        self.dataOutStream.delegate = nil;
        [self.dataOutStream close];
    }
}
- (void)logoff
{
    [self sendCommand:@":QUIT"];
    [self closeDataStream];
    if (self.inputStream.streamStatus != NSStreamStatusClosed)
    {
        [self.inputStream removeFromRunLoop:
            [NSRunLoop currentRunLoop]
            forMode:NSUTFDefaultRunLoopMode];
        self.inputStream.delegate = nil;
        [self.inputStream close];
    }
    if (_outputStream.streamStatus != NSStreamStatusClosed)
    {
        [self.outputStream removeFromRunLoop:
            [NSRunLoop currentRunLoop]
            forMode:NSUTFDefaultRunLoopMode];
        self.outputStream.delegate = nil;
        [self.outputStream close];
    }
    self.isConnected=NO;
    self.isDataStreamAvailable=NO;
    self.isDataStreamConfigured=NO;
}
```

## WORKING WITH AN FTP CLIENT

To work with the complex FTP client you developed, it's important to understand the sequence of sending FTP commands and processing the related FTP responses.

You can find the full RFC specifications at <http://www.w3.org/Protocols/rfc959>.

When the stream is opened, the server will respond with a 220 response and some kind of welcome message. Assuming your FTP server doesn't allow anonymous connections, the first command expected is the `USER` command followed by the username. If the username is correct, the server will respond with a 331 waiting for password response, prompting for a password. You send the password by using the `PASS` command followed by the password.

The `sendUsername` and `sendPassword` methods in the `YDFTPCClient` class are simple wrappers to demonstrate sending the username and password. The server response is either a 230 login succeeded or 530 login failed.

It's good practice to send the `PASV` command directly after the login that is asking the FTP server to use a passive connection, which is more secure than an active connection (always the same IP port). A passive connection is using a random port for the data stream, selected from within an available range of ports that are configured by the server.

You can add this `YDFTPCClient` class to the application framework you developed in Chapter 1.

## SUMMARY

In this chapter, you learned two different ways to implement an `FTPClient` using streams.

With the techniques you learned, you can:

- Set up stream connections to an FTP server
- Upload and download files to and from an FTP server
- Send raw FTP commands to an FTP server and process the responses

In the next chapter you learn how to use the Core Data framework in your application and all the techniques to benefit from this very powerful framework.



# 9

# Implementing Core Data

## **WHAT'S IN THIS CHAPTER?**

---

- Learning how to use Core Data
- Optimizing the use of the Core Data technology

## **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 9 download and individually named according to the names throughout the chapter.

## **INTRODUCTION TO CORE DATA**

Core Data is very often misunderstood by iOS developers because many inexperienced developers think Core Data is a replacement for a Relational Database Management System (RDBMS), which it's not.

Core Data is a framework that will bring with it:

- Object management
- Relationship management
- Change tracking and undo support
- Schema migrations
- Filtering, grouping, and organizing data in memory
- Sophisticated query compilation

Looking at these features it's easy to understand why many confuse Core Data with an RDBMS, and a lot of the functionality available within the Core Data framework looks like an RDBMS.

So what is Core Data? I define the Core Data framework as "An object-graph and persistence framework that allows you to create relational object models and provide the infrastructure to persist the object contents into a persistent store."

## Why Should You Use Core Data?

The Core Data framework is a highly optimized framework that has been severely tested using unit tests, and it is used daily by millions of applications worldwide. With Core Data you have a mature framework that will reduce your number of code lines for writing the model layer by 50 to 75 percent. You don't have to write the logic that comes with the framework, and an even bigger time saver is the fact you don't have to test your database code.

## Introducing Managed Object Context

You can think of a managed object context as a scratch pad with intelligent behavior. Once you retrieve a managed object (fetching) from a data store (persistent store), you have access to its properties, and you can modify these properties and relationships as required by your application logic. As long as you don't save the changes you have made to the managed object, the persistent store remains as is.

The managed object context is the access layer between your object graph and persistent store. The managed object context is generally created in the application's `application:didFinishLaunchingWithOptions:` method and used by a public property throughout your application. The managed object context is responsible for keeping track of all the changes you make to the managed objects, its properties, and relations between the moment you fetch the managed object(s) and save it to the persistent store. This also provides a great way to implement undo logic in your application, because the managed object context is managing all state changes of the managed object within its context.

## Introducing the Managed Object Model

A managed object model contains definitions for your entities, their properties, and their relations to other entities. The managed object model is an instance of the `NSManagedObjectModel`.

A managed object model contains entities, which are instances of the `NSEntityDescription` object. Each entity has attributes; these are instances of the `NSAttributeDescription` object and can have relationships with other entities, instances of the `NSRelationshipDescription` object. Entities can subclass another entity, which enables you to standardize your object model. For example, if you have two entities in which you need to persist address data, like an invoice address and a delivery address, you can create an address entity containing the properties for street, postal code, city, state, and country, and your invoice address can subclass the address entity. When creating entities you also create properties. A property can be an attribute or a relationship. A wide variety of attribute types are supported, such as string, date, and integer. If your entity requires an attribute that is not supported, you can define it as a transformable attribute. Say you want to create an entity that has

an attribute color, which is an instance of `NSColor`. Because `NSColor` is not supported by Core Data as an attribute type, you can define it as a transformable attribute type.

## Introducing Managed Objects

A managed object is an instance of the `NSManagedObject` class, or of a subclass of `NSManagedObject`, and represents an entity.

Managed objects are always associated with an entity description and with a managed object context. You shouldn't confuse a managed object with a row of a table because there can be multiple managed object contexts active, and a managed object represents a record in a persistent store in the managed object context.

This means that you can fetch managed object X using one context and update managed object Y based on the same fetch using a different context, and save the context. In such a scenario, the managed object X no longer equals the managed object Y.

## Introducing Persistent Stores

A persistent store is an object that is associated with a single file or external data store and contains your data. After you initialize the persistent store using a persistent store coordinator, which can manage multiple persistent stores simultaneously, you no longer work with the persistent store directly but you communicate with the managed object context, which sends messages to the persistent store coordinator, who follows by sending messages to the persistent store.

When creating a persistent store you can set a store-type. One often-used type is the `NSSQLiteStoreType`, which shouldn't be confused with a `SQLITE` database. The other available options are the `NSXMLStoreType`, `NSBinaryStoreType`, and `NSInMemoryStoreType`.

The available store-types are explained in detail in the official Apple documentation, which can be found here [https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/Articles/cdPersistentStores.html#/apple\\_ref/doc/uid/TP40002875-SW1](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/Articles/cdPersistentStores.html#/apple_ref/doc/uid/TP40002875-SW1).

## Introducing Fetch Requests

To retrieve data using a managed object context you create a fetch request, which is an object that specifies which information you want to retrieve from the managed object context.

Each fetch request should contain at least the entity that you want to retrieve, and you can specify only one entity in a fetch request. You can extend the fetch request with:

- An `NSPredicate` object, an object that specifies which criteria the entity must match
- An `NSArray` of `NSSortDescriptor` objects, specifying in which order the entity results should be returned

A fetch request is sent to the managed object context, which returns the objects that match your request from the persistent store(s) associated with the managed object context.

## USING CORE DATA IN YOUR APPLICATION

To demonstrate and explain the use of the Core Data framework, start Xcode and create a new project using the Single View Application Project template, and name it named `CoreDemo` using the configuration shown in Figure 9-1.

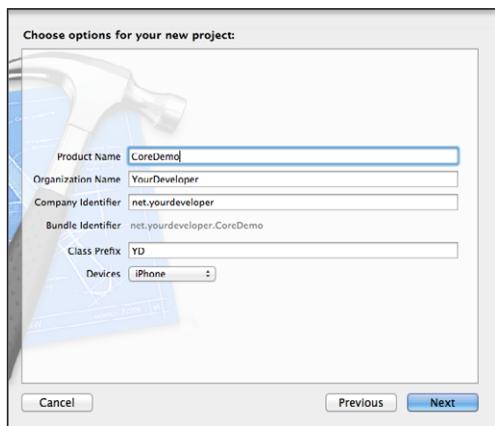


FIGURE 9-1

You can use the `CoreDemo` project to test and implement the code you'll learn in the following parts of this chapter.

Because you need to have the Core Data framework definition in many places of your applications, like your ViewControllers, it's convenient to edit your `CoreDemo-Prefix.pch` file and import the framework there so it's globally available, as shown in the following code:

```
#import <Availability.h>

#ifndef __OBJC__
#import <UIKit/UIKit.h>
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
#endif
```

## Creating a Managed Object Model

To create a managed object model, select `File` → `New` → `File`, select Core Data from the left category, and choose Data Model in the right pane that displays the various templates, as shown in Figure 9-2.

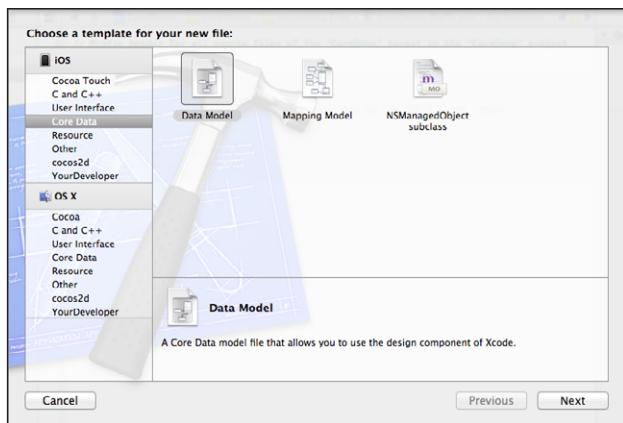


FIGURE 9-2

After you click the Next button, the Save As dialog box appears. Name the model `DemoModel` and save it, as shown in Figure 9-3.

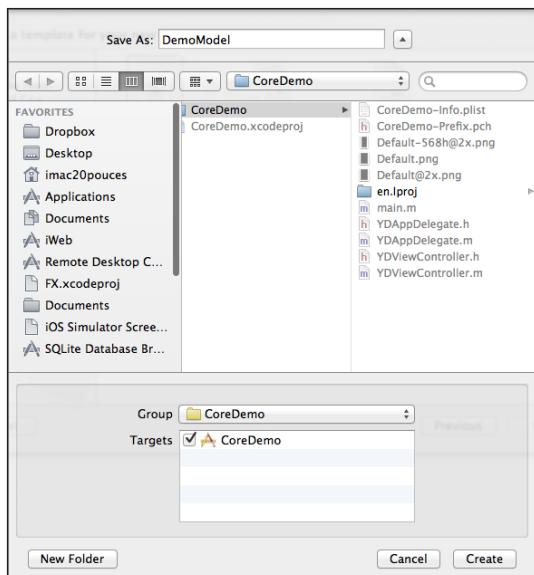


FIGURE 9-3

By default, the data model will be opened and you can see the user interface that enables you to modify the data model you just created.

Start by clicking the Add Entity button at the bottom, and you see a new entity created under the ENTITIES node in the middle pane, as shown in Figure 9-4.

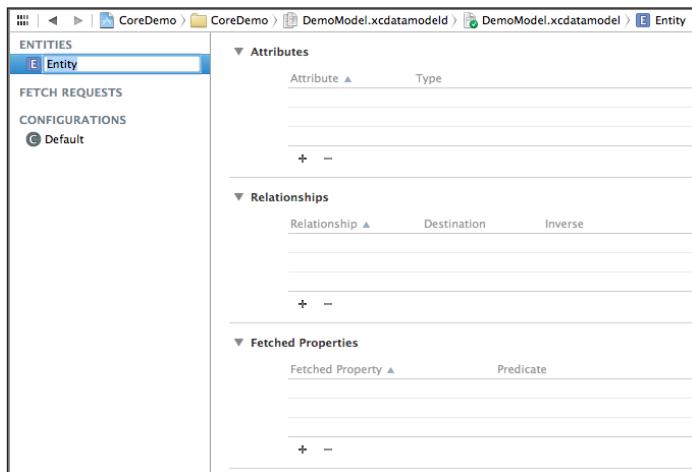


FIGURE 9-4

The entity is editable, so rename it to `Person`. Please be aware that entity names should always start with a capital letter.

In the right pane you see three sections: Attributes, Relationships, and Fetched Properties. Create some attributes for the `Person` entity of different types, as shown in Figure 9-5.

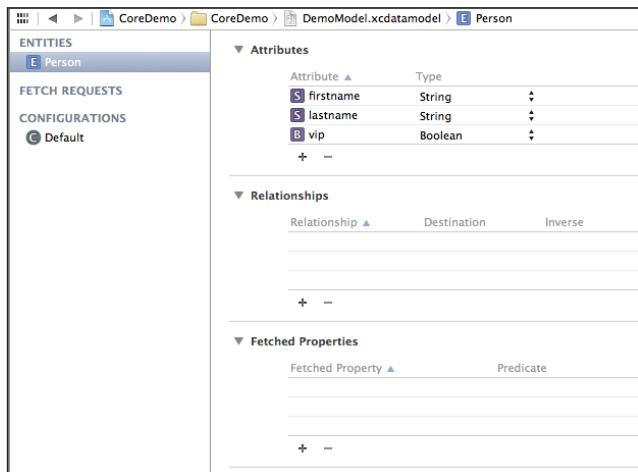


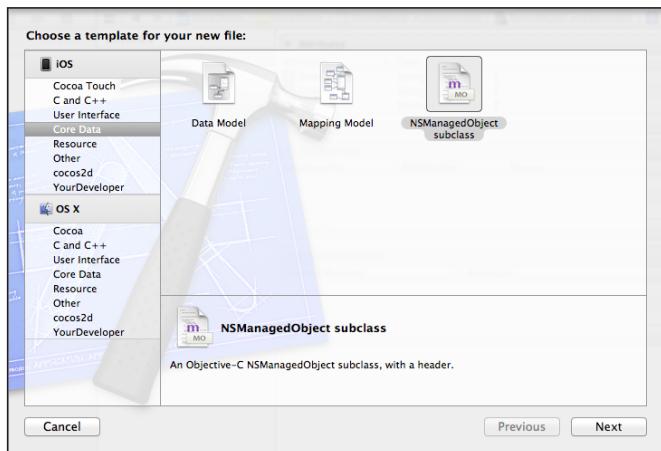
FIGURE 9-5

You now have a very basic managed object model with one entity named `Person` having three attributes: two of type `String` and one of type `Boolean`.

## Creating Managed Objects

Now you need to create the managed object for the Person entity you've just created.

In Xcode, select File ➤ New ➤ File and select Core Data from the category on the left. Choose NSManagedObject Subclass from the window at the right, as shown in Figure 9-6.



**FIGURE 9-6**

Click Next, and name and save the file that is created.

**NOTE** Depending on the changes you've made to the model, the screen as presented in Figure 9-6 might look different. If you have added entities to your model that have not been created as an NSManagedObject yet, you'll see an additional screen in which you select the model and the entities you want to create.

As a result of your last action, two files have been added to your project: Person.h, which is the header definition file, and Person.m, the implementation file for the NSManagedObject that relates to the entity Person.

The Person.h file will look like this:

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
@interface Person : NSManagedObject
@property (nonatomic, retain) NSString * lastname;
@property (nonatomic, retain) NSString * firstname;
@property (nonatomic, retain) NSNumber * vip;
@end
```

As you can see in the interface definition, the Person class is a subclass of NSManagedObject, and a property has been created for each attribute in the entity.

The `vip` property, which you defined as a Boolean, is created as an `NSNumber`.

When you make changes to the data model, you will need to “uninstall the application” from your device/simulator.

**NOTE** *Attributes of type Integer16, Integer32, Integer64, Decimal, Double, Float, and Boolean are created as an NSNumber. When using these properties, make sure you use NSNumber typecasts. You can also use a scalar data type.*

The `Person.m` file will look like this:

```
#import "Person.h"
@implementation Person
@dynamic lastname;
@dynamic firstname;
@dynamic vip;
@end
```

Each property is implemented using the `@dynamic` keyword.

The `@dynamic` keyword means that the responsibility of implementing the accessors is delegated. In this case to the `NSManagedObject` class.

## Creating Persistent Stores

Creating a persistent store is something you normally do only once in your application, at the moment of starting up and initializing.

You create a persistent store by setting up a persistent store coordinator and passing an `NSURL` object to the `NSPersistentStoreCoordinator` together with the `StoreType`.

A typical implementation method in the `AppDelegate` will look like the following code:

```
- (NSPersistentStoreCoordinator *)appStoreCoordinator
{
    if (self.persistentStoreCoordinator != nil) {
        return self.persistentStoreCoordinator;
    }
    NSURL *storeURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"demo.sqlite"];
    NSError *error = nil;
    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:[self appModel]];
    if (![self.persistentStoreCoordinator addPersistentStoreWithType:
        NSSQLiteStoreType configuration:nil
        URL:storeURL
        options:nil
        error:&error]) {
        abort();
    }
    return self.persistentStoreCoordinator;
}
```

## Setting Up Your Appdelegate

You can bring all the preceding together in the `CoreDemo` project you've created by opening the `YDAppDelegate.h` file and implementing the code as shown in Listing 9-1.

### LISTING 9-1: Chapter9/CoreDemo/YDAppDelegate.h

```
#import <UIKit/UIKit.h>
@class YDViewController;
@interface YDAppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) YDViewController *viewController;

@property (readonly, strong, nonatomic) NSManagedObjectContext *managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
    *persistentStoreCoordinator;
    //Helpers
- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;
@end
```

Three properties are defined—the `managedObjectContext`, the `managedObjectModel`, and the `persistentStoreCoordinator`—that will glue together the implementation. Next, there is a `saveContext` method that will actually save the `managedObjectContext`, and a helper method that returns the `applicationDocumentsDirectory` as an `NSURL` object.

You implement the `YDAppDelegate` class by initializing the `managedObjectContext` property by calling the `appContext` method. The `appContext` method is setting up the `NSPersistentStoreCoordinator` by calling the `appStoreCoordinator` method in which the URL is defined to the data store (in this example, `demo.sqlite`) and the managed object model is initialized by calling the `appModel` method. In this method the `NSURL` object is created for the resource `DemoModel.modm`. The complete implementation is shown in Listing 9-2.

### LISTING 9-2: Chapter9/CoreDemo/YDAppDelegate.m

```
#import "YDAppDelegate.h"
#import "YDViewController.h"
@implementation YDAppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    _managedObjectContext = [self appContext];
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    YDViewController* vc = [[YDViewController alloc]
        initWithNibName:@"YDViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
        initWithRootViewController:vc];
```

*continues*

**LISTING 9-2** (*continued*)

```
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];
    return YES;
}

#pragma mark - Core Data stack
- (NSManagedObjectContext *)appContext
{
    if (self.managedObjectContext != nil) {
        return self.managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self appStoreCoordinator];
    if (coordinator != nil) {
        _managedObjectContext = [[NSManagedObjectContext alloc] init];
        [_managedObjectContext setUndoManager:nil];
        [_managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return _managedObjectContext;
}

- (NSManagedObjectModel *)appModel
{
    if (self.managedObjectModel != nil) {
        return self.managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle]
        URLForResource:@"DemoModel" withExtension:@"momd"];
    _managedObjectModel = [[NSManagedObjectModel alloc]
        initWithContentsOfURL:modelURL];
    return self.managedObjectModel;
}

- (NSPersistentStoreCoordinator *)appStoreCoordinator
{
    if (self.persistentStoreCoordinator != nil) {
        return self.persistentStoreCoordinator;
    }
    NSURL *storeURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"demo.sqlite"];
    NSError *error = nil;
    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:[self appModel]];
    if (![self.persistentStoreCoordinator addPersistentStoreWithType:
        NSSQLiteStoreType configuration:nil
        URL:storeURL
        options:nil
        error:&error]) {
        abort();
    }

    return self.persistentStoreCoordinator;
}
```

```

- (NSURL *)applicationDocumentsDirectory
{
    return [[[[NSFileManager defaultManager] URLsForDirectory:
              NSDocumentDirectory inDomains:NSUserDomainMask]
              lastObject];
}

- (void)applicationWillResignActive:(UIApplication *)application
{
}
- (void)saveContext
{
    NSError *error = nil;
    NSManagedObjectContext *managedObjectContext = self.managedObjectContext;
    if (managedObjectContext != nil) {
        if ([self.managedObjectContext hasChanges] &&
            ![self.managedObjectContext save:&error]) {
            abort();
        }
    }
}
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [self saveContext];
}

- (void)applicationWillTerminate:(UIApplication *)application
{
    [self saveContext];
}
@end

```

The `saveContext` method is responsible for actually writing the managed object context to the persistent store.

It is good programming practice to call the `saveContext` method in the `applicationDidEnterBackground:` and `applicationWillTerminate:` methods to ensure all changes are written before the application will terminate or run in the background, so memory will be released.

## USING CORE DATA IN YOUR APPLICATION

Using Core Data in your application requires you to understand how to use managed objects and how to fetch them from the data store.

### Using Managed Objects

In this section you will learn how to perform operations on managed objects. You will learn how to create and delete managed objects and how to fetch managed objects.

## Creating and Deleting Managed Objects

You create a new managed object, like a new Person, by following these steps:

1. Create an instance of the object with the `NSEntityDescription insertNewObjectForEntityForName: inManagedObjectContext:` method.
2. Set the property values of the created managed object.
3. Call the `save` method of the managed object context that is used to create the object.

The following code creates a new Person managed object, sets the property values, and calls the `save` method of the managed object context that is used to create the object:

```
Person* newPerson = [NSEntityDescription
                     insertNewObjectForEntityForName:@"Person"
                     inManagedObjectContext:
                     [self appDelegate].managedObjectContext];
newPerson.firstname=_firstnameField.text;
newPerson.lastname=_lastnameField.text;
newPerson.vip = [NSNumber numberWithBool:_isVip.isOn];
NSError *error = nil;
if (![[self appDelegate].managedObjectContext save:&error]) {
    //handle your error
}
```

Alternatively, you can create an `NSManagedObject` instead of the `Person` object and use the `setValue:` method to set the specific values for the specific keys:

```
NSManagedObject* newObject= [NSEntityDescription
                             insertNewObjectForEntityForName:@"Person"
                             inManagedObjectContext:
                             [self appDelegate].managedObjectContext];
[newObject setValue:_firstnameField.text forKey:@"firstname"];
[newObject setValue:_lastnameField.text forKey:@"lastname"];
[newObject setValue:[NSNumber numberWithBool:_isVip.isOn] forKey:@"vip"];
NSError *error = nil;
if (![[self appDelegate].managedObjectContext save:&error]) {
    //handle your error
}
```

To delete a managed object, you simply call the `deleteObject:` method on the managed object context and pass the managed object variable you want to delete. The actual delete operation is only performed after you call the `save:` method on the managed object context, which performs the actual update on the persistent store. Sample code is shown here:

```
NSError *error=nil;
[[self appDelegate].managedObjectContext deleteObject:THEOBJECT];
if (![[self appDelegate].managedObjectContext save:&error]) {
//handle your error
}
```

## Fetching Managed Objects

Now that you know how to set up your app delegate, initialize the managed object context, persistent store, and the object model, and can insert and delete managed objects, it's time to fetch the objects from the managed object context and display the fetched results in a UITableView.

Open the YDViewController.h file and implement the code as shown in Listing 9-3.

LISTING 9-3: Chapter9/CoreDemo/YDViewController.h

```
#import <UIKit/UIKit.h>
@interface YDViewController : UIViewController
{
    NSMutableArray* people;
}
@property(nonatomic,strong) IBOutlet UITableView* mTableView;
@end
```

Now open the YDViewController.xib file using Interface Builder and the Assistant Editor to create a user interface, as shown in Figure 9-7.

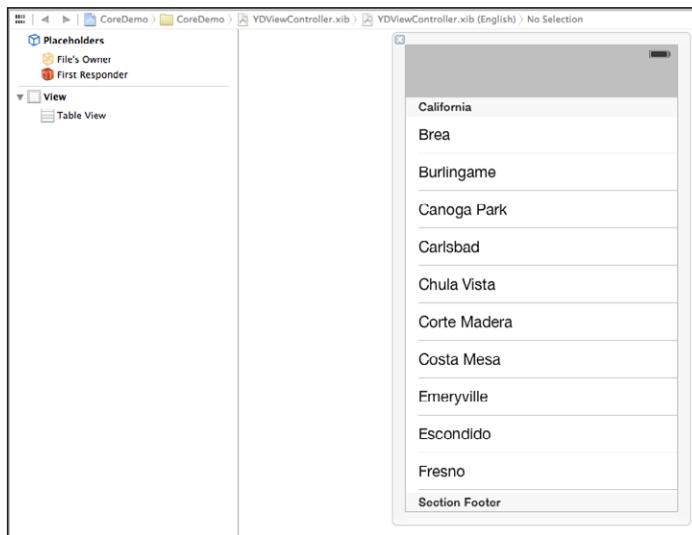


FIGURE 9-7

Different methods are available for fetching the managed objects from the managed object context, such as using an NSFetchedResultsController. I'll show you different methods you can use depending on your personal preferences and requirements.

I prefer to work with an `NSMutableArray` and type cast the fetched data to the managed object type I require, because I prefer to work with objects and properties. A sample implementation might look like this:

```
- (void)loadPeople
{
    if (people)
        people=nil;
    people=[[NSMutableArray alloc] init];
    NSEntityDescription *entity = [NSEntityDescription
        entityForName:@"Person"
        inManagedObjectContext:
        [self appDelegate].managedObjectContext];
    NSFetchedResultsController *fetchRequest = [[NSFetchedResultsController alloc] init];
    [fetchRequest setEntity:entity];
    NSError *error = nil;
    NSArray *fetchedObjects = [[self appDelegate].managedObjectContext
        executeFetchRequest:fetchRequest error:&error];
    for(int i = 0;i<[fetchedObjects count];i++)
    {
        Person *obj = (Person *)[fetchedObjects objectAtIndex:i];
        [people addObject:obj];
    }
    [self.tableView reloadData];
}
```

In the `loadPeople` method, an `NSMutableArray` is created and initialized. Next, an `NSEntityDescription` object named `entity` is created, which is initialized with the `entityForName:inManagedObjectContext:` method using the value of your managed object subclass, in this case `Person`.

An `NSFetchRequest` is created and initialized, and the earlier created entity is set. An `NSArray` named `fetchedObjects` is created by calling the `executeFetchRequest:` method on a managed object context, in this case the one you created in the `YDAppDelegate`.

You loop of all the objects in the `fetchedObjects` array, type cast them to the `Person` object, and add them to the `people` array. Finally, you call your `UITableView` instance's `reloadData` method.

An alternative method is to use an `NSFetchedResultsController` object.

To use the `NSFetchedResultsController`, change the `YDViewController.h` file as shown in the following code:

```
#import <UIKit/UIKit.h>
@interface YDViewController : UIViewController
{
    NSFetchedResultsController *fetchedResultsController;
}
@property(nonatomic,strong) IBOutlet UITableView* mTableView;
@end
```

Open the `YDViewController.m` file and implement the following code:

```
@interface YDViewController ()<NSFetchedResultsControllerDelegate>
@end
```

```

- (NSFetchedResultsController *)fetchedResultsController
{
    if (fetchedResultsController == nil)
    {
        NSEntityDescription *entity =
            [NSEntityDescription entityForName:@"Person"
                inManagedObjectContext:[self appDelegate].managedObjectContext];
        NSSortDescriptor *descriptor1 = [[NSSortDescriptor alloc]
            initWithKey:@"firstname" ascending:YES];
        NSSortDescriptor *descriptor2 = [[NSSortDescriptor alloc]
            initWithKey:@"lastname" ascending:YES];
        NSArray *sortDescriptors = [NSArray arrayWithObjects:
            descriptor1, descriptor2,nil];
        NSFetchedResultsController *fetchedResultsController = [[NSFetchedResultsController alloc]
            initWithFetchRequest:fetchRequest
            managedObjectContext:[self appDelegate].managedObjectContext
            sectionNameKeyPath:@""
            cacheName:nil];
        [fetchedResultsController setDelegate:self];
        NSError *error = nil;
        if (![fetchedResultsController performFetch:&error])
        {
            //handle the error
        }
    }
    return fetchedResultsController;
}
#pragma delegate is called when the controller content did change
- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
{
    [self.tableView reloadData];
}

```

The preceding method is created and initializes the `NSFetchedResultsController` instance. Again, an entity and two `NSSortDescriptors` are created. An `NSSortDescriptor` is taking the property name on which you want to sort as an argument. An `NSArray` named `sortDescriptors` is initialized with the `NSSortDescriptor` object you just created, and with the `setSortDescriptors:` method they are set on the `NSFetchRequest`. The `sectionNameKeyPath` property is convenient if you want to use different sections in your `UITableView` because the `NSFetchedResultsController` will automatically section the fetched results.

If you use the `NSFetchedResultsController` technique you also need to make changes to the `UITableView` delegates to return the number of sections and number of rows in a section, as presented in the following code:

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [[self fetchedResultsController] sections] count];
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {

```

```

        id <NSFetchedResultsSectionInfo> sectionInfo =
            [[self.fetchedResultsController sections] objectAtIndex:section];
        return [sectionInfo numberOfObjects];
    }
}

```

## Fetching Specific Values

Assume you created a managed object named `product` that has a property for product group and a property for product name. You decided to use a single managed object instead of two managed objects with a relationship for whatever reason. In your application's logic you require an `NSArray` that contains all the possible product groups from your managed object context to populate a `UIPickerView` or a `UITableView`.

To support this requirement, the `NSFetchRequest` has a property named `setPropertiesToFetch:` that accepts an `NSArray` of the properties you want to receive in the results. This property works in combination with the `resultType` property that indicates the result type.

**WARNING** *You need to set the entity on the NSFetchRequest before you set the PropertiesToFetch, otherwise the application will crash.*

The following constants are available to set as a `resultType`:

CONSTANT	BEHAVIOR
<code>NSManagedObjectResultType</code>	Specifies that the request returns managed objects
<code>NSManagedObjectIDResultType</code>	Specifies that the request returns managed object IDs
<code>NSDictionaryResultType</code>	Specifies that the request returns dictionaries
<code>NSCountResultType</code>	Specifies that the request returns the count of the objects that match the request

When using the `propertiesToFetch:` limitation wisely, your application's memory footprint can be reduced significantly if you are using managed objects with many properties, which you don't need all at once.

If the returned results are presented in a `UITableView` or a `UIPickerView`, you will see that the returned values are not unique, but all values will be presented.

## Fetching Unique Objects

In a SQL language, if you wanted to see only distinct results displayed, you would write something like `SELECT DISTINCT PRODUCTGROUP FROM PRODUCTS`, but Core Data is not an RDBMS so you can't write queries like this.

Fortunately, Core Data has a property on the `NSFetchRequest` object named `returnDistinctResults`, which is a `BOOL` that works in combination with the previously explained properties-`ToFetch` property.

Simply add the following code line to your logic just before you execute `executeFetchRequest:` and you will receive the distinct results:

```
[fetchRequest setReturnsDistinctResults:YES];
```

## Fetching Specific Objects

Predicates provide a general solution for specifying queries in Cocoa. The `NSPredicate` class is used to define logical conditions to constrain or filter the fetch results.

For example, you only want to fetch the managed objects of people that have the `VIP` property set to `YES`, an age above 35 years, and a function title containing the word “management.”

You would add the following code to your implementation just before executing the `executeFetchRequest:` method:

```
NSString *functionName = @"management";
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"vip == YES AND
    age > 35 AND
    function CONTAINS[cd] %@", functionName];
[fetchRequest setPredicate:predicate];
```

For string comparisons, the following keywords are available:

KEYWORD	EFFECT
BEGINSWITH	The left-hand expression begins with the right-hand expression.
CONTAINS	The left-hand expression contains the right-hand expression.
ENDSWITH	The left-hand expression ends with the right-hand expression.
MATCHES	The left-hand expression equals the right-hand expression using a regex-style comparison according to ICU v3.
LIKE	The left-hand expression equals the right-hand expression: ? and * are allowed as wildcard characters, where ? matches one character and * matches zero or more characters.

After each of the preceding keywords, you can add `c` or `cd` between square brackets, as in `CONTAINS[cd]`, where the `c` stands for case sensitive and the `d` stands for diacritic sensitive.

The `CoreDemo` project you created is not a complete working application; it is only used to provide you with a basic project in which the different code snippets from this chapter could be implemented to test and understand the different techniques.

## Using Relationships

To understand how to work with Relationships in Core Data, start Xcode and create a new project using the Single View Application Project template, and name it `CoreRelationship` using the configuration shown in Figure 9-8.

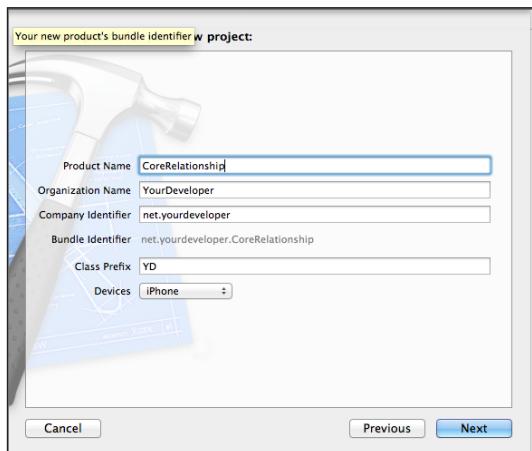


FIGURE 9-8

In this project you create two different `NSManagedObjects` with a Relationship between the two.

Start by adding the Core Data framework to your project. Add the import of the Core Data framework to the `CoreRelationship-Prefix.pch` file so you don't have to import it in each and every class.

For the purpose of this example, you will create a new managed object model with two entities that have a Relationship between them. The first entity will be a continent and the second entity will be a country. The relationship will relate a country to a continent.

Create a new data model as shown earlier in Figure 9-2 and name it `World`. Create an entity named `Continent` with one attribute called `name` of type `String`, and a second entity named `Country`, again with one attribute called `name` of type `String`.

For this example the relationship between the two entities is that each `Continent` has one-to-many `Country` objects and each `Country` belongs to only one `Continent`.

In the Model Editor, select the `Continent` entity and add a Relationship named `countries` and set the Destination to `Country`. In the right pane select Plural To-Many Relationship and set the Delete Rule to Cascade, as shown in Figure 9-9.

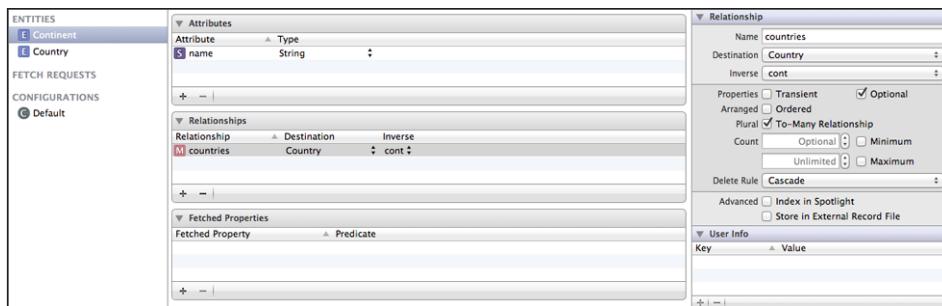


FIGURE 9-9

Select the Country entity and create a relationship named `cont` and set the Destination to Continent and set the inverse to `countries` as shown in Figure 9-10.

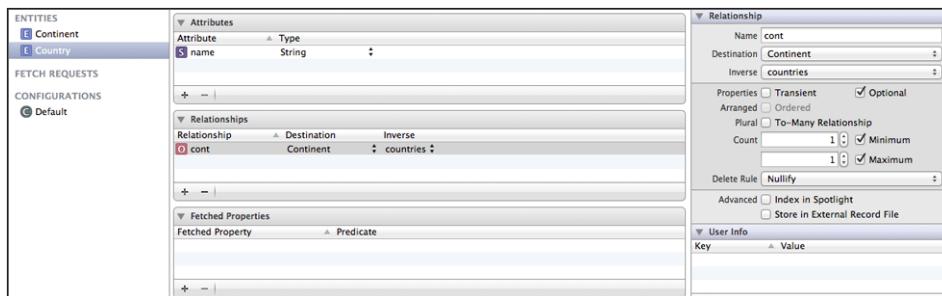


FIGURE 9-10

Now you have established a relationship between two entities. The delete rule (Cascade as you specified on the `Continent` entity) is performing a cascade delete. This means that if you delete a `Continent` object, all related `Country` objects are also deleted from the data store. See Table 9-1.

TABLE 9-1

DELETE RULES	EFFECT
Deny	If there is at least one object at the relationship destination, then the source object cannot be deleted.
Nullify	Set the inverse relationship for objects at the destination to null.
Cascade	Delete the objects at the destination of the relationship.
No Action	Do nothing to the object at the destination of the relationship.

Create the `NSManagedObject` subclasses for the `Continent` and the `Country` object using the `File`  $\Rightarrow$  `New`  $\Rightarrow$  `File` option from the Xcode menu. An additional screen will appear that looks similar to Figure 9-11, in which you choose the data model you are working on.

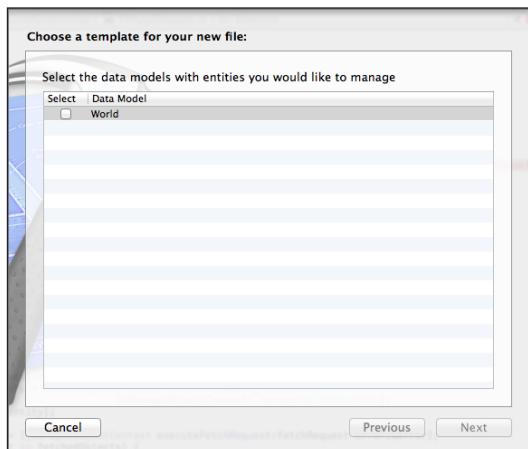


FIGURE 9-11

Because multiple managed objects are available in the data model, after you click Next a screen will appear that looks like Figure 9-12 asking you to select the entities for which you want to create the `NSManagedObject` subclass. Select both the entities, click Next, and save them to your project folder.

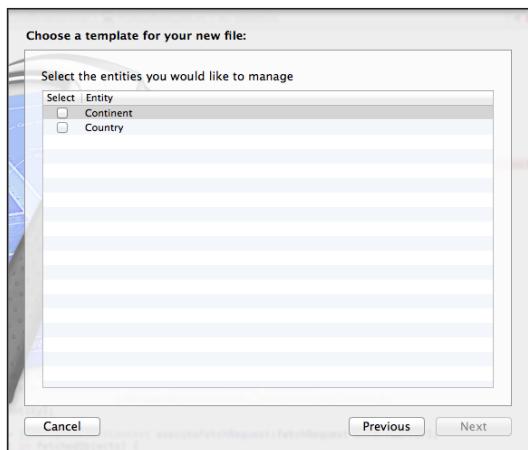


FIGURE 9-12

Because of the defined relationship, the generated `Continent` class has some additional methods defined, as you can see in Listing 9-4.

**LISTING 9-4: Chapter9/CoreRelationShip/Continent.h**

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>
@class Country;
@interface Continent : NSManagedObject
@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSSet *countries;
@end
@interface Continent (CoreDataGeneratedAccessors)
- (void)addCountriesObject:(Country *)value;
- (void)removeCountriesObject:(Country *)value;
- (void)addCountries:(NSSet *)values;
- (void)removeCountries:(NSSet *)values;
@end
```

In the following example you can see how to create a new `Continent` and some `Country` managed objects and set the relationship between them:

```
Continent* continent = [NSEntityDescription
                        insertNewObjectForEntityForName:@"Continent"
                        inManagedObjectContext:_managedObjectContext];
continent.name = @"Europe";

Country* france = [NSEntityDescription
                    insertNewObjectForEntityForName:@"Country"
                    inManagedObjectContext:_managedObjectContext];
france.name = @"France";
//here we set the relationship
france.cont = continent;
//another country
Country* spain = [NSEntityDescription
                    insertNewObjectForEntityForName:@"Country"
                    inManagedObjectContext:_managedObjectContext];
spain.name = @"Spain";
//here we set the relationship
spain.cont = continent;
//and one more
Country* unitedkingdom = [NSEntityDescription
                            insertNewObjectForEntityForName:@"Country"
                            inManagedObjectContext:_managedObjectContext];
unitedkingdom.name = @"United Kingdom";
//here we set the relationship
unitedkingdom.cont = continent;
//Save them
NSError *error = nil;
if (![_managedObjectContext save:&error]) {
    //handle error
}
```

In this program the continents are shown in a `UITableView` and when one is selected, the selected continent is passed to a `DetailViewController` that will perform an `NSFetchRequest` and display the

Country objects for the selected continent. In the `[predicateWithFormat]` method the attribute `cont` is checked against the passed property value `thisContinent`, as you can see in the following code:

```
NSEntityDescription *entity = [NSEntityDescription
                               entityForName:@"Country"
                               inManagedObjectContext:
                               [self appDelegate].managedObjectContext];
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
[fetchRequest setEntity:entity];
NSPredicate* pred = [NSPredicate predicateWithFormat:
                     @"cont == %@", self.thisContinent];
[fetchRequest setPredicate:pred];
NSError *error = nil;
NSArray *fetchedObjects = [[self appDelegate].managedObjectContext
                           executeFetchRequest:fetchRequest error:&error];
```

You can download the complete source code for this example from the website under `Chapter9/CoreRelationship`.

## Understanding Model Changes

During the life cycle of your application, model changes might be applicable because you might need to add an entity to your model, or perhaps add an attribute to an already existing entity.

A very important restriction you need to know and respect is the fact that you can only open a Core Data store using the managed object model used to create it. When you add an attribute to the model, re-generate the `NSManagedObject` subclasses, and launch your application without any additional action, it will crash.

If you make changes to your data model, you need to change the data in the existing store(s) to the new version of the data model. This phenomenon is known as *migration*.

To perform a migration of a data store, you therefore need both the version of the data model used to create it, and the current version of the model you want to migrate to.

Core Data needs to know how to map entities and attributes in a source model to entities and attributes in the destination model. In many cases, Core Data can infer the mapping from existing versions of the managed object model; this is known as *lightweight migration*, where with more complicated model design changes, Core Data needs to have a mapping model in which it's specified how to transform entities and attributes in the source model into entities and attributes in the destination model.

## Understanding Lightweight Migration

Core Data's lightweight migration can process simple changes to your data model, such as adding a new attribute to an existing entity. Lightweight migration is fundamentally the same as ordinary migration, with the difference that a mapping model is not required because Core Data can infer the mapping from the data model.

During the lightweight migration process, Core Data looks for models in the bundles returned by the `NSBundle allBundles` result. For that reason, it's common practice to store your data models in your application bundle.

A lightweight migration is carried out by generating an inferred mapping model, where model changes must fit one of the following scenarios:

- Adding a new attribute to an existing entity
- Removing an attribute from an entity
- Changing an attribute from optional to non-optional and vice versa
- Renaming an entity
- Renaming an attribute
- Adding a new relationship
- Deleting an existing relationship
- Renaming a relationship

When renaming an entity, relationship, or attribute, you need to set the renaming identified in the destination model to the name of the corresponding object in the source model using the property inspector.

The following code explains how to request lightweight migration by using an options dictionary with predefined values. You change the code in your `YAppDelegate.m` to the following code, as shown here:

```
- (NSPersistentStoreCoordinator *)appStoreCoordinator
{
    if (self.persistentStoreCoordinator != nil) {
        return self.persistentStoreCoordinator;
    }

    NSURL *storeURL = [[self applicationDocumentsDirectory]
        URLByAppendingPathComponent:@"world.sqlite"];
    NSError *error = nil;
    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:[self appModel]];

    NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
        [NSNumber numberWithBool:YES],
        NSMigratePersistentStoresAutomaticallyOption,
        [NSNumber numberWithBool:YES],
        NSInferMappingModelAutomaticallyOption, nil];

    if (![self.persistentStoreCoordinator
        addPersistentStoreWithType:NSSQLiteStoreType
        configuration:nil
        URL:storeURL
        options:options
        error:&error]) {
        abort();
    }

    return self.persistentStoreCoordinator;
}
```

During the development stage of your application, it's good to know that if you delete the created data store from your simulator's application directory and relaunch the application, a new store is created using the latest data model.

## Versioning a Data Model

Your data model requirements might change during the development cycles of your application, resulting in a situation where you want to change your data model. To support a smooth migration, either by lightweight migration or by using a mapping model, it's important to use model versions.

Select your data model in the project navigator and choose Editor ⇔ Add Model Version.

In the pop-up window, name your version, use the Based on Model drop-down to select the correct model, and click Finish as shown in Figure 9-13.

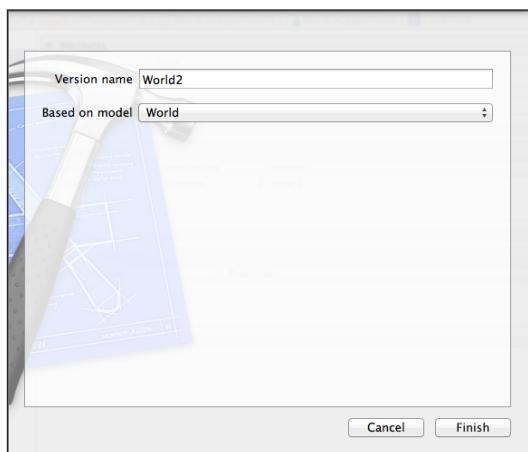


FIGURE 9-13

A new version of the model is created, and if you unfold the `World.xcdatamodeld` node in the project navigator, you'll find the World and the World2 version of your data model.

In the right-hand pane under the Versioned Core Data Model node you see a Current drop-down, which you can change from World to World2 to make the World2 version the current version, as shown in Figure 9-14.



FIGURE 9-14

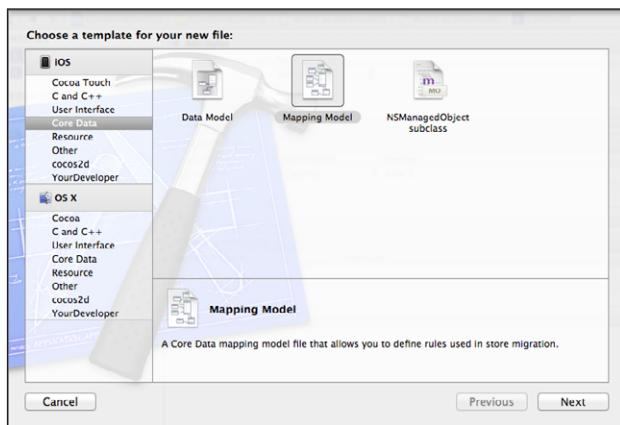
Make the change in the World2 version of the model you require; for example, by adding an attribute named `isocode` to the `Country` entity.

If you now build and run the application, it will launch without problems. You've created a new version of your model and both versions are accessible from the application bundle, so lightweight migration is performed.

## Using a Mapping Model

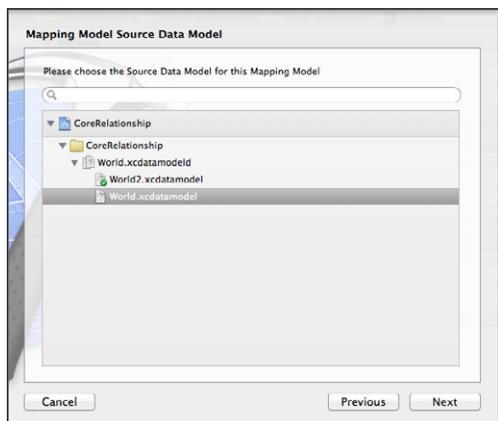
When your new version of the model has more complicated changes, you can create a mapping model in which you specify how the source model maps to the destination model.

To create a mapping model, select `File` → `New` → `File` from the Xcode menu, select the `Mapping Model` template as shown in Figure 9-15, and click `Next`.

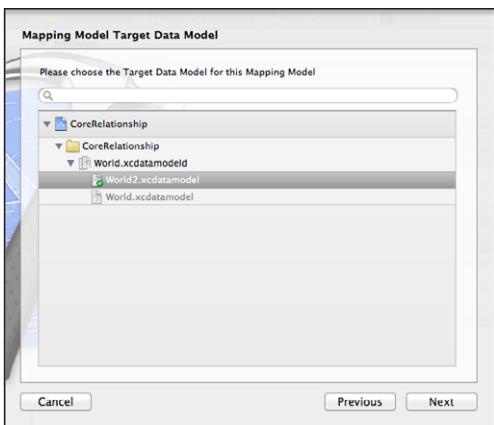


**FIGURE 9-15**

In the subsequent dialog box, select the source model and click `Next` to select the destination model, as shown in Figures 9-16 and 9-17.

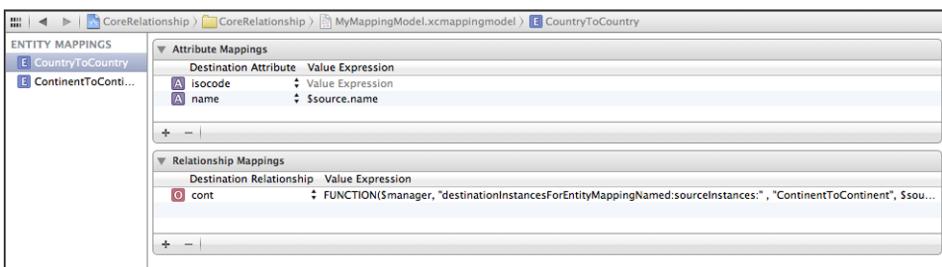


**FIGURE 9-16**

**FIGURE 9-17**

Finally, in the Save As dialog box name the mapping model `MyMappingModel` and save it under your project folder.

Xcode now has added the `MyMappingModel.xcMappingModel` file to your project navigator, and when you select this file it will open the mapping editor as shown in Figure 9-18.

**FIGURE 9-18**

You can map the entities, attributes, and relationships from your source model to your destination model.

## TUNING FOR PERFORMANCE

It's important that the performance of your application is in line with a user's expectations. When an application loads data and it's taking a long time, many users will just close the application and not use it anymore.

Performance can be defined as speed, as in how long it takes to perform a certain action, but also as a memory footprint, as in how much memory is used for a certain action.

When you are using Core Data in your application, most of the time you choose to do so because you need to process a significant amount of data. Performance is a subjective word, because what one user would consider fast enough might be too slow for another user.

When using Core Data in your application, you can apply several techniques to improve the performance of your application.

In this section you learn several techniques that will have an effect on the memory footprint and/or the performance of your application. For the purpose of demonstration, let's create a simple application without a lot of user interface elements to learn these techniques.

Start Xcode and create a new project using the Single View Application Project template, and name it CorePerformance using the settings shown in Figure 9-19.

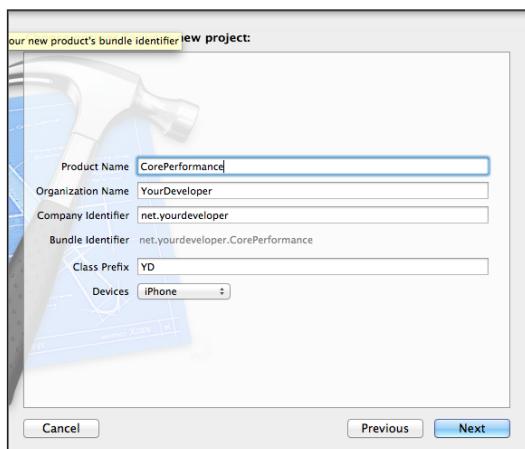


FIGURE 9-19

Open the CorePerformance-Prefix.pch file and import the Core Data framework so it's globally available in your project.

Add the Core Data framework to your project and implement the code in Listing 9-5 in your YAppDelegate.h file.

#### LISTING 9-5: Chapter9/CorePerformance/YAppDelegate.h

```
#import <UIKit/UIKit.h>
@class YDViewController;
@interface YAppDelegate : UIResponder <UIApplicationDelegate>
@property (readonly, strong, nonatomic) NSManagedObjectContext *managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
    *persistentStoreCoordinator;
@property (strong, nonatomic) UINavigationController *navigationController;
//Helpers
```

*continues*

**LISTING 9-5** (*continued*)

```

- (void)saveContext;
- (void)resetStore;
- (NSURL *)applicationDocumentsDirectory;
@end

```

Open your YAppDelegate.m file and implement the standard code for setting up a Core Data implementation, as shown in Listing 9-6.

**LISTING 9-6:** Chapter9/CorePerformance/YAppDelegate.m

```

#import "YAppDelegate.h"
#import "YDViewController.h"
@implementation YAppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    _managedObjectContext = [self appContext];
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    YDViewController* vc = [[YDViewController alloc]
                           initWithNibName:@"YDViewController" bundle:nil];
    self.navigationController = [[UINavigationController alloc]
                               initWithRootViewController:vc];
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];
    return YES;
}
#pragma mark - Core Data stack
- (NSManagedObjectContext *)appContext
{
    if (_managedObjectContext != nil) {
        return _managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self appStoreCoordinator];
    if (coordinator != nil) {
        _managedObjectContext = [[NSManagedObjectContext alloc] init];
        [_managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return _managedObjectContext;
}
- (NSManagedObjectModel *)appModel
{
    if (_managedObjectModel != nil) {
        return _managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle]
                        URLForResource:@"Performance"
                        withExtension:@"momd"];
    _managedObjectModel = [[NSManagedObjectModel alloc]
                          initWithContentsOfURL:modelURL];
    return _managedObjectModel;
}

```

```

    }

    - (NSPersistentStoreCoordinator *)appStoreCoordinator
    {
        if (self.persistentStoreCoordinator != nil) {
            return self.persistentStoreCoordinator;
        }

        NSURL *storeURL = [[self applicationDocumentsDirectory]
                            URLByAppendingPathComponent:@"Performance.sqlite"];
        NSError *error = nil;
        _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                                       initWithManagedObjectModel:[self appModel]];
        if (![self.persistentStoreCoordinator
              addPersistentStoreWithType:NSSQLiteStoreType
              configuration:nil
              URL:storeURL
              options:nil
              error:&error]) {

            abort();
        }
    }

    return self.persistentStoreCoordinator;
}
-(void)resetStore
{
    //Set the managed object context to nil;
    _managedObjectContext = nil;
    NSURL *storeURL = [[self applicationDocumentsDirectory]
                        URLByAppendingPathComponent:@"Performance.sqlite"];
    NSError *error = nil;
    //drop the sqlite file
    [[NSFileManager defaultManager] removeItemAtURL:storeURL error:&error];
    //reinitialize the managed object recreating the store
    _managedObjectContext = [self applicationContext];
}
-(NSURL *)applicationDocumentsDirectory
{
    return [[[[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
                                                     inDomains:NSUTFUserDomainMask] lastObject];
}

-(void)applicationWillResignActive:(UIApplication *)application
{
}
-(void)saveContext
{
    NSError *error = nil;
    NSManagedObjectContext *managedObjectContext = _managedObjectContext;
    if (managedObjectContext != nil) {
        if ([_managedObjectContext hasChanges] &&
            ![_managedObjectContext save:&error]) {

            abort();
        }
    }
}

```

*continues*

**LISTING 9-6** (continued)

```

        }
    }
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [self saveContext];
}
- (void)applicationWillEnterForeground:(UIApplication *)application
{
}
- (void)applicationDidBecomeActive:(UIApplication *)application
{
}
- (void)applicationWillTerminate:(UIApplication *)application
{
    [self saveContext];
}
@end

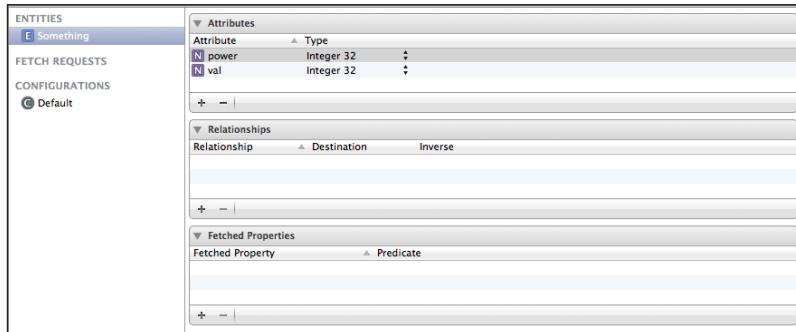
```

A method named `resetStore` has been added, which simply deletes the `sqlite` file using the `NSFileManager` and re-initializes the managed object context to re-create the `sqlite` store. This is a simple yet effective and fast way to delete a complete data store and all its contents.

The preceding implementation uses a data model named `Performance`, so let's create it now.

For the purpose of this chapter, a simple entity with some simple attributes is sufficient to demonstrate the performance effects of the different configuration techniques you'll apply in the following sections.

Create a new entity and name it `Something`. Add two attributes of type `int32`—name one `val` and the other one `power`, as shown in Figure 9-20.



**FIGURE 9-20**

Create the `NSManagedObject` subclass for the entity named `Something`. The attribute `val` will be used to store an integer value and the attribute `power` will be used to store the result of the formula `pow(val, 2)`.

## Optimize Saving

During many code reviews performed for other iOS developers, I've seen over and over again Core Data implementations where the `NSManagedObjectContext save` method is called after each `NSManagedObject` manipulation, resulting in a significant performance decrease.

This happens most of the time during the process of loading or creating a bulk set of data. To illustrate the effect of this, implement the following code in the `YDViewController.m` file:

```
- (void)CreateDefaultData
{
    [[self appDelegate] resetStore];
    for (int i = 0 ;i<10000;i++)
    {
        Something* something =[NSEntityDescription
                               insertNewObjectForEntityForName:@"Something"
                               inManagedObjectContext:
                               [self appDelegate].managedObjectContext];
        something.val = [NSNumber numberWithInt:i];
        something.power = [NSNumber numberWithInt:(pow(i,2))];
        NSError *error = nil;
        if (![[self appDelegate].managedObjectContext save:&error]) {
            //handle your error
        }
    }
}
```

This code simply starts by resetting the data store, and using a loop, 10,000 `Something` instances are created, assigned property values, and saved. Call this method in the `viewDidLoad:` method of the `YDViewController`.

Using Instruments to profile the effect of running this method shows the following results:

Duration of the method	79 seconds
Overall memory usage	70 MB

If you change the method and remove the save action outside the loop so it won't be executed 10,000 times, the performance improvement is amazing, as you would expect.

Change the method as shown in the following code where the change is highlighted:

```
- (void)CreateDefaultData
{
    [[self appDelegate] resetStore];
    for (int i = 0 ;i<10000;i++)
    {
        Something* something =[NSEntityDescription
                               insertNewObjectForEntityForName:@"Something"
                               inManagedObjectContext:
                               [self appDelegate].managedObjectContext];
        something.val = [NSNumber numberWithInt:i];
        something.power = [NSNumber numberWithInt:(pow(i,2))];
```

```

        }
        NSError *error = nil;
        if (![[self appDelegate].managedObjectContext save:&error]) {
            //handle your error
        }
    }
}

```

Using Instruments again to profile the effect of running this method shows the following results:

Duration of the method	< 1 seconds
Overall memory usage	5.5 MB

As you can see, a simple change in your code has a significant effect on the performance of your application, and therefore also the users' experience of your application.

## Configuring the Managed Object Context

You can set an `NSUndoManager` on an `NSManagedObjectContext` to support undo and redo functionality in your application. It's obvious when you configure an `NSUndoManager` for the `NSManagedObjectContext` being used that the performance of your application will decrease, because the changes on the `NSManagedObjectContext` need to be kept in memory to enable the undo and redo support.

When you create an `NSUndoManager` instance, the `levelsOfUndo` property has a default value of 0. The `levelsOfUndo` property holds the number of groups (not operations), which should be kept by the manager. When this value is reached, the oldest changes are thrown away free memory. The default value of 0 causes a continuously increasing memory footprint during the use of your application, because the `NSUndoManager` will keep an unlimited number of groups.

By calling the `setLevelsOfUndo:` method on the `NSUndoManager` instance, you can set a value that makes more sense for the requirement of your application's functionality.

If your application doesn't support undo or redo functionality at all, it is good practice to set the `UndoManager` to `nil` during the configuration of the `NSManagedObjectContext`, as you can see in the following code:

```

- (NSManagedObjectContext *)appContext
{
    if (self.managedObjectContext != nil) {
        return self.managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self appStoreCoordinator];
    if (coordinator != nil) {
        _managedObjectContext = [[NSManagedObjectContext alloc] init];
        [_managedObjectContext setUndoManager:nil];
        [_managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return _managedObjectContext;
}

```

## CONCURRENCY WITH CORE DATA

In many real-world applications, there is a need to load data in the background and use Core Data for storing it. When a serious amount of data is involved, you don't want to process it on the main thread to avoid blocking of the user interface.

The developer documentation from Apple is very clear regarding the recommended pattern for concurrent programming with Core Data. It states, "The pattern recommended for concurrent programming with Core Data is thread confinement: each thread must have its own entirely private managed object context."

There are two possible ways to adopt the pattern:

- Create a separate managed object context for each thread and share a single persistent store coordinator. This is the typically recommended approach.
- Create a separate managed object context and persistent store coordinator for each thread. This option provides for greater concurrency at the expense of greater complexity.

You can find more information in the Apple documentation at <http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/CoreData/Articles/cdConcurrency.html>.

In the source code of this chapter, you can download a project named `ConcurrentThreads` to see a working example.

To adapt to the first approach you have to make changes to the application delegate (if that is the place where you initialize and configure Core Data).

The first step is to add the following two highlighted code lines to the `appContext` method. The first line is calling the `setMergePolicy:` method on the `managedObjectContext`, and the second line configures an Observer with a selector to the method `mocDidSaveNotification`:

```
- (NSManagedObjectContext *)appContext
{
    if (self.managedObjectContext != nil) {
        return self.managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self appStoreCoordinator];
    if (coordinator != nil) {
        _managedObjectContext = [[NSManagedObjectContext alloc] init];
        [self.managedObjectContext setUndoManager:nil];
        [self.managedObjectContext
            setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
        // subscribe to change notifications
        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(mocDidSaveNotification:)
            name:NSManagedObjectContextDidSaveNotification
            object:nil];
        [self.managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return self.managedObjectContext;
}
```

The implementation of the `mocDidSaveNotification:notification` method is performing the following logic:

- If the change notification is from the main managed object context, ignore it because it will be saved by the `saveContext` method.
- If the `persistentStoreCoordinator` of the managed object context is different than the `persistentStoreCoordinator` of the managed object context belonging to the save notification, again no action is performed.
- In all other cases, the managed object context `mergeChangesFromContextDidSaveNotification:notification` method is called in the main dispatch queue, saving and merging the changes:

```
- (void)mocDidSaveNotification:(NSNotification *)notification
{
    NSManagedObjectContext *savedContext = [notification object];

    // ignore change notifications for the main MOC
    if (self.managedObjectContext == savedContext)
    {
        return;
    }

    if (self.managedObjectContext.persistentStoreCoordinator != savedContext.persistentStoreCoordinator)
    {
        // that's another database
        return;
    }

    dispatch_sync(dispatch_get_main_queue(), ^{
        [self.managedObjectContext
         mergeChangesFromContextDidSaveNotification:notification];
    });
}
```

## SUMMARY

In this chapter you learned how to use Core Data for data storage in your application and how to implement it for the best performance.

In Part III of this book you learn how to integrate your iOS application with the outside world and other applications on the device.

# PART III

## Integrating Your App

---

- ▶ **CHAPTER 10:** Notifications
- ▶ **CHAPTER 11:** Sending E-Mail, SMS and Dialing a Phone
- ▶ **CHAPTER 12:** Understanding the Address Book
- ▶ **CHAPTER 13:** Event Programming
- ▶ **CHAPTER 14:** Integrating with Social Media



# 10

## Notifications

---

### WHAT'S IN THIS CHAPTER?

---

- Learning how to implement internal and external notifications in your application
- Learning how to implement Apple Push notifications using Urban Airship
- Registering custom URL schemes

### WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 10 download and individually named according to the names throughout the chapter.

Local notifications and push notifications are technologies you can implement in your iOS application that inform your application (when not running in the foreground) that an event happened or new information is available.

Like in a mail application, when your application is running in the background and you receive a new mail, a local notification is created to let you know a new mail has been delivered to your inbox.

## IMPLEMENTING LOCAL NOTIFICATIONS

Local notifications are notifications that are generated and received by your application. The purpose of the local notification is to inform your application's user that something interesting for him happened while the application was not running in the foreground.

### Understanding Local Notifications

The main characteristics of a local notification are:

- They are launched and delivered by iOS on the same device.
- They can be scheduled by date and time.
- They can be created instantly based on an event.
- They can include an `NSDictionary` with custom data.
- They can contain a sound to play, an application badge number, and the title of the action button.

To learn and understand how to work with local notifications, you will create a new application that will act like an alarm clock.

Start Xcode and create a new project using the Single View Application Project template, and name it `AlarmClock` using the options shown in Figure 10-1.

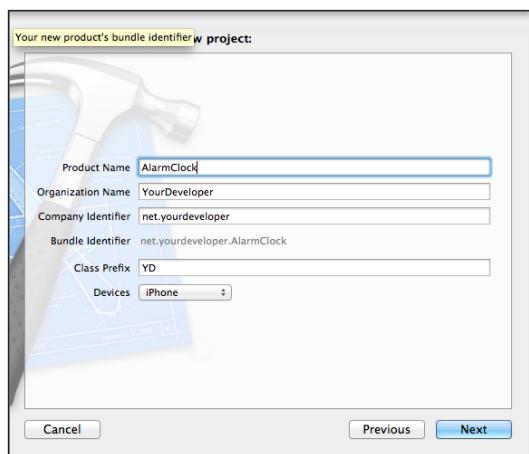


FIGURE 10-1

Use Interface Builder and the Assistant Editor to create a user interface for the `YDViewController.xib` as shown in Figure 10-2.

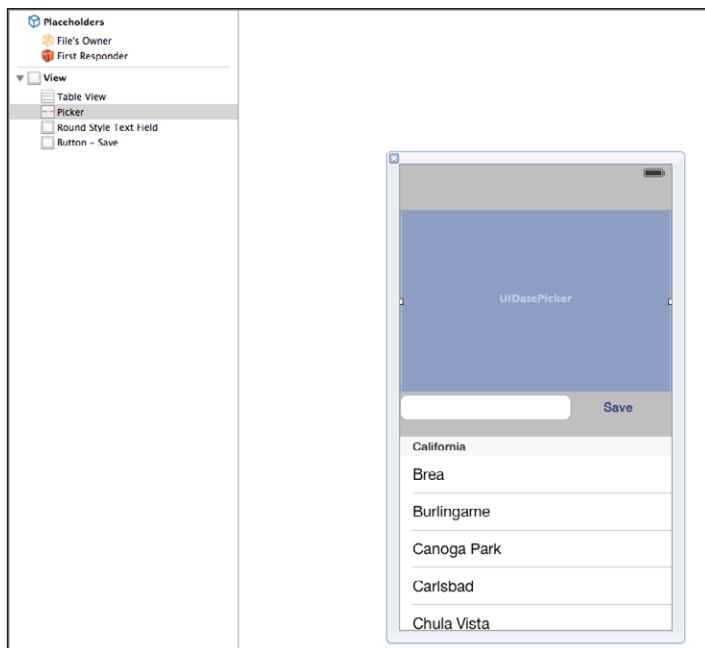


FIGURE 10-2

## Creating a Notification

Verify your YDViewController.h file is equal to the one shown in Listing 10-1.

### LISTING 10-1: Chapter10/AlarmClock/YDViewController.h

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

@interface YDViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIDatePicker *dPicker;
@property (weak, nonatomic) IBOutlet UITextField *EventText;

@property (weak, nonatomic) IBOutlet UITableView *mTableView;
- (IBAction)saveEvent:(UIButton *)sender;

@end
```

The implementation is shown in Listing 10-2. The `saveEvent:` method is where the `UILocalNotification` object is created and initialized. The `DatePicker` value is converted to a date and time on which the event should be launched, and the `alertBody` is set.

Finally, you create a custom NSDictionary with values and keys you would like to pass to the userinfo dictionary.

**LISTING 10-2:** Chapter10/AlarmClock/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController : UIViewController

@implementation YDViewController

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

#pragma mark Table view data source

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

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

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:CellIdentifier];
    }
    NSArray *notificationArray = [[UIApplication sharedApplication]
        scheduledLocalNotifications];
    UILocalNotification *notification = [notificationArray
        objectAtIndex:indexPath.row];

    [cell.textLabel setText:notification.alertBody];
    [cell.detailTextLabel setText:[notification.fireDate description]];

    return cell;
}
```

```

- (IBAction)saveEvent:(UIButton *)sender {
    [self.EventText resignFirstResponder];

    NSCalendar *calendar = [NSCalendar autoupdatingCurrentCalendar];

    // Get the current date
    NSDate *pickerDate = [self.dPicker date];

    // Break the date up into components
    NSDateComponents *dateComponents =
        [calendar components:
            ( NSYearCalendarUnit |
              NSMonthCalendarUnit |
              NSDayCalendarUnit )
            fromDate:pickerDate];
    NSDateComponents *timeComponents =
        [calendar components:
            ( NSHourCalendarUnit |
              NSMinuteCalendarUnit |
              NSSecondCalendarUnit )
            fromDate:pickerDate];

    // Set up the fire time
    NSDateComponents *dateComps = [[NSDateComponents alloc] init];
    [dateComps setDay:[dateComponents day]];
    [dateComps setMonth:[dateComponents month]];
    [dateComps setYear:[dateComponents year]];
    [dateComps setHour:[timeComponents hour]];
    // Notification will fire in one minute
    [dateComps setMinute:[timeComponents minute]];
    [dateComps setSecond:[timeComponents second]];
    NSDate *itemDate = [calendar dateFromComponents:dateComps];

    UILocalNotification *localNotif = [[UILocalNotification alloc] init];
    if (localNotif == nil)
        return;
    localNotif.fireDate = itemDate;
    localNotif.timeZone = [NSTimeZone defaultTimeZone];

    localNotif.alertBody = [self.EventText text];

    localNotif.alertAction = @"View";

    localNotif.soundName = UILocalNotificationDefaultSoundName;
    localNotif.applicationIconBadgeNumber = 1;

    // Specify custom data for the notification
    NSMutableDictionary *infoDict = [[NSMutableDictionary alloc] init];
    [infoDict setObject:[self.EventText text] forKey:@"msg"];
    [infoDict setObject:@"alarmclock" forKey:@"sender"];

    // NSDictionary *infoDict = [NSDictionary
    //     dictionaryWithObject:@"someValue"

```

*continues*

**LISTING 10-2** (continued)

```

        forKey:@"someKey"];
localNotif.userInfo = infoDict;

        // Schedule the notification
[[UIApplication sharedApplication] scheduleLocalNotification:localNotif];

[self.tableView reloadData];
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

## Receiving a Notification

Receiving a notification is usually handled within the application's delegate implementation.

Simply implement the following method to create a `UIAlertView` displaying the message you've passed to the custom object.

```

- (void)application:(UIApplication *)app didReceiveLocalNotification:
    (UILocalNotification *)notif {
    // Handle the notificaton when the app is running

    UIAlertView* alert = [[UIAlertView alloc]
        initWithTitle:@"Alert" message:[[notif userInfo]
            objectForKey:@"msg"] delegate:self cancelButtonTitle:@"Ok"
            otherButtonTitles:nil, nil];
    [alert show];
}

```

When the application is launched from the notification, you have to process the notification in the `application:didFinishLaunchingWithOptions:` method by implementing the code as shown in the next example:

```

application.applicationIconBadgeNumber = 0;

// Handle launching from a notification
UILocalNotification *localNotif =
[launchOptions objectForKey:UIApplicationLaunchOptionsLocalNotificationKey];
if (localNotif) {

    UIAlertView* alert = [[UIAlertView alloc]
        initWithTitle:@"Alert"

        message:[[localNotif userInfo] valueForKey:@"msg"]]
}

```

```

delegate:self
cancelButtonTitle:@"Ok"
otherButtonTitles:nil, nil];
[alert show];
}

```

The complete project is available from the download site.

## UNDERSTANDING PUSH NOTIFICATIONS

Apple Push Notification Service, also known as APNS, is the centerpiece in the architecture of sending external notifications to iOS devices.

A push notification consists of two elements. One is the unique identifier of the target device, also known as the device token, and the other is the payload, a JSON-defined property list that will allow the receiving application to parse and present as desired. The flow of a push notification is presented in Figure 10-3.

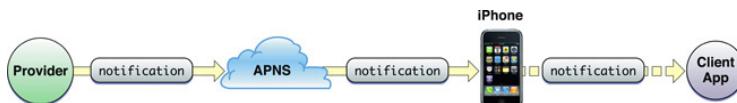


FIGURE 10-3:

You can find the complete architectural components and security flow at <http://developer.apple.com/library/mac/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Introduction.html>.

**NOTE** *The device token of the device is not the same as the UDID returned by the uniqueidentifier property of the UIDevice. As you can see in Figure 10-3, the provider that initiates the sending of the push notification is presented as a process. You can develop your own custom push notification provider in your preferred programming language, or you can use a publicly available service like Urban Airship (<http://urbanairship.com/>) that provides you with a ready-to-use infrastructure and technology that is easy to implement using its iOS SDK.*

During each project where push notifications are required, I decide which provider to use and how to use them. For several projects I used my custom push notification provider developed in C# to have a maximal integration with other back-end processes. On the other hand, for “simple” push notification support, I use Urban Airship as well, and because most developers do, next you learn how to implement Urban Airship push notifications in your iOS application.

Start Xcode and create a new project using the Single View Application Project template, and name it PushDemo using the options shown in Figure 10-4.

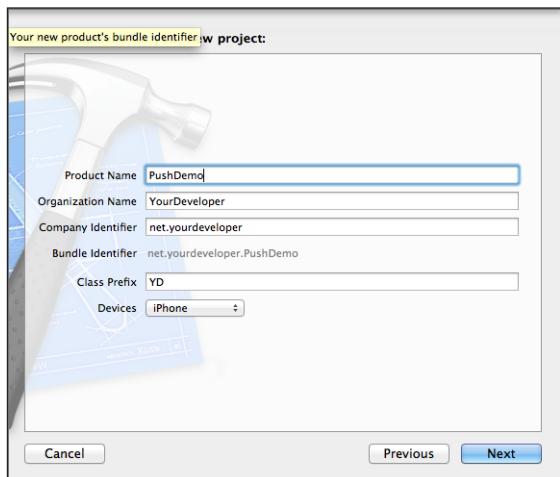


FIGURE 10-4

**NOTE** At the time of writing this chapter, the Urban Airship iOS SDK doesn't support Automatic Reference Counting (ARC). To use the Urban Airship libraries in your ARC project, don't forget to use the `-fno-objc-arc` compiler flag for all classes starting with UA (Urban Airship classes). When you compile your project you will see a lot of warnings, because the UrbanAirship library is using a lot of deprecated methods.

The full Urban Airship SDK requires you to add several frameworks and libraries to your project. Open the project target, select the Build Phases tab, and under Link Binary with Libraries add the following frameworks/libraries:

- MobileCoreServices.framework
- MapKit.framework
- MessageUI.framework
- Security.framework
- SystemConfiguration.framework
- CoreTelephony.framework
- CoreLocation.framework
- StoreKit.framework
- AudioToolbox.framework

- CFNetwork.framework
- Libsqlite3.dylib
- Libz.dylib

## Configuring the Developer Portal

Log on to <http://developer.apple.com> and sign in using your credentials in the member center.

Once you are logged in, select the Certificates, Identifiers & Profiles option from the page as shown in Figure 10-5.

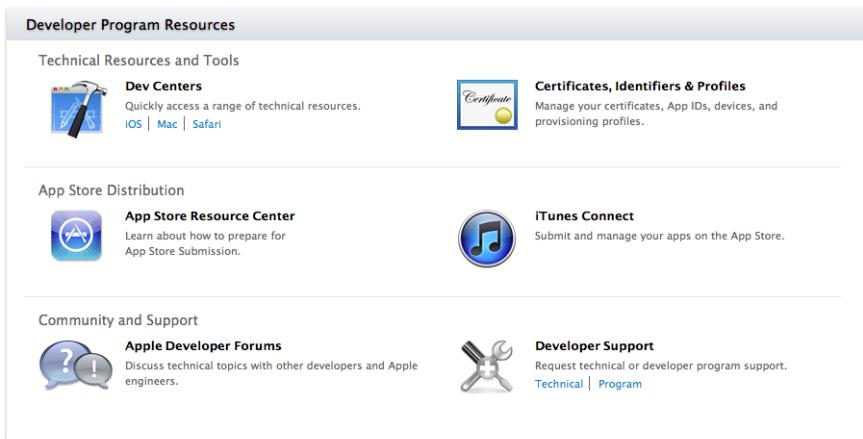


FIGURE 10-5

In the next screen select Identifiers within the iOS Apps section as shown in Figure 10-6.



FIGURE 10-6

Register your new application like you normally would. Make sure you select the Push Notification checkbox and use an Explicit App ID with a unique bundle ID (in my case I always use the reverse domain name style suffixed by the application name), such as `net.yourdeveloper.pushdemo` as shown in Figure 10-7.

The screenshot shows the 'App Services' configuration screen. At the top, there's a 'Name' field containing 'PushDemo' with a note below it: 'You cannot use special characters such as @, &, ^, "'. Below this, the 'App Services' section allows enabling various services. Under 'Enable Services', 'Push Notifications' is checked. Other options like Data Protection, Game Center, iCloud, In-App Purchase, and Passbook are also listed. The 'App ID Prefix' field contains '7M34WB43QV (Team ID)'. The 'App ID Suffix' section shows 'Explicit App ID' selected, with a note about creating a unique string matching the Bundle ID. The 'Bundle ID' field contains 'net.yourdeveloper.pushdemo'.

Name: PushDemo  
You cannot use special characters such as @, &, ^, "  
  
**App Services**  
Select the services you would like to enable in your app. You can edit your choices after this App ID has been registered.  
  
Enable Services:  
 Data Protection  
 Complete Protection  
 Protected Unless Open  
 Protected Until First User Authentication  
 Game Center  
 iCloud  
 In-App Purchase  
 Passbook  
 Push Notifications  
  
**App ID Prefix**  
Value: 7M34WB43QV (Team ID)  
  
**App ID Suffix**  
 **Explicit App ID**  
If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.  
To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.  
Bundle ID: net.yourdeveloper.pushdemo

FIGURE 10-7

Follow the instructions on the screen and confirm your application submission. The confirmation screen will summarize the options you've selected and display the identifier for your application as shown in Figure 10-8.



FIGURE 10-8

A final confirmation screen is shown and your application is registered.

**NOTE** *The Apple developer portal is changed quite frequently, where the main changes are in the area of the user interface. For that reason it's possible that at the moment you are reading this chapter, the user interface is slightly different than presented in the figures. However, the functionality and required steps you have to take will not change, giving you a clear understanding on how to commence in case the user interface looks a little different.*

Your application is now listed, and if you select the application line from the grid it will display as shown in Figure 10-9.

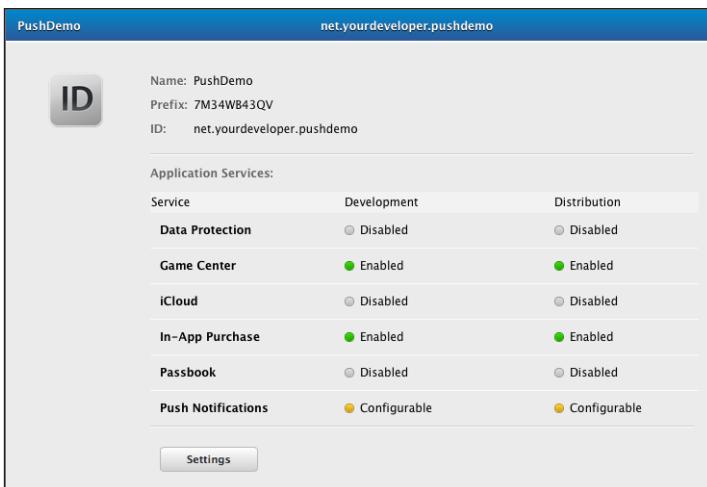


FIGURE 10-9

## Obtaining Certificates

Because the communication between the Apple Push Notification Server and the iOS device is a secure connection, you need to obtain certificates, not only for signing your application but also to make sure the Urban Airship configuration, or your customer Push Notification Service Provider, is able to sign the request when sending the payload to the server.

If you click the Settings button on the screen shown in Figure 10-9, you will see the screen in which you can configure the settings for this application. You will see that Push Notifications are enabled but the screen also will display that the Development SSL Certificate and the Production SSL Certificate are not yet created, as shown in Figure 10-10.

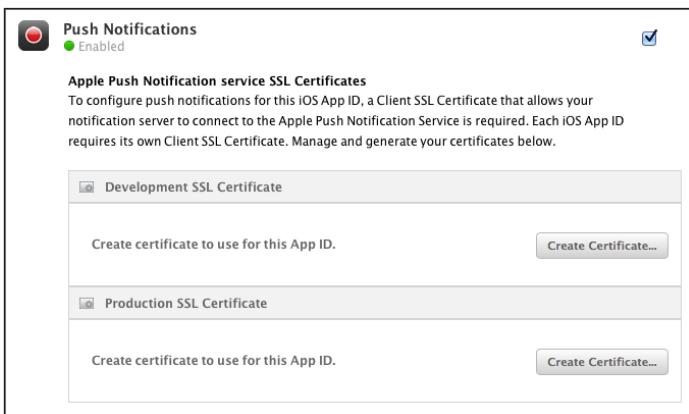


FIGURE 10-10

You need to create both the development and the production certificate, which you can do by clicking the Request button, and following the instructions on the screen.

After uploading your CSR file, the system will generate the certificate and indicate when the certificate is ready, as shown in Figure 10-11.

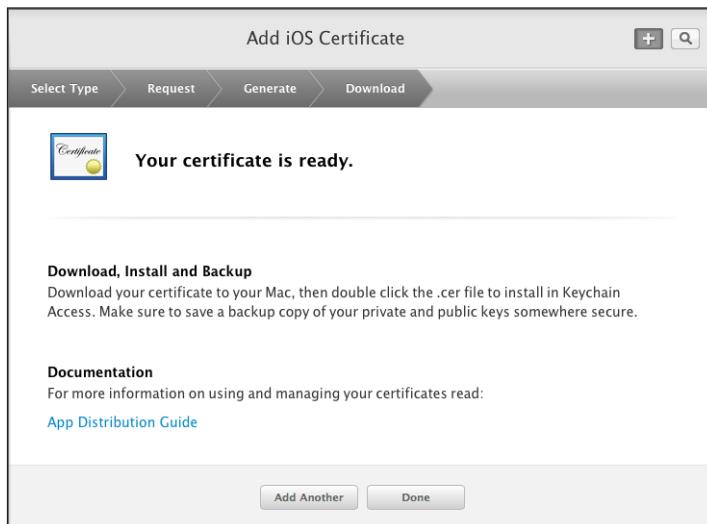


FIGURE 10-11

After you've repeated this action for the production certificate, your screen will look like Figure 10-12, allowing you to download both certificates.

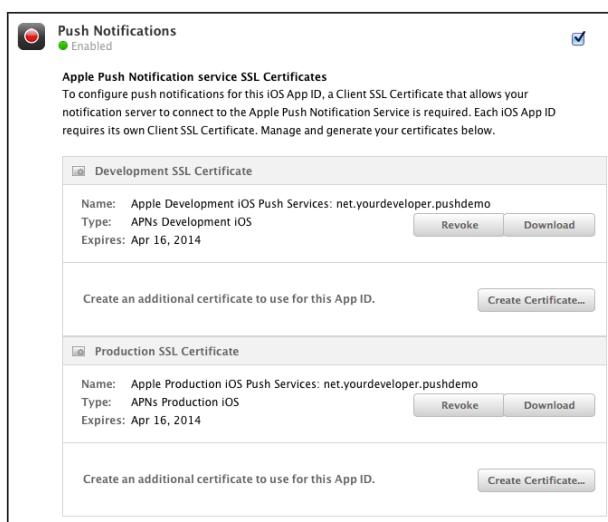


FIGURE 10-12

After downloading both the development and the production certificates, you can use Finder to locate them and move them to the folder where you keep track of all certificates.

**NOTE** Because the certificates are always named the same (aps\_development.cer and aps\_production.cer), it's convenient to create a folder containing all your certificates by storing them in a subfolder with the application name. In my case I have the folder Certificates\ClientName\ApplicationName to keep track of the certificates for different clients and applications.

So now you have an aps\_development.cer and an aps\_production.cer certificate. Use Finder and click both certificates to have them automatically installed in the keychain access. Open Keychain Access and select My Certificates under the Category tabs and you'll see a certificate in right-hand pane with the name Apple Development IOS Push Services: xxxxxxx, where xxxxxxx is the name of your application.

Select the certificate you need (development or production) in the right pane, and right-click and choose Export or choose File ↗ Export from the menu. In the pop-up screen you can enter the name and choose the location. Choose Personal Information Exchange (.p12) as the file format as shown in Figure 10-13.

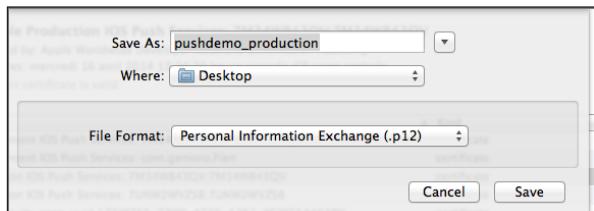


FIGURE 10-13

To distinguish the certificates, name yours pushdemo\_production and click Save. A new window as shown in Figure 10-14 pops up, and prompts you for a password and verification for this certificate. Enter a strong password, verify it, and click OK.

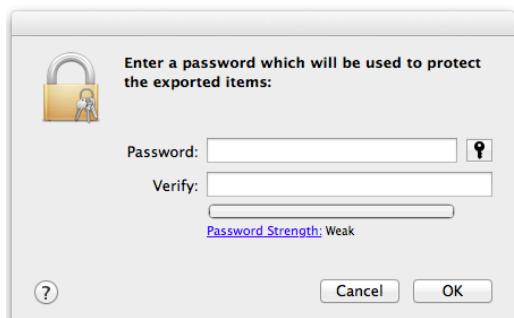


FIGURE 10-14

Make sure the .p12 certificates are also copied in the folder where you keep the certificates so they are all in one place.

## Implementation with Urban Airship

To implement push notifications using Urban Airship as the provider, follow the steps in this section carefully.

**NOTE** At this point it's important to emphasize that I'm not in any way affiliated with Urban Airship, its services, or subsidiaries. The main reason I've chosen to use Urban Airship as the push notification provider in this chapter is simply because I know from experience that most iOS developers are not backend developers and don't have the knowledge and skills available to develop a custom push notification provider. For the purpose of this chapter—understanding how push notifications work and how to implement them—Urban Airship is just a simple free-to-setup vehicle to give a jump start.

### Creating a Free Account

Visit the website [www.urbanairship.com](http://www.urbanairship.com), and if this is your first time use the Choose Plan button in the menu to sign up for a free developer account. You can choose a different account plan if you prefer.

Follow the instructions on the website to register and gain access to the developer portal. Use the credentials you received/created to log in to the developer portal.

Go to Apps (Your Applications) and select + New App to add a new application. Follow the instructions on the screen, name your application, upload an icon, and specify whether this configuration is for development or production.

During the configuration of the services, select Apple Push Notification Service and click the Configure button. The system will ask you to upload a certificate (this is the exported .p12 certificate you created earlier) and enter the certificate password you used during the export of this certificate. Enter the correct password, upload the certificate, and click Save. Your configuration in Urban Airship is now done.

Under Settings you can find an option named API Keys, and when you click it you'll find an App Key, an App Secret, and an App Master Secret. You need to copy the first two because you need them in the configuration.

To send test messages after you have finalized the implementation of your application, you can use the submenus available under the application you created earlier. It's outside the scope of this chapter to explain in detail which capabilities are available with Urban Airship.

## Downloading the SDK

The exact location of the download link to the SDK has changed various times in the past months, but a steady factor is a higher-level page you can find at <http://urbanairship.com/resources/developer-resources>.

Find the latest stable iOS SDK library and download it from there following the latest instructions on the website.

## Implementing the SDK

To implement the SDK simply drag and drop the complete folder structure you have downloaded into your project structure, as shown in Figure 10-15.

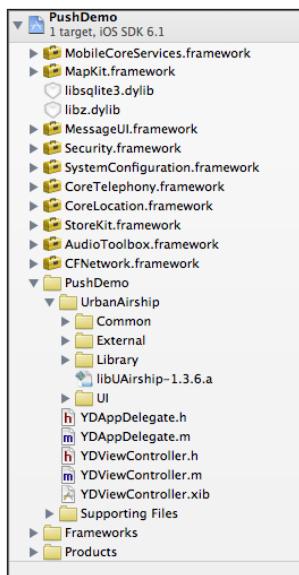


FIGURE 10-15

In the Supporting Files group add a new plist file named `AirshipConfig.plist` and create the settings as shown in Figure 10-16. Use the App Key and App Secret you found in the Urban Airship portal.

Key	Type	Value
Root	Dictionary	(5 items)
APP_STORE_OR_AD_HOC_BUILD	String	YES
DEVELOPMENT_APP_KEY	String	
DEVELOPMENT_APP_SECRET	String	
PRODUCTION_APP_KEY	String	
PRODUCTION_APP_SECRET	String	

FIGURE 10-16

## Enhancing Your AppDelegate

Listing 10-3 shows the implementation of the push notification logic using the Urban Airship provider library. After the import, the `application:didFinishLaunchingWithOptions:` method is extended with the `UAPush` initialization code.

The `application:didRegisterForRemoteNotificationsWithDeviceToken:` method is called by the `registerForRemoteNotificationTypes:` response method and the device token will be sent to Urban Airship.

The `application:didReceiveRemoteNotification:handleNotification:applicationState:` method is called if the APNS server is sending a payload to this device, by sending it to the device token. You can leave handling the notification in the hands of the Urban Airship SDK, or you can write your own logic to process the information that is sent as payload in the `UserInfo` object.

**LISTING 10-3: Chapter10/PushDemo/YAppDelegate.h**

```
#import "YAppDelegate.h"

#import "YDViewController.h"
#import "UAirship.h"
#import "UAPush.h"
#import "UAUtils.h"
@implementation YAppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
    (NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    // the app is launched from a push notification
    NSMutableDictionary *takeOffOptions = [[NSMutableDictionary alloc] init];
    [takeOffOptions setValue:launchOptions forKey:
        UAirshipTakeOffOptionsLaunchOptionsKey];
    [UAPush setDefaultPushEnabledValue:YES];
    [UAirship takeOff:takeOffOptions];

    [[UAPush shared] resetBadge];//zero badge on startup
    [[UAPush shared]
    registerForRemoteNotificationTypes:
        (UIRemoteNotificationTypeBadge |
         UIRemoteNotificationTypeSound |
         UIRemoteNotificationTypeAlert)];
    // Override point for customization after application launch.
    self.viewController = [[YDViewController alloc]
        initWithNibName:
            @"YDViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

*continues*

**LISTING 10-3 (continued)**

```

// Implement the iOS device token registration callback
- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken {
    [[UAPush shared] registerDeviceToken:deviceToken];
}
- (void)application:(UIApplication *)application didReceiveRemoteNotification:
(NSDictionary *)userInfo {
    [[UAPush shared] handleNotification:userInfo
        applicationContext:application.applicationState];
    // Reset the badge if you are using that functionality
    [[UAPush shared] resetBadge]; // zero badge after push received
}

```

You now can log in to Urban Airship and send a test push notification message once your device token has been registered. It's important to realize the push notifications don't work on the iOS Simulator and won't work in real devices if the project is not signed with an ad-hoc or production profile.

## EXTERNAL NOTIFICATIONS

If you want your application to be accessible or launchable by other iOS applications on the user's device, you can use URL schemes.

For example, in an application I've developed for a client the user can record a movie and send an e-mail to a friend with a link to share this movie. A custom URL scheme is used in the e-mail, so that when the URL is opened on an iPad or iPhone where the application is installed, the recorded movie is opened and played in the application and the back-end management system is updated with relevant information from the receiving user.

Start Xcode and create a new project using the Single View Application Project template, and name it `UrlSchemeDemo`.

In this small project you will implement a custom URL scheme and the logic to make use of it in your own applications.

## Defining a Custom URL Scheme

To define a custom URL scheme, navigate to the `URLSchemeDemo-Info.plist` file and open it. To create a custom URL scheme, you start by adding a row (right-click and choose Add Row) and selecting URL Types from the drop-down list. Navigate to the created node, and under Item 0 add a new row of type URL Schemes, which is an array of items. As Item 0, create the identifier you would like to use like in the example `urldemo`.

After taking these steps, the property list will look like Figure 10-17.

Key	Type	Value
▼Information Property List	Dictionary	(15 items)
Application fonts resource path	String	
Localization native development reg	String	en
Bundle display name	String	\$(PRODUCT_NAME)
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	net.yourdeveloper.\$(PRODUCT_NAME)identifier
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environment	Boolean	YES
► Required device capabilities	Array	(1 item)
▼ URL types	Array	(1 item)
▼ Item 0	Dictionary	(1 item)
▼ URL Schemes	Array	(1 item)
Item 0	String	urldemo
► Supported interface orientations	Array	(3 items)

FIGURE 10-17

What you now have achieved is a custom URL scheme with an identifier named `urldemo`. This means that you can trigger your application by calling `urldemo://xxxxxx`, where `xxxxxx` is free and `urldemo:` is the identifier you have created.

## Responding to the URL Request

The `UIApplication` has a delegate method named `application: openURL:sourceApplication:annotation:` that is allowing you to write your custom processing code to respond to the URL request.

In the following code you see a sample implementation of this method in the `YDAppDelegate.m` file:

```
- (BOOL)application:(UIApplication *)application
    openURL:(NSURL *)url
  sourceApplication:(NSString *)sourceApplication
 annotation:(id)annotation {
    //You can perform an additional check to make sure you're not picking up an
    //url identifier from another application
    if ([[url host] isEqualToString:@"net.yourdeveloper.somevalue"])
    {
        //parse the query string if used to get access
        //to the query string variables and values
        NSArray *querySplitter = [[url query] componentsSeparatedByString:@"&"];
        if ([querySplitter count] ==1)
        {

        }
    }
    return YES;
}
```

Set a breakpoint in the preceding method and run your application. Press the Home button and launch Safari on your device or simulator, and your application will keep running in the background. Open Safari, type `urldemo://net.yourdeveloper/whatever` in the URL bar, and press Enter, and your application will enter the foreground mode and will stop execution at the set breakpoint. Within this method you can verify the URL that has been passed, access the query string to parse the values and keys that have been passed, and based upon the receiving values, implement your process.

By using custom URL schemes you can provide functionality to other applications to launch your applications, and even pass information using query strings in the URL.

## SUMMARY

In this chapter you learned how to make use of internal notifications and how to implement push notifications. You also learned how to make use of custom URL schemes to make your application accessible and launchable from other applications.

In the next chapter you learn how to integrate e-mail, SMS, and phone calls in your application.

# 11

## Sending E-Mail, SMS, and Dialing a Phone

### **WHAT'S IN THIS CHAPTER?**

---

- Learning how to send e-mails from your application
- Sending text messages from your application
- Dial a phone number from your application

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 11 download and individually named according to the names throughout the chapter.

### **SENDING E-MAIL**

In this chapter you create a simple application that enables you to present the user interface element for writing an e-mail message and that contains the logic for sending the message to a recipient.

Start Xcode and create a new project using the Single View Application Project template, and name it `emailler` using the options shown in Figure 11-1.

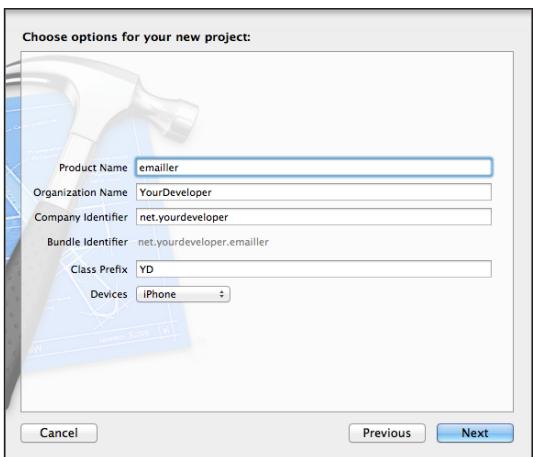


FIGURE 11-1

To send e-mail messages you need to start by adding the `MessageUI` framework, which contains the user interface elements and base classes for e-mail messaging.

## Composing an E-Mail

You compose an e-mail using an `MFMailComposeViewController` that you create, initialize, configure, and present with code, as shown in Listing 11-1.

### LISTING 11-1: `sendEmail:` method

```
- (IBAction)sendEmail:(UIButton *)sender {
    [self.subjectField resignFirstResponder];
    if ([MFMailComposeViewController canSendMail])
    {
        MFMailComposeViewController *mailer =
        [[MFMailComposeViewController alloc] init];
        mailer.mailComposeDelegate = self;
        [mailer setSubject:self.subjectField.text];
        NSArray *toRecipients =
        [NSArray arrayWithObjects:
         @"email@domain.com", nil];
        [mailer setToRecipients:toRecipients];
        [mailer setMessageBody:@"" isHTML:YES];
        [self presentViewController:mailer animated:YES completion:nil];
    }
    else
    {
        UIAlertView *alert =
        [[UIAlertView alloc]
        initWithTitle:@"Sorry"
        message:@"Your device doesn't support sending email."
        delegate:nil
```

```
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
[alert show];
}
}
```

The `setToRecipients:`, `setCcRecipients:`, and `setBccRecipients:` methods all accept an `NSArray` of e-mail addresses.

You can use the `setMessageBody:isHTML:` method to set predefined text in the message body before the ViewController is presented. The `isHTML` boolean defines whether the message body itself is HTML or plain text.

You initiate the actual sending of the e-mail by pressing the Send button of the `MFMailComposeViewController`. There is also a Cancel button that can be tapped by the user. In any case it's important for you to implement the `mailComposeController:didFinishWithResult:error:` delegate method so you can take the appropriate action depending on the result of the send or cancel action in the `MFMailComposeViewController`, as shown in Listing 11-2.

#### LISTING 11-2: `mailComposeController:` method

```
- (void)mailComposeController:(MFMailComposeViewController*)controller
didFinishWithResult:(MFMailComposeResult)result
error:(NSError*)error
{
    switch (result)
    {
        case MFMailComposeResultCancelled:
            // you cancelled the operation and no email message was queued.
            break;
        case MFMailComposeResultSaved:
            //Mail saved: you saved the email message in the drafts folder.
            break;
        case MFMailComposeResultSent:
        {
            // the email message is queued in the outbox. It is ready to send. ;

        }
        break;
        case MFMailComposeResultFailed:
            // the email message was not saved or queued, possibly due to an error.
            break;
        default:

            break;
    }
    // Remove the mail view
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

**NOTE** The `MFMailComposerViewController` requires an SMTP account to have been set up on the phone. If no SMTP account has been set up, the e-mail can't be sent.

## Working with Attachments

The `MFMailComposeViewController` has the delegate method `addAttachmentData:mimeType:fileName:` that you can use to add attachment data to the message you are composing.

To attach an image that is located in the bundle of your application, you can add the following code in the `sendEmail:` method:

```
//sample add some image
UIImage* imgToAttach = [UIImage imageNamed:@"banner-ios.png"];
NSData *data = UIImageJPEGRepresentation(imgToAttach, 1.0);
[mailer addAttachmentData:data mimeType:@"image/jpg"
fileName:@"banner-ios.png"];
```

If you launch the application, you can enter a subject, tap the Send Email button, and the ViewController is presented. This is the configured `MFMailComposeViewController`, and it will display like the one shown in Figure 11-2.

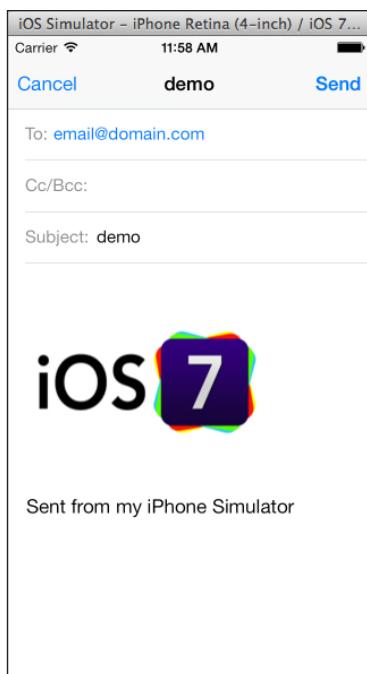


FIGURE 11-2

## SENDING SMS (TEXT MESSAGE)

Sending text messages, also known as SMS on an iOS device, is an action that can't be performed without user interaction and user authorization. Whereas on an Android device you can send a text message completely from code without the user of the device even knowing it has been sent, iOS has a more profound security layer in place.

### Verifying if SMS Is Available

The `MFMessageComposeViewController` is used to send an SMS, and therefore you need to start with importing the `<MessageUI/MFMessageComposeViewController.h>` header definition after you've added the `MessageUI` framework to your project.

The `MFMessageComposeViewController` has a static method named `canSendText` that will return a `BOOL` if a text message can be sent. If the return value is `NO`, the device is not capable of sending an SMS, so you can inform the user that this function is not available.

### Composing a Text Message

After creating a user interface with a `UITextField` or a `UITextView` in which you capture the user's message that needs to be sent, you perform the following operations:

- Create and initialize an instance of the `MFMessageComposeViewController`
- Set the `body` property of the your controller's instance
- Set an `NSArray` of recipients, where a recipient is a mobile number
- Set the `messageComposeDelegate` property
- Present the created ViewController modally

Implement the `sendSMSToNumber:` method as shown in Listing 11-3.

#### LISTING 11-3: `sendSMSToNumber:` method

```
- (void)sendSMSToNumber:(NSString *)mobileNumber
{
    MFMessageComposeViewController *controller =
    [[MFMessageComposeViewController alloc] init];
    if ([MFMessageComposeViewController canSendText])
    {
        controller.body = NSLocalizedString(@"SMSMESSAGE", nil);
        controller.recipients = [NSArray arrayWithObjects:mobileNumber, nil];
        controller.messageComposeDelegate = self;
        [self presentModalViewController:controller animated:YES];
    }
}
```

Because the user has two options in the ViewController dialog box—cancel or send the message—you need to implement the delegate method `messageComposeViewController:didFinishWithResult:`.

After dismissing the modal ViewController, you perform the code required for canceling or sending the message, depending on your application’s requirement.

Implement the `messageComposeViewController:didFinishWithResult:` as shown in Listing 11-4.

#### LISTING 11-4: `messageComposeViewController:` method

```
- (void)messageComposeViewController:(MFMessageComposeViewController *)controller
didFinishWithResult:(MessageComposeResult)result
{
    [self dismissModalViewControllerAnimated:YES];

    if (result == MessageComposeResultCancelled)
    {
        //cancelled
    }
    else if (result == MessageComposeResultSent)
    {
        //sent
    }
    else
    {
        //failed
    }
}
```

## DIALING A PHONE NUMBER

Your application might have an About or Contact view, in which your or your client’s contact information is displayed, including a phone number.

When an iPhone user views a phone number in an application, he expects that he can just click on it and the number will be dialed. That’s the whole concept of a smartphone.

## Verifying Dialing Capability

Not every iOS device is capable of dialing a phone number, so before implementing the dial code it’s necessary to verify the capabilities of the device to avoid an application crash.

Dialing a phone number consists of simply opening an URL on the device that starts with the URL identifier `tel://` or `telprompt://`, which are built-in URL schemes that work exactly like the custom URL schemes you used in the previous chapter.

---

Use the following snippet of code to verify whether the device has the capability of dialing a phone number:

```
If ([[UIApplication sharedApplication] canOpenURL:  
    [NSURL URLWithString:@"tel://"]]) {  
    // device can dial  
}  
else {  
    //device can't dial  
}
```

You probably figured this one out yourself, didn't you?

Indeed, you use the `openURL:` method of the `UIApplication` class and pass in the URL identifier with the phone number like this:

```
NSString* phoneNumber = @"+15558888";  
[[UIApplication sharedApplication] openURL:  
[NSURL URLWithString:[NSString stringWithFormat:@"tel://%@", phoneNumber]]]];
```

If your application predefines the phone number to be dialed after performing an action like clicking a button, make sure you use the international notation to guarantee the number can be dialed from any user's device.

A special URL scheme is available that will dial the phone number but will return to your application once the call is terminated. For this purpose use the following code to start the dialing. It will also present a `UIAlertView` before the dialing starts.

```
NSString* phoneNumber = @"+15558888";  
[[UIApplication sharedApplication] openURL:  
[NSURL URLWithString:[NSString stringWithFormat:@"telprompt://%@", phoneNumber]]]];
```

## SUMMARY

In this chapter you learned how to send e-mail messages, text messages, and dial a phone number.

In the next chapter you learn how to access and work with the Address Book on your device.



# 12

## Understanding the Address Book

### **WHAT'S IN THIS CHAPTER?**

---

- Introducing the Address Book
- Accessing the Address Book programmatically

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 12 download and individually named according to the names throughout the chapter.

### **INTRODUCTION TO THE ADDRESS BOOK FRAMEWORK**

The Address Book technology provides a way to store contacts and other personal information in a centralized database in an iOS device. You can share this information between applications by using the Address Book framework.

You can find the complete Address Book framework reference at [http://developer.apple.com/library/ios/#documentation/AddressBook/Reference/AddressBook\\_iPhoneOS\\_Framework/\\_index.html](http://developer.apple.com/library/ios/#documentation/AddressBook/Reference/AddressBook_iPhoneOS_Framework/_index.html).

You can find the Address Book programming guide for iOS at [http://developer.apple.com/library/ios/#documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Introduction.html%23//apple\\_ref/doc/uid/TP40007744](http://developer.apple.com/library/ios/#documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Introduction.html%23//apple_ref/doc/uid/TP40007744).

The Address Book technology consists of the following parts:

- The Address Book framework itself, which provides access to the centralized database with contact information.
- The Address Book UI framework, which provides user interface elements to present the information.
- The Address Book database, for storing the contact information on the iOS device.
- The Contacts application on the iOS device, which is a complete iOS application delivered by Apple to access the contact information.

In this chapter a contact from the Address Book is also referred to as a *person*.

## ACCESSING THE ADDRESS BOOK

In this section you create an application that shows the main functionalities required to access the Address Book. You learn how to select a contact, display the contact details, add, edit, and delete a contact, and create a new contact from existing data.

### Selecting a Contact

Start Xcode and create a new project using the Single View Application Project template, and name it MyAddressBook using the options shown in Figure 12-1.

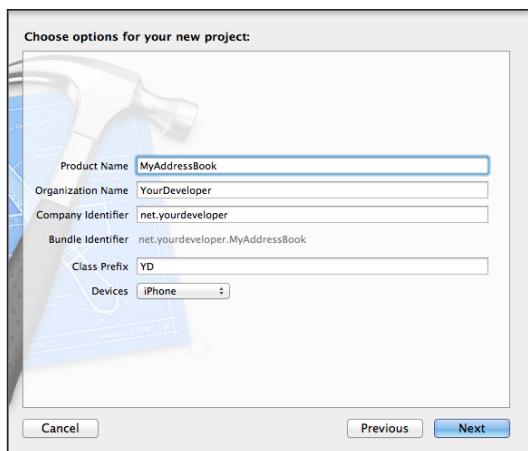
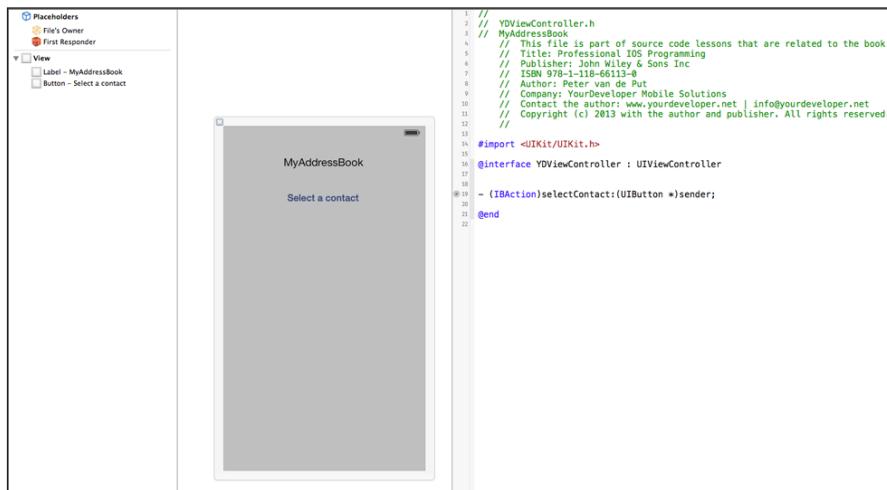


FIGURE 12-1

To access the Address Book, add the AddressBook and the AddressBookUI framework to your project.

Use Interface Builder and the Assistant Editor to create a user interface for the `YDViewController.xib` file, as shown in Figure 12-2.



**FIGURE 12-2**

Open the `YDViewController.m` file and import the `AddressBookUI` header file. Next, subscribe to the `ABPeoplePickerNavigationControllerDelegate` protocol.

In the `selectContact:` method, create and initialize an instance of the `ABPeoplePickerController` class named `peoplePicker`. Because you've subscribed to the `ABPeopleNavigationControllerDelegate` protocol, set the `peoplePickerDelegate` property to `self`. Optionally, you can use the `displayedProperties` property of the `ABPeoplePickerNavigationController` class to pass an `NSArray` with properties you want to display. These properties are wrapped as `NSNumber` objects, which are created from the constants whose names start with the `kABPerson` integer value.

All the available constants are defined in the `ABPerson.h` file.

**WARNING** A very common mistake is to set the delegate property of the peoplePicker instance to self. If you do that none of the delegate methods will be called, since you need to set the peoplePickerDelegate property to self instead.

The presented `peoplePicker` calls one of its delegate methods depending on the user's action:

- If the user taps Cancel, the `peoplePickerNavigationControllerDidCancel:` method is called. In your implementation you should dismiss the peoplePicker.

- If the user selects a person, the `peoplePickerController:shouldContinueAfterSelectingPerson:` method is called to determine if the `peoplePicker` should continue. If you want to dismiss the `peoplePicker` and continue your application's logic with the selected person, you should return `NO` after dismissing the `peoplePicker`. However, if you want to display the selected person you should return `YES`.
- If the user selects a property on the contact detail screen, the `peoplePickerController:shouldContinueAfterSelectingPerson:property:identifier:` method is called. If you want to perform the default action return `YES`, otherwise return `NO`.

When you run the application in the simulator and tap the Select a Contact button, the `ABPeoplePickerController` is presented, as shown in Figure 12-3.

If the `peoplePickerController:shouldContinueAfterSelectingPerson:` method returns `YES`, the details of the selected contact are shown as in Figure 12-4.

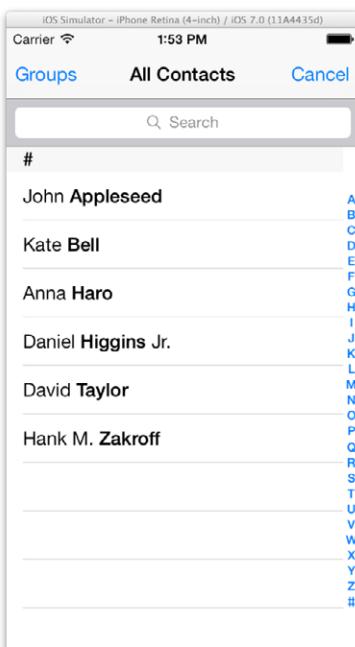


FIGURE 12-3

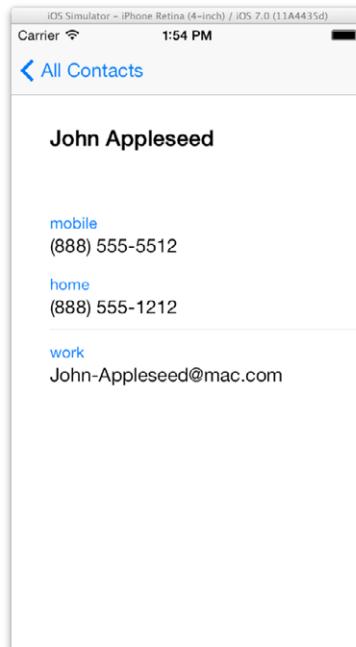


FIGURE 12-4

If the `peoplePickerController:shouldContinueAfterSelectingPerson:property:identifier:` method returns `YES` and you tap a property, the default action is performed. When you tap the work address of John Appleseed, the address is shown on a map, as you can see in Figure 12-5.

Enter a work address for John Appleseed in the Contacts database before trying this since by default he doesn't have a work address.



FIGURE 12-5

## Requesting Access Permission

If you launch the application on an iOS device running iOS 6.x or higher, a UIAlertView will appear when you tap the Select a Contact button. With the introduction of iOS 6.0, Apple has improved security and the application is explicitly requesting you to give permission to the application to access your contact database.

The presented UIAlertView is shown in Figure 12-6.

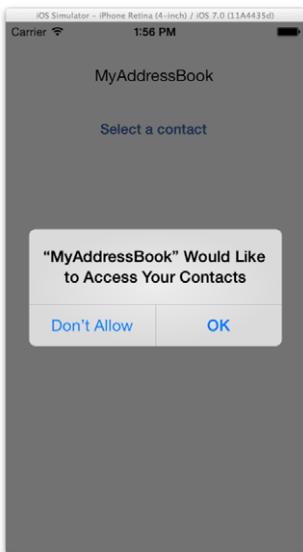


FIGURE 12-6

If you select OK from the prompted `UIAlertView`, your application will run without any problem. If you don't allow access, the application won't run and a default screen is displayed, as shown in Figure 12-7.

A user of your application can always change the access permission to an application by following the `Settings`  $\leftrightarrow$  `Privacy`  $\leftrightarrow$  `Contacts` path on the device, as shown in Figure 12-8.

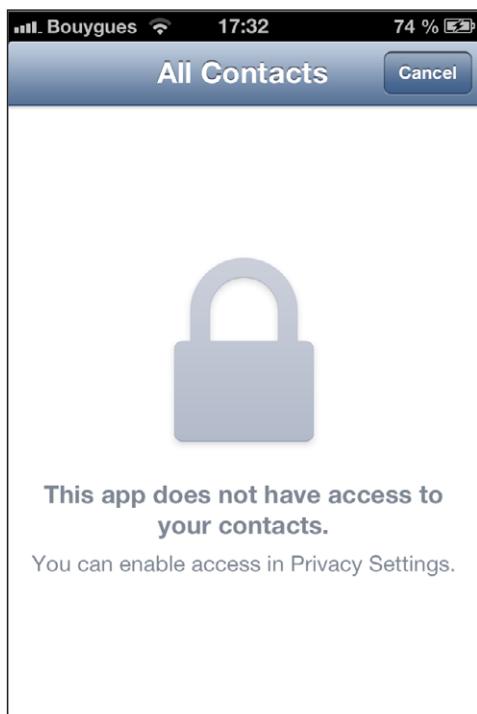


FIGURE 12-7



FIGURE 12-8

The complete `YDViewController.m` implementation is shown in Listing 12-1.

**LISTING 12-1: Chapter12/MyAddressBook/YDViewController.m**

```
#import "YDViewController.h"
#import <AddressBookUI/AddressBookUI.h>

@interface YDViewController : ABPeoplePickerNavigationControllerDelegate

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}
```

```
}

- (IBAction)selectContact:(UIButton *)sender
{
    ABPeoplePickerNavigationController *peoplePicker =
        [[ABPeoplePickerNavigationController alloc] init];
    peoplePicker.peoplePickerDelegate=self;
    //optional to limit the number of properties displayed
    NSArray *displayedItems = [NSArray arrayWithObjects:
        [NSNumber numberWithInt:kABPersonPhoneProperty],
        [NSNumber numberWithInt:kABPersonEmailProperty],
        [NSNumber numberWithInt:kABPersonFirstNameProperty], nil];
    peoplePicker.displayedProperties = displayedItems;

    [self presentViewController: peoplePicker animated:NO completion:nil];
}

#pragma mark delegates
// Called after the user has pressed cancel
// The delegate is responsible for dismissing the peoplePicker
- (void)peoplePickerNavigationControllerDidCancel:
    (ABPeoplePickerNavigationController *)peoplePicker
{
    [self dismissViewControllerAnimated:NO completion:nil];
}

// Called after a person has been selected by the user.
// Return YES if you want the person to be displayed.
// Return NO to do nothing (the delegate is responsible for dismissing the
// peoplePicker).
- (BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController *)
    peoplePicker shouldContinueAfterSelectingPerson:(ABRecordRef)person
{
    //[[self dismissViewControllerAnimated:NO completion:nil];
    return YES;
}
// Called after a value has been selected by the user.
// Return YES if you want default action to be performed.
// Return NO to do nothing (the delegate is responsible for dismissing the
// peoplePicker).
- (BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController *)
    peoplePicker shouldContinueAfterSelectingPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifier
{
    return YES;
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```

@end

You can display, edit, and delete contacts by using the `ABPersonViewController`. It's important to know that the `ABPersonViewController` object must be used with a `UINavigationController` in order to function properly.

## Displaying and Editing a Contact

Start Xcode and create a new project using the Single View Application Project template, and name it `InteractiveAB` using the options shown in Figure 12-9.

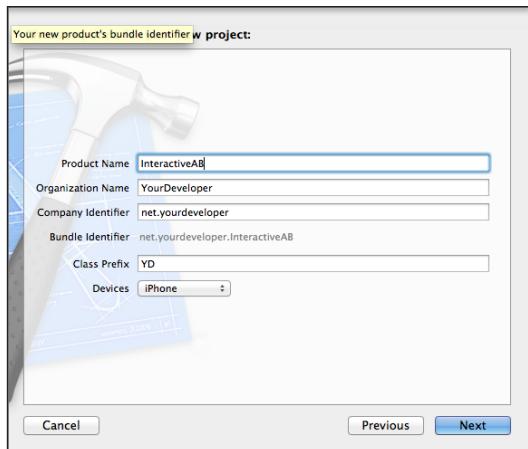


FIGURE 12-9

Remove the `YDViewController` header, implementation, and Interface Builder file from your project. Copy these three files from the `MyAddressBook` project into the `InteractiveAB` project.

Because the `ABPersonViewController` works properly only with a `UINavigationController`, modify the `YDAppDelegate` and add the strong property of type `UINavigationController` as shown in Listing 12-2.

### LISTING 12-2: Chapter12/InteractiveAB/YDAppDelegate.h

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

@class YDViewController;

@interface YDAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong) UINavigationController *navController;
@property (strong, nonatomic) YDViewController *viewController;

@end
```

Open the YDViewController.m file and replace the application:didFinishLaunchingWithOptions: method with the implementation shown in Listing 12-3.

#### LISTING 12-3: application:didFinishLaunchingWithOptions: method

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.viewController = [[YDViewController alloc]
    initWithNibName:@"YDViewController" bundle:nil];
    self.navController = [[UINavigationController alloc]
    initWithRootViewController:self.viewController];
    self.window.rootViewController = self.navController;
    self.window.backgroundColor = [UIColor clearColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Open the YDViewController.m file and subscribe to the ABPersonViewControllerDelegate protocol.

Follow these steps to display and edit a contact by using the ABPersonViewController:

1. Create and initialize an instance of the ABPersonViewController class.
2. Set the personViewDelegate property to self.
3. Set the displayedPerson property to the person record you want to display.
4. Optionally, set the displayedProperties as you did in the MyAddressBook example.
5. Display the ABPersonViewController using the pushViewControllerAnimated: method of your UINavigationController.

Implement a method named showPersonViewController: in which you create an ABPersonViewController instance named pvController. Then set the personViewDelegate to self and set the allowsEditing property. Finally, display the pvController by using the pushViewController:animated: method of your UINavigationController as shown in Listing 12-4.

#### LISTING 12-4: showPersonViewController: method implementation

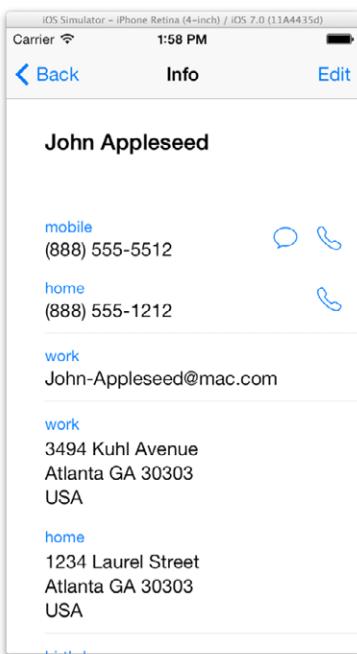
```
- (void)showPersonViewController:(ABRecordRef)person
{
    ABPersonViewController *pvController =
    [[ABPersonViewController alloc] init];
    pvController.personViewDelegate = self;
    pvController.displayedPerson = person;
    // Allow users to edit the information
    pvController.allowsEditing = YES;
    [self.navigationController pushViewController:pvController animated:YES];
}
```

In your `peoplePickerController: shouldContinueAfterSelectingPerson:` method, you first call the `dismissViewControllerAnimated:` method to remove the `ABPeoplePickerController` from the view stack. Call the `showPersonViewController:` method and pass the selected person record as shown in Listing 12-5.

**LISTING 12-5: `peoplePickerController: shouldContinueAfterSelectingPerson:` method**

```
- (BOOL)peoplePickerController:(ABPeoplePickerController *)  
    peoplePicker shouldContinueAfterSelectingPerson:(ABRecordRef)person  
{  
    [self dismissViewControllerAnimated:NO completion:nil];  
    [self showPersonViewController:person];  
    return NO;  
}
```

When you launch the application and tap the Select a Contact button, the `ABPeoplePickerController` is presented. When you select a contact from the list, the `showPersonViewController:` method is called with the selected person, and the `ABPersonViewController` is presented as shown in Figure 12-10.



**FIGURE 12-10**

Tap the `Edit` button to make the changes you want to make, and tap the `Done` button to save these changes to the Address Book database.

## Creating a Contact

To create a new contact in the Address Book, you can use the `ABNewPersonViewController` class, which allows users to create a new contact.

The steps to implement are similar to the steps for displaying and editing a contact. To implement this, follow these steps:

1. Subscribe to the `ABNewPersonViewControllerDelegate` protocol.
2. Create and initialize an instance of the `ABNewPersonViewController` class named `newController`.
3. Set the `newPersonViewDelegate` property to `self`.
4. Create and initialize a new `UINavigationController` named `newNavController` and set its root view controller to the `newController` object you just created.
5. Present the `newNavController` as a modal view using the `presentModalViewControllerAnimated:` method.

Start Xcode and create a new project using the Single View Application Project template, and name it `NewContact` using the options shown in Figure 12-11.

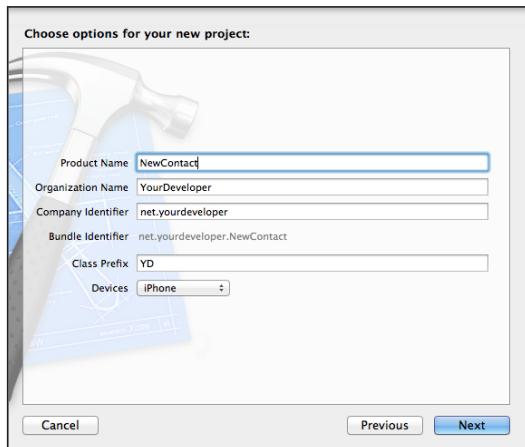


FIGURE 12-11

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor and create a user interface as shown in Figure 12-12.



FIGURE 12-12

Import the AddressBook and the AddressBookUI frameworks in your project and open the `YDViewController.m` file.

Import the AddressBookUI header file and subscribe to the `ABNewPersonViewControllerDelegate` protocol.

Implement the `createNewContact:` method as shown in Listing 12-6, in which you create the `ABNewPersonViewController`, set the `newPersonViewDelegate` property, create a `UINavigationController`, and present it modally.

#### LISTING 12-6: `createNewContact:` method

```

- (IBAction)createNewContact:(UIButton *)sender
{
    ABNewPersonViewController* newPersonController =
        [[ABNewPersonViewController alloc] init];
    newPersonController.newPersonViewDelegate=self;
    UINavigationController* newNavController =
        [[UINavigationController alloc]
         initWithRootViewController:newPersonController];
    [self presentViewController:newNavController animated:YES completion:nil];
}

```

When the user taps the Save or Cancel button, the `newPersonController` calls the method `newPersonViewController:didCompleteWithNewPerson:`, which you need to implement.

Dismiss the ViewController by calling the `dismissViewControllerAnimated:completion:` method. If the user has tapped the Cancel button, the value of `person` is `NULL`; otherwise, it will represent the person record just created.

When you launch the application and tap the Create a Contact button, the application will display the ABNewPersonViewController as shown in Figure 12-13.

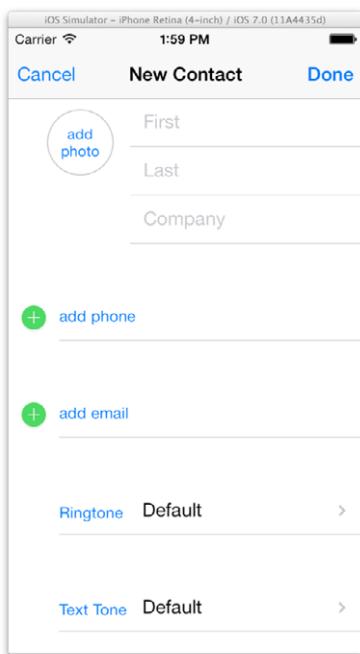


FIGURE 12-13

## Deleting a Contact

Deleting a contact from the Address Book database is not possible without programmatically accessing the Address Book. This is covered later in this chapter.

So far you've been using classes and methods of the ABAddressBookUI framework to display, create, and edit contacts in the Address Book database. In the next section you learn how to programmatically access the Address Book without using the user interface element of the ABAddressBookUI framework.

## PROGRAMMATICALLY ACCESSING THE ADDRESS BOOK

In some scenarios you want your application to use the contacts from the Address Book database, but you want to use your custom `UIViewController` and `UIView` objects rather than the Address Book user interface components to display or capture the contact details. Another scenario is that you need to make use of the contacts in the Address Book database and want to store the details in Core Data with additional properties and/or relationships.

In those cases you need to access the Address Book database programmatically and write custom logic to access the individual contact records and properties.

Start Xcode and create a new project using the Single View Application Project template, and name it ProgAB using the options shown in Figure 12-14.

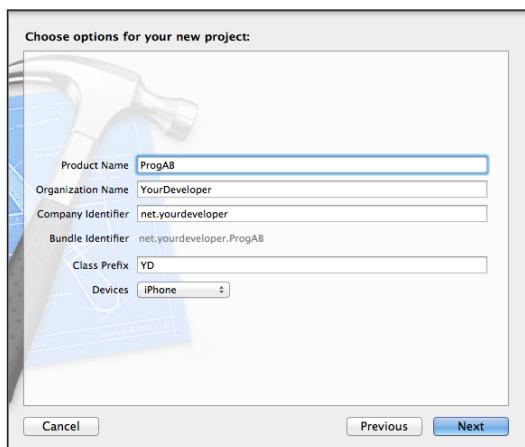


FIGURE 12-14

Add the AddressBook and AddressBookUI frameworks to your project.

Four basic objects are involved in the interaction with the Address Book:

- Address Books
- Records
- Single-value properties
- Multi-value properties

These four basic objects are explained in detail in the following sections.

## Understanding Address Books

The `ABAddressBookRef` object is the object you use to access the Address Book database. You need to create an instance of the `ABAddressBookRef` object using the `ABAddressBookCreateWithOptions` function, as shown in Listing 12-7.

### LISTING 12-7: Creating an ABAddressBookRef instance

```
CFErrorRef error = NULL;  
ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);
```

**WARNING** Instances of ABAddressBookRef can't be used by multiple threads.  
Each thread needs to create its own instance of ABAddressBookRef.

## Request Access Permission

As you learned earlier, the user needs to give explicit access to your application to access the Address Book.

When you want to access the Address Book programmatically, you have a method containing your application's logic to select or manipulate records. To guarantee a flawless operation of your method, you need to ensure that the user has given access to the application to use the Address Book.

**WARNING** The user always has the option to change access to the Address Book for your application. If you don't explicitly check if the application has been authorized, your application might crash—something you need to avoid under all circumstances.

Use the ABAddressBookRequestAccessWithCompletion function, which will prompt the user for access to his Address Book. You can use the ABAddressBookGetAuthorizationStatus function to check the authorization status for accessing the Address Book.

This function returns four possible values:

- kABAAuthorizationStatusNotDetermined
- kABAAuthorizationStatusDenied
- kABAAuthorizationStatusAuthorized
- kABAAuthorizationStatusRestricted

When the user didn't authorize access to his Address Book, you can use a UIAlertView to notify the user with an informational message about what will happen, such as the application can't access the Address Book and therefore will not operate with all functionality.

Listing 12-8 shows an implementation of the logic to request access and check the result using the ABAddressBookGetAuthorizationStatus function.

### LISTING 12-8: Asking for authorization

```
— block BOOL accessGranted = NO;
    if (ABAddressBookRequestAccessWithCompletion != NULL)
    {
        ABAddressBookRequestAccessWithCompletion(addressBook,
            ^(bool granted, CFErrorRef error)
        {
            accessGranted = granted;
        });
    }
}
```

*continues*

**LISTING 12-8** (continued)

```

        dispatch_async(dispatch_get_main_queue(), ^{
            accessGranted=YES;
        });
    });
}

if (ABAddressBookGetAuthorizationStatus() ==
    kABAAuthorizationStatusNotDetermined)
{
    //Access is not determined
}
else if (ABAddressBookGetAuthorizationStatus() ==
    kABAAuthorizationStatusDenied)
{
    //Access is denied
}
else if (ABAddressBookGetAuthorizationStatus() ==
    kABAAuthorizationStatusAuthorized)
{
    //Access is authorized
}
else if (ABAddressBookGetAuthorizationStatus() ==
    kABAAuthorizationStatusRestricted)
{
    //Access is restricted
}

```

**Saving or Abandoning Address Book Changes**

After you have created the `ABAddressBookRef` instance and checked the authorization, you can read and write to the Address Book database. Once you're done with your operations you can either save the changes you've made or abandon them. Using the `ABAddressBookHasUnsavedChanges` function you can check if changes have been made to the Address Book from your application. If you want to abandon the changes, you call the `ABAddressBookRevert` function, and if you want to save the changes, you call the `ABAddressBookSave` function. Both functions are shown in Listing 12-9.

**LISTING 12-9:** Save or abandon changes

```

if (ABAddressBookHasUnsavedChanges(addressBook) )
{
    //save changes
    BOOL didSave = ABAddressBookSave(addressBook, &error);
    if (!didSave)
    {
        //handle the error here
    }
    //abandon changes
    ABAddressBookRevert(addressBook);
}

```

## Stay Synchronized

Once your application has been authorized by the user to access the Address Book, you can read and write to the Address Book database. Other applications also might have created an instance to the Address Book, and performed changes to the Address Book database (for example, in a background operation).

Use the `ABAddressBookRegisterExternalChangeCallback` function to register a function of the prototype `ABExternalChangeCallBack` that will handle changes in the Address Book from another application.

It's possible to create multiple callback functions by calling the `ABAddressBookRegisterExternalChangeCallback` function multiple times with a different function.

You can unregister the callback function by using the `ABAddressBookUnregisterExternalChangeCallback` function.

In your callback function you can decide to save your changes or to abandon your changes—this is completely up to your application's logic and requirements. A sample callback function is shown in Listing 12-10.

### LISTING 12-10: addressBookChanged function

```
void addressBookChanged(ABAddressBookRef reference,
                       CFDictionaryRef dictionary,
                       void *context)
{
    //Something has changed perform your action
}
```

**WARNING** *When your application is running in the background and you've set up a callback function, it is called only when your application returns to the foreground.*

## Understanding Records

All the contact information in the Address Book is stored in records. A record in the Address Book database is of type `ABRecordRef` and contains a unique record identifier, which can be retrieved with the `ABRecordGetRecordID` function. It is not safe to pass records across threads; instead, you pass the record identifier. Each record represents a person or group. You can determine the type of record using the `ABRecordGetRecordType` function, because a person record will return the value `kABPersonType` and a group will return the value `kABGroupType`.

The actual data of a record is stored in a collection of properties. The available properties for a person record are different from the available properties for a group record.

## Person Records

A person record represents a person and is made up of two types of properties. Properties such as a first name or a last name (of which a person has only one) are stored as single-value properties. Properties like a phone number (of which a person can have more than one) are multi-value properties.

## Group Records

A user can use group records to organize his Address Book database by creating groups. A group record only has one single-value property named `kABGroupNameProperty`, which holds the name of the group.

To get all person objects related to a group record, use the `ABGroupCopyArrayOfAllMembers` or `ABGroupCopyArrayOfAllMembersWithSortOrdering` function, which both return a `CFArrayRef` of `ABRecordRef` objects.

## Understanding Properties

`ABRecord` objects contain property, which you can set, copy or remove.

### Single-Value Properties

Three functions are available to work with the properties of a record:

- `ABRecordCopyValue`: This is the getter for the property.
- `ABRecordSetValue`: This is the setter for the property.
- `ABRecordRemoveValue`: This is used to delete a property.

The following code shows how to access the first name property of a person record:

```
CFStringRef firstName;
//assumption aRecord is a valid ABRecordRef object
firstName = ABRecordCopyValue(aRecord, kABPersonFirstNameProperty);
```

The following code sets the value Peter to the first name property of a person record:

```
NSString* firstname = @"Peter";
ABRecordSetValue(aRecord, kABPersonFirstNameProperty,
(__bridge CFTypeRef)(firstname), &error);
```

### Multi-Value Properties

Multi-value properties consist of a list of values. Each value has an identifier and a text label. There can be multiple values with the same label, but the identifier is always unique. Some generic property labels are defined in the `ABPerson.h` file, which are `kABWorkLabel`, `kABHomeLabel`, and `kABOtherLabel`.

Individual values of multi-value properties are accessible by identifier or by index. Use the functions `ABMultiValueGetIndexForIdentifier` and `ABMultiValueGetIdentifierAtIndex` to convert between indices and multi-value identifiers. To keep a reference to a particular value, store and use the identifier because the index might change if values are added or removed.

The following code shows how to access the multi-value phone property of an `ABRecordRef` instance:

```
ABRecordRef aRecord;
//get the Phone Property which is a multi value property
ABMultiValueRef phones = ABRecordCopyValue(aRecord, kABPersonPhoneProperty);
//ABMultiValueGetCount returns the number of property values
//in the collection
for(CFIndex j = 0; j < ABMultiValueGetCount(phones); j++)
{
    //
    CFStringRef phoneNumberRef = ABMultiValueCopyValueAtIndex(phones, j);
    CFStringRef locLabel = ABMultiValueCopyLabelAtIndex(phones, j);
    NSString *phoneLabel = (__bridge NSString*)
        ABAddressBookCopyLocalizedLabel(locLabel);
    NSString *phoneNumber = (__bridge NSString *)phoneNumberRef;
    //do something with the values
}
```

Multi-value objects are immutable; to change a multi-value property you make a mutable copy using the function `ABMultiValueCreateMutableCopy`. You can also create a new mutable multi-value object using the function `ABMultiValueCreateMutable`.

## Creating a Contact Programmatically

To create a new contact programmatically, start by creating the `ABAddressBookRef` instance named `addressBook` and make sure you're granted access to the Address Book.

Create an `ABRecordRef` instance named `newPerson` using the `ABPersonCreate` function. Set the values for the first name and the last name properties. To add a mobile phone number, create an instance of `ABMutableMultiValueRef` named `multiPhone` and set the property value for the `kABPersonPhoneMobileLabel`. Finally, call the `ABAddressbookAddRecord` function to add the record to the Address Book. You must realize that although you've added the new record to the Address Book, it's not saved yet because you didn't call the `ABAddressBookSave` function, which is the final thing to do. A sample implementation is shown in Listing 12-11.

### LISTING 12-11: Creating a new contact

```
CFErrorRef error = NULL;
ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);
__block BOOL accessGranted = NO;
if (ABAddressBookRequestAccessWithCompletion != NULL)
{
    ABAddressBookRequestAccessWithCompletion(addressBook,
        ^(bool granted, CFErrorRef error)
    {
        accessGranted = granted;
        dispatch_async(dispatch_get_main_queue(), ^{
            accessGranted=YES;
        });
    });
}
```

*continues*

**LISTING 12-11** (continued)

```

ABRecordRef newPerson = ABPersonCreate();
//add firstname
ABRecordSetValue(newPerson, kABPersonFirstNameProperty,
(_bridge CFTypeRef) (@"Peter"), &error);
//add lastname
ABRecordSetValue(newPerson, kABPersonLastNameProperty,
(_bridge CFTypeRef) (@"van de Put"), &error);
//add mobile phone number
ABMutableMultiValueRef multiPhone =
    ABMultiValueCreateMutable(kABMultiStringPropertyType);
ABMultiValueAddValueAndLabel(multiPhone,
    (_bridge CFTypeRef) (@"+3311111111"),
    kABPersonPhoneMobileLabel, NULL);
ABRecordSetValue(newPerson, kABPersonPhoneProperty, multiPhone, nil);

//Add the record to the address book
ABAddressBookAddRecord(addressBook, newPerson, &error);
//Save the changes
ABAddressBookSave(addressBook, &error);
if (error != NULL)
{
    //handle your error here
}

```

## Selecting One or More Contacts

To select all contacts from the Address Book you can use the `ABAddressBookCopyArrayOfAllPeople` function, which returns a `CFArrayRef`. You can determine the number of records in the result by using the `ABAddressBookGetPersonCount` function.

Now create a loop to create an `ABRecordRef` instance for each record in the array by using the `CFArrayGetValueAtIndex` function. A sample implementation is shown in Listing 12-12.

**LISTING 12-12:** The `loadAllPeople` implementation

```

- (void)loadAllPeople
{
    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);
    _block BOOL accessGranted = NO;
    if (ABAddressBookRequestAccessWithCompletion != NULL)
    {
        ABAddressBookRequestAccessWithCompletion(addressBook,
            ^(bool granted, CFErrorRef error)
            {
                accessGranted = granted;

                dispatch_async(dispatch_get_main_queue(), ^{
                    accessGranted=YES;
                });
            }
        );
    }
}

```

```

        });
    }

CFArrayRef allPeople = ABAddressBookCopyArrayOfAllPeople(addressBook);
CFIndex nPeople = ABAddressBookGetPersonCount(addressBook);
for( int i = 0 ; i < nPeople ; i++ )
{
    ABRecordRef ref = CFArrayGetValueAtIndex(allPeople, i );
    ABRecordType rectype = ABRecordGetRecordType(ref);
    //check if record is a person record
    if (rectype == kABPersonType)
    {
        //do what you want to do
    }
}
//release the addressBook
CFRelease(addressBook);
}

```

To select a series of contacts, create an `NSArray` named `allContacts` by calling the `ABAddressBookCopyArrayOfAllPeople` function. Create an `NSPredicate` to filter this array using the `filteredArrayUsingPredicate:` method of the `NSArray` class, as shown in Listing 12-13.

#### LISTING 12-13: A filterContacts implementation

```

-(void)filterContacts
{
    CFErrorRef error = NULL;
    ABAddressBookRef addressBook = ABAddressBookCreateWithOptions(NULL, &error);
    __block BOOL accessGranted = NO;
    if (ABAddressBookRequestAccessWithCompletion != NULL)
    {
        ABAddressBookRequestAccessWithCompletion(addressBook,
        ^(bool granted, CFErrorRef error)
        {
            accessGranted = granted;

            dispatch_async(dispatch_get_main_queue(), ^{
                accessGranted=YES;
            });
        });
    }
    //Create an NSArray containing all contacts
    NSArray* allContacts= (__bridge NSArray *)
        (ABAddressBookCopyArrayOfAllPeople(addressBook));
    // Build a predicate that searches for contacts with at least one phone
    // number starting with +33.
    NSPredicate* predicate = [NSPredicate predicateWithBlock: ^(id record,
        NSDictionary* bindings) {
        ABMultiValueRef phoneNumbers =
        ABRecordCopyValue( __bridge ABRecordRef)record,
        kABPersonPhoneProperty);

```

*continues*

**LISTING 12-13 (continued)**

```

BOOL result = NO;
for (CFIndex i = 0; i < ABMultiValueGetCount(phoneNumbers); i++) {
    NSString* phoneNumber = (_bridge_transfer NSString*)
        ABMultiValueCopyValueAtIndex(phoneNumbers, i);
    if ([phoneNumber hasPrefix:@"+33"]) {
        result = YES;
        break;
    }
}
CFRelease(phoneNumbers);
return result;
}];
//release the addressBook
CFRelease(addressBook);
}

```

## Deleting a Contact Programmatically

To delete a contact from the Address Book, you simply use the `ABAddressBookRemoveRecord` function. Again you must realize that although you've deleted the record from the Address Book, it's not yet physically removed from the Address Book database because you didn't call the `ABAddressBookSave` function, which is the final thing to do. The next example shows how to delete the `aPerson` record from the Address Book, assuming that `aPerson` is a valid initialized record:

```

ABAddressBookRemoveRecord(addressBook, aPerson, &error);
//Save the changes
ABAddressBookSave(addressBook, &error);
if (error != NULL)
{
    //handle your error here
}

```

## SUMMARY

In this chapter you learned how to work with the Address Book. You are now capable of presenting the Address Book, selecting a person, and editing or removing a person both programmatically and by using the user interface elements from the framework.

In the next chapter you learn how to access and work with the Event Kit framework, and work with calendars and reminders.

# 13

## Event Programming

---

### WHAT'S IN THIS CHAPTER?

---

- Working with calendars and reminders
- Programmatically accessing the event store where the calendars and reminders are stored

### WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 13 download and individually named according to the names throughout the chapter.

### INTRODUCTION TO THE EVENT KIT FRAMEWORK

The Event Kit framework provides you access to the user's Calendar and Reminders information. The native iOS applications Calendar and Reminders use the same database for storing their data, which is called the Calendar database. In programming terms this is known as an *event store*.

*Events* refer to calendar items and reminders.

The Event Kit technology consists of the following two parts:

- The EventKit framework
- The EventKitUI framework, which provides user interface elements for manipulating events

You can find the Calendar and Reminders programming guide at [http://developer.apple.com/library/ios/#documentation/DataManagement/Conceptual/EventKitProgGuide/Introduction/Introduction.html%23//apple\\_ref/doc/uid/TP40009765](http://developer.apple.com/library/ios/#documentation/DataManagement/Conceptual/EventKitProgGuide/Introduction/Introduction.html%23//apple_ref/doc/uid/TP40009765).

The Event Kit framework reference guide is available at [http://developer.apple.com/library/ios/#documentation/EventKit/Reference/EventKitFrameworkRef/\\_index.html](http://developer.apple.com/library/ios/#documentation/EventKit/Reference/EventKitFrameworkRef/_index.html).

The Event Kit framework enables you to read the user's calendar database and also modify it by creating or editing events. To stay synchronized between applications, your application can create an observer to receive notifications from the Event Kit framework, in case the event store is modified by another application.

## USING THE EVENTKITUI FRAMEWORK

The EventKitUI framework provides you with the following user interface elements to work with events:

- `EKCalendarChooser`, to let the user select one or more calendars
- `EKEventViewController`, to create or edit an existing event

It's important to realize that a user can have multiple calendars. In the device's settings a default calendar can be set under Settings ➔ "Mail, Contacts, Calendars."

Start Xcode and create a new project using the Single View Application Project template, and name it `MyEvents`, using the options shown in Figure 13-1.

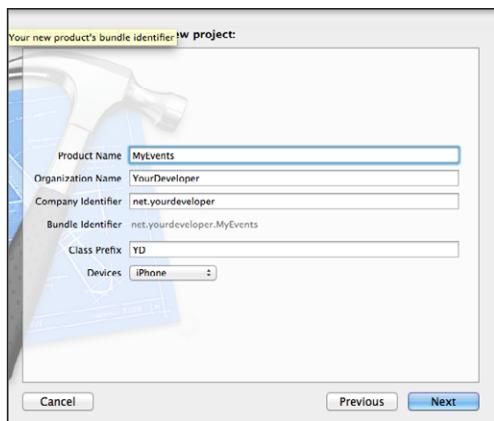


FIGURE 13-1

Start by adding the EventKit and the EventKitUI frameworks to your project.

## Requesting Access Permission

If you run the application, a `UIAlertView` will appear. With the introduction of iOS 6.0, Apple has improved security, and the user is explicitly asked to give permission to your application to access the Calendar database. You deal with this in a very similar way to how you access the Address Book database.

Listing 13-1 shows how to prompt the user to authorize your application to access the Calendar database. It creates and initializes an instance of the `EKEventStore`, and the `requestAccessToEntityType:completion:` method of the `EKEventStore` object is called in a block. You can pass two constants to the `requestAccessToEntityType:`:

- `EKEntityTypeReminder` to ask authorization to access the reminders in the Calendar database
- `EKEntityTypeEvent` to ask for authorization to access the events in the Calendar database

If your application requires access to both the reminders and the events, you have to call `requestAccessToEntityType:completion:` twice, once for each entity type.

The user can always change the access permission to an application by following the Settings ⇔ Privacy ⇔ Calendars path or the Settings ⇔ Privacy ⇔ Reminders path on the device.

#### **LISTING 13-1: `requestAccessToCalendar` for reminders**

```
- (void)requestAccessToCalendar
{
    self.myEventStore = [[EKEventStore alloc] init];
    __block BOOL accessGranted = NO;
    [self.myEventStore requestAccessToEntityType:EKEntityTypeEvent
        completion:^(BOOL granted, NSError *error)
    {
        // handle access here
        accessGranted = granted;
        dispatch_async(dispatch_get_main_queue(), ^{
            accessGranted=YES;
        });
    }];
}
```

Before you interact with the `EKEventStore` it's important to check if the user has indeed granted your application access to the Calendar database. You can perform a simple check by calling the `authorizationStatusForEntityType:` method of the `EKEventStore` class, as shown in Listing 13-2.

#### **LISTING 13-2: Checking authorization**

```
EKAutorizationStatus status = [EKEventStore
    authorizationStatusForEntityType:EKEntityTypeEvent];
if (status == EKAutorizationStatusDenied ||
    status == EKAutorizationStatusRestricted) {

    return;
}
```

This method returns one of the following four constants:

- EKAutorizationStatusNotDetermined
- EKAutorizationStatusRestricted
- EKAutorizationStatusDenied
- EKAutorizationStatusAuthorized

## Accessing a Calendar

Once your application has been authorized to access the user's Calendar database, you can start creating and editing events.

It's important to realize that the Calendar database can contain multiple calendars. A user can decide to create a Personal calendar for personal events and a Work calendar for all work-related events.

The first step, therefore, is to either choose the default calendar or ask the user explicitly to allow your application to select the calendar you want to work with or choose the default calendar.

Open the YDViewController.xib file using Interface Builder and the Assistant Editor to create a simple user interface with a UIButton and a UILabel, as shown in Figure 13-2.

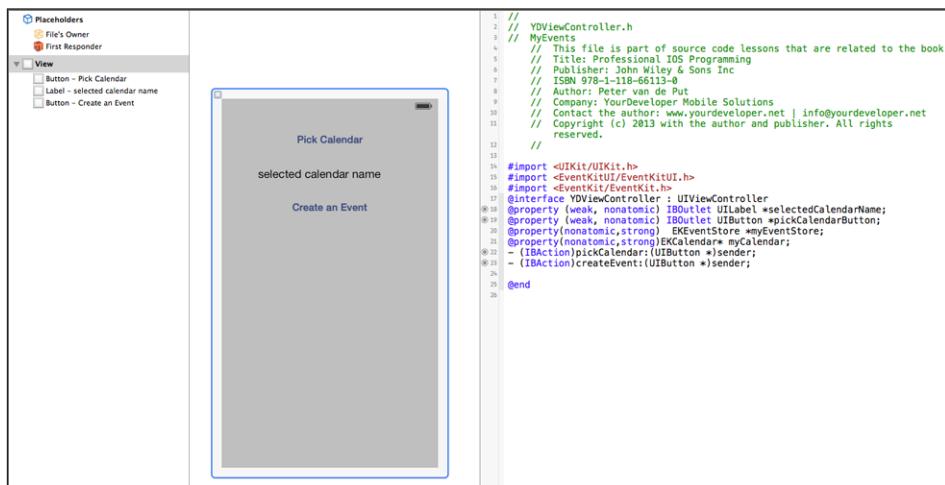


FIGURE 13-2

Open the YDViewController.h file and create a strong property of type EKEEventStore named myEventstore. To hold the selected calendar chosen by the user, create a strong property of type EKCalendar named myCalendar as shown in Listing 13-3.

**LISTING 13-3:** Chapter13/MyEvents/YDViewController.h

```
#import <UIKit/UIKit.h>
#import <EventKitUI/EventKitUI.h>
#import <EventKit/EventKit.h>
@interface YDViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *selectedCalendarName;
@property (weak, nonatomic) IBOutlet UIButton *pickCalendarButton;
@property (nonatomic, strong) EKEventStore *myEventStore;
@property (nonatomic, strong) EKCalendar* myCalendar;
- (IBAction)pickCalendar:(UIButton *)sender;

@end
```

Now open the `YDViewController.m` file and subscribe to the `EKCalendarChooserDelegate` protocol.

Next in the `viewDidLoad:` method, call the `requestToAccessCalendar` method to ask for the authorization.

Implement the `pickCalendar:` method as shown in Listing 13-4. It starts by checking whether the application has been authorized by the user to access the event store. Notice that the `authorizationStatusForEntityType:` method of the `EKEventStore` class is a static method, so call `[EKEventStore authorizationStatusForEntityType:EKEEntityTypeEvent]` instead of `[self.myEventStore authorizationStatusForEntityType:EKEEntityTypeEvent]`.

Next, create and initialize an instance of the `EKCalenderChooser` object named `calendarChooser` and set the properties.

Finally, create a `UINavigationController`, which you use to present the `calendarChooser`. The complete method is shown in Listing 13-4.

**LISTING 13-4:** The `pickCalendar:` method

```
- (IBAction)pickCalendar:(UIButton *)sender
{
    EKAutorizationStatus status = [EKEventStore
        authorizationStatusForEntityType:EKEEntityTypeEvent];
    if (status == EKAutorizationStatusDenied ||
        status == EKAutorizationStatusRestricted) {

        return;
    }
    EKCalendarChooser *calendarChooser = [[EKCalendarChooser alloc]
        initWithSelectionStyle:EKCalendarChooserSelectionStyleSingle
        displayStyle:EKCalendarChooserDisplayStyleAllCalendars
        eventStore:self.myEventStore];
    //if you don't set the selectedCalendars to an initialized NSSet it always
    returns nil
```

*continues*

**LISTING 13-4 (continued)**

```

calendarChooser.selectedCalendars = [[NSSet alloc] init];
calendarChooser.showsDoneButton = YES;
calendarChooser.showsCancelButton = YES;
calendarChooser.delegate = self;

UINavigationController* newNavController = [[UINavigationController alloc]
    initWithRootViewController:calendarChooser];
[self presentViewController:newNavController animated:YES completion:nil];

}

```

Because you've subscribed to the `EKCalendarChooserDelegate` protocol, you need to implement the following three delegate methods:

- `calendarChooserSelectionDidChange:`
- `calendarChooserDidFinish:`
- `calendarChooserDidCancel:`

The `calendarChooserDidFinish:` and `calendarChooserDidCancel:` methods are called if the user selects the Done or the Cancel button. Whether they appear or not depends on the value of the `showsDoneButton` and `showsCancelButton` properties of the `calendarChooser` instance. The Done and Cancel buttons only show if the respective properties are set to true and the selection style has been set to `EKCalendarChooserSelectionStyleMultiple`. If you set the selection style to `EKCalendarChooserSelectionStyleSingle`, tapping a calendar from the presented list automatically calls the `calendarChooserSelectionDidChange:` method of the `EKCalendarChooser` class.

## Creating and Editing a Calendar Event

To create or edit a calendar event, you use the `EKEVENTEditViewController` object. Open the `YDviewController.xib` file using Interface Builder and the Assistant Editor and add a new `UIButton` to the user interface with the label Create Event and an `IBAction` `createEvent:`. Now open the `YDViewController.m` file, subscribe to the `EKEVENTEditViewDelegate` protocol, and implement the `createEvent:` method. Create and initialize an instance of `EKEVENTEditViewController` named `addEventController`. Set the `addEventController`'s `eventStore` property to `myEventStore` and the `editViewDelegate` property to `self`.

Present the `addEventController` controller. If you want to edit an existing event, you need to pass an instance of the event to the `event` property, otherwise set the `event` property to `nil`.

The `createEvent:` method is shown in Listing 13-5.

**LISTING 13-5: The createEvent: method**

```

- (IBAction)createEvent:(UIButton *)sender
{
    EKEEventEditViewController *addEventController =
        [[EKEEventEditViewController alloc]
            initWithNibName:nil bundle:nil];

    // set the addController's event store to the current event store.
    addEventController.eventStore = self.myEventStore;
    addEventController.editViewDelegate = self;
    //If you have an existing event named myEvent and want to edit it pass it to
    the event property
    //addEventController.event = myEvent;
    // present EventsAddViewController as a modal view controller
    [self presentViewController:addEventController animated:YES completion:nil];

}

```

The `EKEEventEditViewController` delegate protocol has one required delegate method named `eventEditViewController:didCompleteWithAction:` and one optional method named `eventEditViewControllerDefaultCalendarForNewEvents:`.

You need to implement the mandatory `eventEditViewController:didCompleteWithAction:` delegate method. This method returns one of the following `EKEEventEditViewAction` constant values to inform you if the event edit has been canceled, deleted, or saved by the user:

- `EKEEventEditViewActionCanceled`
- `EKEEventEditViewActionSaved`
- `EKEEventEditViewActionDeleted`

You are responsible for dismissing the presented ViewController by calling `[self dismissViewControllerAnimated:YES completion:nil]`.

You can find the complete source for this project in the download for this chapter.

## PROGRAMMATICALLY ACCESSING THE CALENDAR DATABASE

In the previous section you learned the basics of interacting with the Calendar database and using the EventKitUI framework user interface elements to interact with the user. In this section you learn how to programmatically create an event, a reminder, and query the Calendar database.

## Creating an Event

Start Xcode and create a new project using the Single View Application Project template, and name it MyCalDB using the options shown in Figure 13-3.

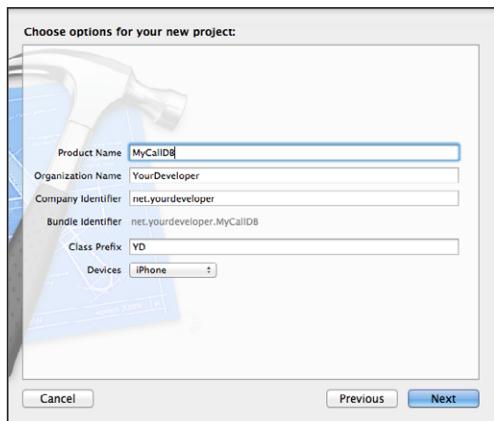


FIGURE 13-3

Add the EventKit and the EventKitUI frameworks to your project.

Open your YAppDelegate.h file and create a strong property of type UINavigationController named navController.

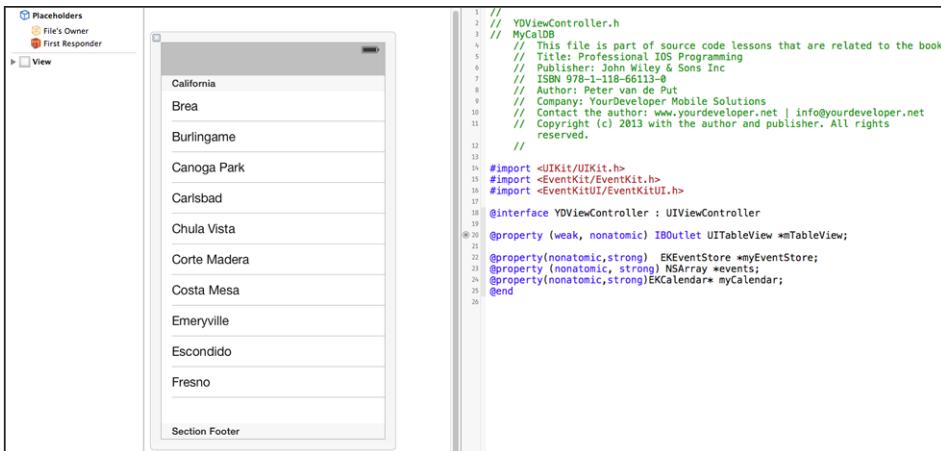
Open your YAppDelegate.m file and change the application:didFinishWithLaunchingOptions: method, as shown in Listing 13-6.

### LISTING 13-6: The application:didFinishWithLaunchingOptions: method

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.viewController = [[YDViewController alloc]
                           initWithNibName:@"YDViewController" bundle:nil];
    self.navController = [[UINavigationController alloc]
                           initWithRootViewController:self.viewController];

    self.window.rootViewController = self.navController;
    self.window.backgroundColor = [UIColor clearColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Now open the YDViewController.xib file using Interface Builder and the Assistant Editor and set up a user interface with a UITableView as shown in Figure 13-4.



**FIGURE 13-4**

Open the `YDViewController.h` file and import the `EventKit` and the `EventKitUI` header files. Create strong properties as shown in Listing 13-7.

**LISTING 13-7:** Chapter13/MyCalDB/YDViewController.h

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

@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;

@property (nonatomic, strong) EKEventStore *myEventStore;
@property (nonatomic, strong) NSArray *events;
@property (nonatomic, strong) EKCalendar* myCalendar;
@end
```

Now open the `YDViewController.m` file and implement the `requestAccessCalendar` method as used in the previous program. Create a `loadEvents` method that uses an `NSPredicate` to query the `myEventStore` for events between now and 24 hours from now.

Implement the `addEvent:` method, which programmatically creates an event in the default calendar for the `myEventStore`. Create and initialize an `EKEvent` instance named `newEvent` and set the properties for the title, notes, calendar, start date, and end date, and call the `saveEvent:span:commit:` method of the `myEventStore` instance.

The complete implementation is shown in Listing 13-8.

**LISTING 13-8:** Chapter13/MyCalDB/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()<UITableViewDataSource, UITableViewDelegate>

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self requestAccessToCalendar];
    //Create an Add Event button
    UIBarButtonItem *addEventButtonItem =
        [[UIBarButtonItem alloc] initWithBarButtonSystemItem:
            UIBarButtonSystemItemAdd
            target:self
            action:@selector(addEvent:)];
    self.navigationItem.rightBarButtonItem = addEventButtonItem;
}

- (void)requestAccessToCalendar
{
    self.myEventStore = [[EKEventStore alloc] init];
    __block BOOL accessGranted = NO;
    [self.myEventStore requestAccessToEntityType:EKEntityTypeEvent
        completion:^(BOOL granted,
                    NSError *error) {
        // handle access here
        accessGranted = granted;
        dispatch_async(dispatch_get_main_queue(), ^{
            accessGranted=YES;
            //call loadEvents here now you now the user has granted access
            [self loadEvents];
        });
    }];
}

- (void)loadEvents
{
    EKAutorizationStatus status = [EKEventStore
        authorizationStatusForEntityType:EKEntityTypeEvent];
    if (status == EKAutorizationStatusDenied ||
        status == EKAutorizationStatusRestricted) {

        return;
    }
    if (self.events)
        self.events=nil;

    self.myCalendar = [self.myEventStore defaultCalendarForNewEvents];
```

```

NSDate *startDate = [NSDate date];
NSDate *endDate = [NSDate dateWithTimeIntervalSinceNow:86400];
// Create the predicate. Pass it the default calendar.
NSArray *calendarArray = [NSArray arrayWithObject:self.myCalendar];
NSPredicate *predicate = [self.myEventStore
                           predicateForEventsWithStartDate:startDate
                           endDate:endDate
                           calendars:calendarArray];

// Fetch all events that match the predicate and store in self.events
self.events = [[NSArray alloc] initWithArray:
               [self.myEventStore eventsMatchingPredicate:predicate]];
[self.tableView reloadData];
}

- (void)addEvent:(id)sender
{
    //Add an event without UI
EKEvent *newEvent = [EKEvent eventWithEventStore:self.myEventStore];
newEvent.calendar = self.myCalendar;
newEvent.title = @"Finalize chapter 13";
newEvent.notes = @"Finish learning chapter 13 of the book Professional
                  iOS programming";
newEvent.startDate = [NSDate date];
newEvent.endDate = [[NSDate date]
                   initWithTimeInterval:600
                   sinceDate:newEvent.startDate];
NSError *err=nil;
[self.myEventStore saveEvent:newEvent span:EKSpanThisEvent
                     commit:YES error:&err];
if (err)
{
    //Handle errors here
}
[self loadEvents];
}

#pragma mark Table View

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

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCellAccessoryType editableCellAccessoryType
        =UITableViewCellAccessoryDisclosureIndicator;
    UITableViewCell *cell = [tableView

```

*continues*

**LISTING 13-8** (continued)

```

dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                reuseIdentifier:CellIdentifier];
}
cell.accessoryType = editableCellAccessoryType;
// Get the event at the row selected and display it's title
cell.textLabel.text = [[self.events objectAtIndex:indexPath.row] title];
return cell;
}
-(void)viewDidAppear:(BOOL)animated
{
    [self loadEvents];
}
#pragma mark UITableView delegates

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath {
    // Upon selecting an event, create an EKEventViewController to display the
    // event.
    EKEventViewController* editController =
    [[EKEventViewController alloc] initWithNibName:nil bundle:nil];
    editController.event = [self.events objectAtIndex:indexPath.row];
    // Allow event editing.
    editController.allowsEditing = YES;
    [self.navigationController pushViewController:editController animated:YES];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

## Editing an Event

Using the `EKEventViewController` to edit an event is the easy way to use the user control elements of the `EventKitUI` framework. If you want to edit an existing event, simply change the properties of the event and call the `saveEvent:span:commit:` method of the `myEventStore`. A sample method implementation is shown in Listing 13-9.

**LISTING 13-9:** An `editEvent:` method

```

- (void)editEvent:(EKEvent* )theEvent
{
    theEvent.calendar = self.myCalendar;

```

```
theEvent.title      = @"Changed event title";
NSError *err=nil;
[self.myEventStore saveEvent:theEvent span:EKSpanThisEvent
commit:YES error:&err];
if (err)
{
    //Handle errors here
}

}
```

## Deleting an Event

Delete an event by simply passing the event you want to delete to the `removeEvent:span:commit:error:` method of the `myEventStore` instance.

## Stay Synchronized

Other applications can make changes to the Calendar database when running in the background. To stay synchronized, the EventKit framework sends an `NSNotification` each time a change is applied to the Calendar database.

To stay synchronized, set up an observer to receive the notifications broadcasted by the event store using the code in Listing 13-10 in your `viewDidLoad` method.

**LISTING 13-10:** Add an observer in the `viewDidLoad` method

```
[ [NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(storeChanged:)
                                         name:EKEventStoreChangedNotification
                                         object:self.myEventStore];
```

Of course, you need to create the `storeChanged:` method and process the response in such a way that your application remains synchronized, such as refreshing your list of events and reloading the `UITableView`.

# WORKING WITH REMINDERS

Start Xcode and create a new project using the Single View Application Project template, and name it `MyReminders` using the options shown in Figure 13-5.

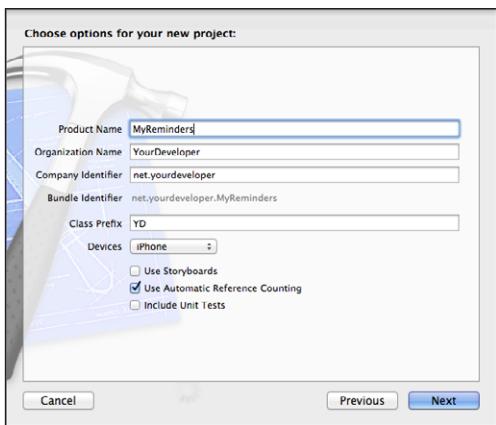


FIGURE 13-5

Add the EventKit, EventKitUI, and CoreLocation frameworks to your project.

The reason you are also adding the CoreLocation framework is because you'll learn how to create a geofenced reminder, for which the CoreLocation framework is required. Working with reminders is very similar to working with events, because they both use the Calendar database to store their information. This means that you can reuse a lot of the code you have already written, such as for requesting authorization to access the Calendar database and querying the event store.

**NOTE** A geo-fence is a virtual perimeter set around a GPS coordinate. Geofencing in combination with reminder allows you to be notified when you enter or leave the virtual perimeter. For example if you set a reminder with a geo-fence at your office GPS location you can be notified when you leave your office.

## Creating a Reminder

Create a reminder by creating and initializing an instance of the `EKReminder` object, named `reminder`, by calling the static `reminderWithEventStore:` method.

Set the different properties on your reminder instance, such as the title, notes, start date, due date, or completion date. To save the reminder, call the `saveReminder:commit:error:` method of the `myEventStore` object. A sample `addReminder:` implementation is shown in Listing 13-11.

### LISTING 13-11: A sample addReminder: method implementation

```
- (void)addReminder:(id)sender
{
    EKReminder *reminder = [EKReminder reminderWithEventStore:self.myEventStore];
    [reminder setTitle:@"Pick up 2 pizzas"];
    [reminder setNotes:@"Calzone's Pizza Cucina. 430 Columbus Ave, San
Francisco"];
}
```

```

reminder.calendar = [self.myEventStore defaultCalendarForNewReminders];
NSError *err;
[self.myEventStore saveReminder:reminder commit:YES error:&err];
if (err)
{
    //Handle errors here
}
[self loadReminders];
}

}

```

## Editing a Reminder

To edit an existing reminder you simply make changes to the properties of your `EKReminder` object, call the `saveReminder:commit:error:` method of the `myEventStore` object, and the changes are saved.

## Deleting a Reminder

Deleting a reminder is very similar to deleting an event. Pass the reminder instance you want to delete to the `removeReminder:commit:error:` method of the `myEventStore` instance.

## Working with Alarms

It's possible to add an alarm to a reminder by creating and initializing an instance of the `EKAlarm` object. You can use the `EKAlarm` object in several ways—you can set an absolute date or you can set an `EKStructuredLocation` object.

An `EKStructuredLocation` object is an object with three properties named `title`, `geoLocation`, and `radius`. The `EKStructuredLocation` represents your home address, work address, or the address of the grocery store.

If you set an `EKStructuredLocation` object for the `EKAlarm` instance, the alarm will fire if you either enter or leave the `EKStructuredLocation` related to the alarm.

This technology is known as *geofencing* and provides you with options to be notified by a reminder if you leave or enter the set location.

A practical implementation would be in an application where you set a reminder to pick up some pizzas before coming home from work. Once you leave your office, the reminder will notify you. The `addReminder:` method is shown in Listing 13-12.

### LISTING 13-12: Adding a reminder with geofencing

```

- (void)addReminder:(id)sender
{
    EKReminder *reminder = [EKReminder reminderWithEventStore:self.myEventStore];
    [reminder setTitle:@"Pick up 2 pizzas"];
    [reminder setNotes:@"Calzone's Pizza Cucina. 430 Columbus Ave, San
Francisco"];
}

```

*continues*

**LISTING 13-12 (continued)**

```

//create the geofence alarm
EKAlarm *enterAlarm = [[EKAlarm alloc] init];
[enterAlarm setProximity:EKAlarmProximityEnter];
EKStructuredLocation *enterLocation =
    [EKStructuredLocation locationWithTitle:@"Grocery store"];
CLLocationDegrees lat = 37.799052;
CLLocationDegrees lng = -122.408187;
CLLocation *shopLocation = [[CLLocation alloc]
    initWithLatitude:lat longitude:lng];
[enterLocation setGeoLocation:shopLocation];
//set the radius in meters
[enterLocation setRadius:200];
reminder.calendar = [self.myEventStore defaultCalendarForNewReminders];
[enterAlarm setStructuredLocation:enterLocation];
[reminder addAlarm:enterAlarm];
[reminder setCalendar:[self.myEventStore defaultCalendarForNewReminders]];

NSError *err;
[self.myEventStore saveReminder:reminder commit:YES error:&err];
if (err)
{
    //Handle errors here
}
}

```

When you create a reminder with an alarm it shows as in Figure 13-6. When you create a reminder with a geo-fence, it shows as in Figure 13-7.

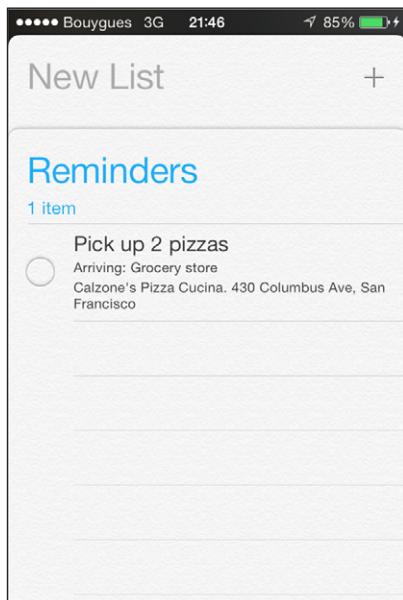


FIGURE 13-6

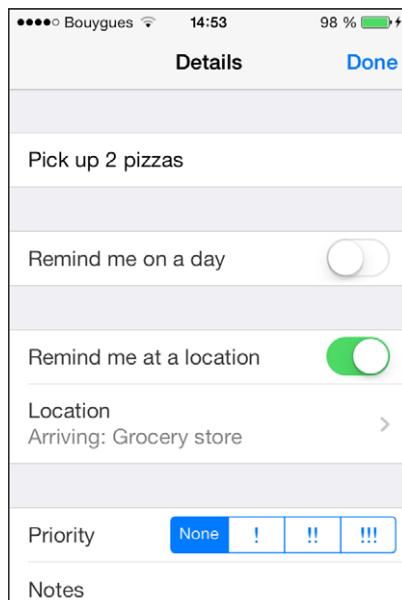


FIGURE 13-7

A sample implementation is given in the MyReminders project, which you can find in the download of this chapter.

## SUMMARY

In this chapter learned how to ask a user to authorize your application to access his Calendar database to interact with his events and reminders. You learned how to use the user interface elements that come with the EventKitUI framework to choose a calendar and create and edit events. You also learned how to create events and reminders programmatically in the Calendar database.

In the next chapter you learn how to integrate your application with social media platforms like Facebook and Twitter.



# 14

## Integrating with Social Media

### **WHAT'S IN THIS CHAPTER?**

---

- Using the Social and Accounts frameworks
- Connecting your application with social media
- Creating a single sign-in solution

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 14 download and individually named according to the names throughout the chapter.

### **INTRODUCTION TO SOCIAL MEDIA INTEGRATION**

Integration with social media has become standard in today's world. In iOS 5.0 Apple introduced the Twitter framework and the Accounts framework, which enable a developer to easily integrate his application with Twitter. Of course, other social networks exist, and therefore in iOS 6.0 Apple has decided to introduce the Social framework as a replacement for the Twitter framework, which has been deprecated.

The Social framework was developed with the knowledge that more social media platforms could be supported in a standardized way. Since iOS 6.0, the Social framework supports Twitter, Facebook, and Sina Weibo.

You can find the Accounts framework reference guide at [http://developer.apple.com/library/ios/#documentation/Accounts/Reference/AccountsFrameworkRef/\\_index.html%23//apple\\_ref/doc/uid/TP40011024](http://developer.apple.com/library/ios/#documentation/Accounts/Reference/AccountsFrameworkRef/_index.html%23//apple_ref/doc/uid/TP40011024).

The Social framework reference guide is available at [http://developer.apple.com/library/ios/#documentation/Social/Reference/Social\\_Framework/\\_index.html%23//apple\\_ref/doc/uid/TP40012233](http://developer.apple.com/library/ios/#documentation/Social/Reference/Social_Framework/_index.html%23//apple_ref/doc/uid/TP40012233).

The full documentation on the Facebook SDK for iOS is located at <https://developers.facebook.com/ios/>.

## UNDERSTANDING THE ACCOUNTS FRAMEWORK

With the introduction of iOS 5.0, the Accounts framework was added to the series of available frameworks. The Accounts framework provides you with easy access to accounts that are stored in the Accounts database. An account is capable of storing the login credentials of a particular service, such as Twitter.

When the user grants your application access to his Accounts database, you have access to the credentials required to authenticate the related service, such as logging in to the Twitter account. If you are developing an application that is using a particular service, you can use the Accounts framework to save the credentials to the Accounts database.

The Accounts framework consists of four classes:

- `ACAccount`
- `ACAccountCredential`
- `ACAccountStore`
- `ACAccountType`

The `ACAccountStore` class provides you with an interface for accessing the Accounts database. You can create, change, delete, and renew account credentials.

Start Xcode and create a new project using the Single View Application Project template, and name it `MyAccounts` using the options shown in Figure 14-1.

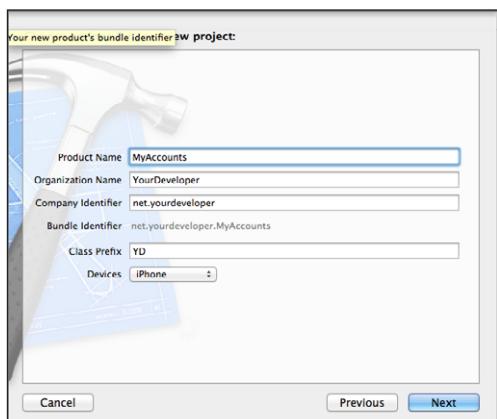


FIGURE 14-1

In this example you make a simple application that reveals all the accounts that are stored in the Accounts database and shows them in a UITableView.

Open the YDViewController.xib file using Interface Builder and the Assistant Editor and create a user interface with a UITableView, as shown in Figure 14-2.

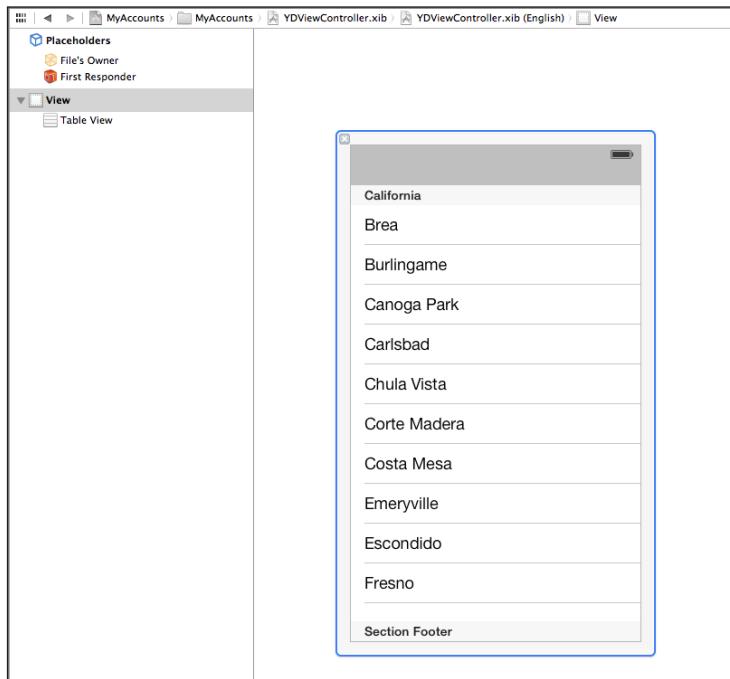


FIGURE 14-2

Add the Accounts framework to your project. Open the YDViewController.h file and import the Accounts header file. Create three strong properties as shown in Listing 14-1.

#### LISTING 14-1: Chapter14/MyAccounts/YDViewController.h

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

@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;

@property (nonatomic, strong) ACAccountStore* accountStore;
@property (nonatomic, strong) ACAccountType* twitterAccountType;
@property (nonatomic, strong) NSArray* twitterAccounts;
@end
```

Open the `YDViewController.m` file and subscribe to the `UITableViewDelegate` and `UITableViewDataSource` protocols.

In the `viewDidLoad` method, call the `requestAccessToAccountStore` method to prompt the user to authorize access to his Accounts database.

When an `ACAccountStore` is changed it broadcasts an `NSNotification` named `ACAccountStoreDidChangeNotification`. In your `viewDidLoad` method, set up an observer for the `ACAccountStoreDidChangeNotification` notification and implement the method to handle the notification. The notification will not contain any information on what has been changed in the `ACAccountStore`, so you need to request access again and reload your accounts because the change might be that the user revoked his authorization.

The `requestAccessToAccountStore` method starts by creating and initializing the `accountStore` instance. In this example you want to be authorized to access the Twitter accounts of the user, so initialize the `twitterAccountType` property by calling the `AccountTypeWithAccountTypeIdentifier` method of your `accountStore` instance.

Call the `requestAccessToAccountsWithType:options:completion:` method of your `accountStore` instance, which prompts the user to authorize your application. In the completion block you can call the `loadAccounts` method asynchronously on the `main_queue` if access has been granted, or present a `UIAlertView` with an error message if access has been denied.

The complete implementation is shown in Listing 14-2.

#### LISTING 14-2: `requestAccessToAccountStore` method

```
- (void)requestAccessToAccountStore
{
    self.accountStore = [[ACAccountStore alloc] init];
    self.twitterAccountType = [self.accountStore
        accountTypeWithAccountTypeIdentifier:
        ACAccountTypeIdentifierTwitter];
    NSDictionary* options = nil;
    [self.accountStore requestAccessToAccountsWithType:
        self.twitterAccountType options:options
        completion:^(BOOL granted, NSError *error)
    {
        if (granted == YES)
        {
            // do what you need to do here
            dispatch_async(dispatch_get_main_queue(), ^{
                //call loadAccounts here now you now the user has granted access
                [self loadAccounts];
            });
        }
        else
        {
            dispatch_async(dispatch_get_main_queue(), ^{

```

```
        [self showAuthorizationError];
    });
}
});
```

When you call the `requestAccessToAccountsWithType:options:completion:` method you need to pass an `NSDictionary` with options. When requesting access to Twitter accounts you can pass a nil value, because no options are required to access Twitter accounts. However, when you want to access accounts of type `ACAccountTypeIdentifierFacebook` to access Facebook accounts or `ACAccountTypeIdentifierSinaWeibo` to access Sina Weibo accounts, you need to pass the options `NSDictionary`. The `requestAccessToAccountsWithType:options:completion:` method will throw an `NSInvalidArgumentException` if the options dictionary is not provided for such account types.

You should configure the options dictionary with the following keys:

- `ACFacebookAppIdKey` containing a valid Facebook App ID.
  - `ACFacebookPermissionsKey`, which is an `NSArray` of permissions you are requesting.
  - `ACFacebookAudienceKey`, which is required only when posting permissions are requested. You can pass one of the following constants:
    - `ACFacebookAudienceEveryone`: Posts from your app are visible to everyone.
    - `ACFacebookAudienceFriends`: Posts are visible only to friends.
    - `ACFacebookAudienceOnlyMe`: Posts are visible to the user only.

A sample configuration is shown in Listing 14-3.

LISTING 14-3: Options configuration to access a Facebook account

```
NSDictionary *options = @{
    ACFacebookAppIdKey: @“012345678901234”,
    ACFacebookPermissionsKey: @[@"publish_stream", @"publish_actions"],
    ACFacebookAudienceKey: ACFacebookAudienceFriends };
```

Implement the table view delegate methods as shown in Listing 14-4.

#### LISTING 14-4 THE TABLE VIEW DELEGATE METHODS

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

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)sectionIndex
```

*continues*

**LISTING 14-4** (continued)

```

{
    return [self.twitterAccounts count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
    {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier];
    }
    //Get the currentAccount from the NSArray
    ACAccount* currentAccount = [self.twitterAccounts objectAtIndex:indexPath.row];

    //Display the accountDescription
    cell.textLabel.text = currentAccount.accountDescription;
    return cell;
}

```

## UNDERSTANDING THE SOCIAL FRAMEWORK

In iOS 6.0 Apple has introduced the Social framework as a replacement for the Twitter framework, which was introduced in iOS 5.0 and deprecated in iOS 6.0.

Although many developers still use the Twitter framework in their applications despite the fact that it's deprecated, you should use the Social framework instead.

In this section you learn how to create an application that is capable of posting and retrieving tweets from a Twitter account.

The Social framework contains two classes:

- ▶ SLComposeViewController
- ▶ SLRequest

The `SLComposeViewController` class presents a view to the user to compose a post for one of the supported social networks.

Start Xcode and create a new project using the Single View Application Project template, and name it `MyTwitter` using the options shown in Figure 14-3.

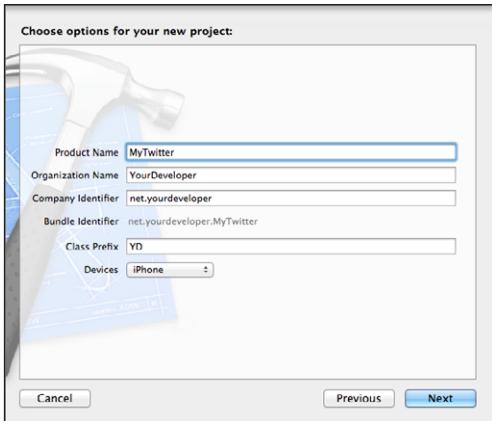


FIGURE 14-3

Open the `YDAppDelegate.h` file and create a strong property of type `UINavigationController`.

In the `YDAppDelegate.m` file, implement the `UINavigationController` to present the `YDViewController`.

Now add the Social framework to your project. Open the `YDViewController.h` file and import the Social header file.

## Making a Post

If you want to post to Twitter or any of the other supported social networks, you can use the `SLComposeViewController` class, which provides a user interface to create a post message, or you can use the `SLRequest` class, which provides a template for you to make the HTTP requests involved in making a post.

### Using the `SLComposeViewController`

Open the `YDViewController.m` file, and in the `viewDidLoad` method create and initialize a `UIBarButtonItem` that targets the `newTweet:` method.

Implement a `newTweet:` method that verifies whether the `SLComposeViewController` is available for the service type requested by calling the `isAvailableForServiceType:` method of the `SLComposeViewController` class.

Next, create a completion handler that will dismiss the presented `ViewController` and return an `SLComposeViewControllerResult` constant. This can be `SLComposeViewControllerResultCancelled` if the user canceled his post, or `SLComposeViewControllerResultDone` when the user taps Done.

Create and initialize an `SLComposeViewController` instance named `composeVC` by calling the static method `composeViewControllerForServiceType:`, which accepts one of the supported service types. The available service types are:

- `SLServiceTypeTwitter`
- `SLServiceTypeFacebook`
- `SLServiceTypeSinaWeibo`

On your `composeVC` instance you can call the following methods to preconfigure the post:

- `setInitialText::`: The text to appear in the post
- `addImage::`: An image to attach to the post
- `addURL::`: An `NSURL` to attach to the post
- `setCompletionHandler::`: To set your completion handler

Now call the `presentViewController:animated:completion:` method on the `navigationController` to present your preconfigured `composeVC` instance.

If the user has configured multiple Twitter accounts on his device, the `SLComposeViewController` class simply selects the first from the list as a default. However, when you tap behind the `From:` label on the name, all available accounts are presented, so you can choose from which account you want to make this post.

The complete implementation of the `YDViewController.m` file is shown in Listing 14-5.

#### LISTING 14-5: Chapter14/MyTwitter/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController : UIViewController

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    UIBarButtonItem *addEventButtonItem =
        [[UIBarButtonItem alloc] initWithBarButtonSystemItem:
            UIBarButtonSystemItemAdd target:self action:@selector(newTweet:)];
    self.navigationItem.rightBarButtonItem = addEventButtonItem;
}

- (void)newTweet:(id)sender
{
    if ([SLComposeViewController isAvailableForServiceType:SLServiceTypeTwitter]) {
```

```

//Start by setting up a completion handler
SLComposeViewControllerCompletionHandler __block
completionHandler:^(SLComposeViewControllerResult result){
    //dismiss the view controller
    [self.composeVC dismissViewControllerAnimated:YES completion:nil];
    switch(result){
        case SLComposeViewControllerResultCancelled:
        {
            //Cancel action
        }
        break;
        case SLComposeViewControllerResultDone:
        {
            UIAlertView *alertView = [[UIAlertView alloc]
                initWithTitle:@"Congratulations"
                message:@"Your message have been posted to Twitter."
                delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
            [alertView show];
        }
        break;
    }};
}

// Initialize Compose View Controller
self.composeVC = [SLComposeViewController
    composeViewControllerForServiceType:SLServiceTypeTwitter];
// Configure the compose View Controller
[self.composeVC setInitialText:@"A plane just flew over my head."];
UIImage* imageToPost = [UIImage imageNamed:@"plane.jpg"];
[self.composeVC addImage:imageToPost];
[self.composeVC addURL:[NSURL
    URLWithString:@"http://www.yourdeveloper.net/"]];
[self.composeVC setCompletionHandler:completionHandler];

[self.navigationController presentViewController:
    self.composeVC animated:YES completion:nil];

} else {
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Error"
    message:@"Can't connect to your Twitter account"
    delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
    [alertView show];
}
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

When you run the application on your device, it will look similar to Figure 14-4.

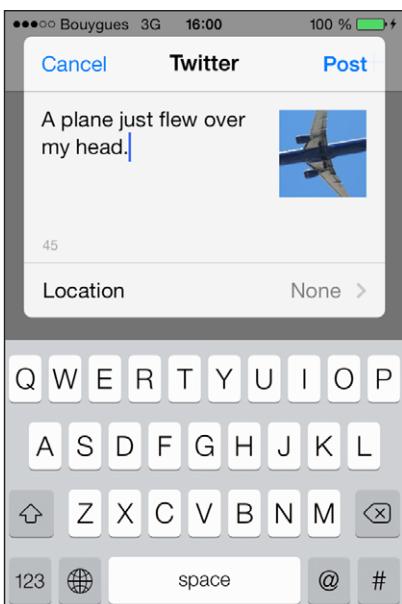


FIGURE 14-4

## Posting using the SLRequest Class

Using the `SLRequest` class in conjunction with the Accounts framework gives you the capability to construct HTTP-based requests and send them to the application programming interfaces (APIs) of social networks like Twitter, Facebook, and Sina Weibo and receive the responses.

Combining the `SLRequest` and the Accounts framework enables you to perform just about anything the API of the respective social network supports.

Start Xcode and create a new project using the Single View Application Project template, and name it `TwitterPostRequest` using the options shown in Figure 14-5.

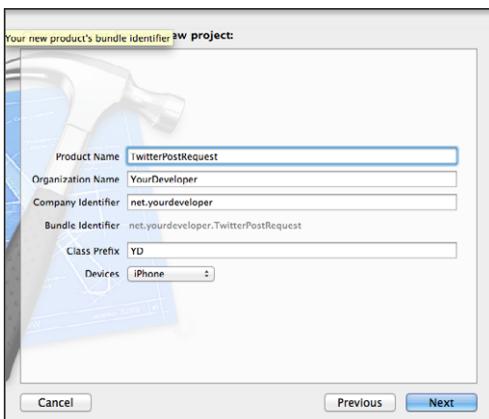


FIGURE 14-5

Open the `YDAppDelegate.h` file and create a strong property of type `UINavigationController`.

In the `YDAppDelegate.m` file, implement the `UINavigationController` to present the `YDViewController`.

Now add the Social framework and the Accounts framework to your project. Open the `YDViewController.h` file and import the Social and Accounts header files.

Like you did in the `MyAccounts` sample program, create strong properties of type `ACAccountStore` and `ACAccountType`, and an `NSArray` to hold the Twitter accounts.

In the `YDViewController.m` file, implement the `requestAccessToAccountStore` method as you did in the `MyAccounts` project. Also, you can copy the `loadAccounts` and the `showAuthorizationError` methods from the `MyAccounts` project.

Create a `UIBarButtonItem` in the `viewDidLoad` method targeting the `newTweet:` method, and add it to the navigationbar. Call the `requestAccessToAccountStore` method to prompt the user to authorize access to the Accounts database.

To post a message to Twitter using the `SLRequest` class, create the `newTweet:` method and start by implementing an `SLRequestHandler` block. Create an `ACAccount` instance named `currentAccount` by assigning the first object from the `twitterAccounts` array.

Create an `NSDictionary` named `message` containing the message you want to post. Next, create an `NSURL` object to the Twitter API.

Create an instance of `SLRequest` named `postRequest` by calling the `requestForServiceType:requestMethod:url:parameters:` method of the `SLRequest` class. Set the `currentAccount` to the account property of the `postRequest` instance and call the `performRequestWithHandler:` method with your `requestHandler`.

The complete implementation is shown in Listing 14-6.

#### LISTING 14-6: Chapter14/TwitterPostRequest/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController : UIViewController
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    [self requestAccessToAccountStore];
    UIBarButtonItem *addEventButtonItem =
        [[UIBarButtonItem alloc] initWithBarButtonSystemItem:
            UIBarButtonSystemItemAdd target:self action:@selector(newTweet:)];
    self.navigationItem.rightBarButtonItem = addEventButtonItem;
}


```

*continues*

**LISTING 14-6 (continued)**

```

- (void)newTweet:(id)sender
{
    //check if there are twitter accounts
    if ([self.twitterAccounts count] >0)
    {
        SLRequestHandler requestHandler =
        ^(NSData *responseData, NSHTTPURLResponse *urlResponse, NSError *error) {
            if (responseData) {
                NSInteger statusCode = urlResponse.statusCode;
                if (statusCode >= 200 && statusCode < 300) {
                    NSDictionary *postResponseData =
                    [NSJSONSerialization JSONObjectWithData:responseData
                        options:NSJSONReadingMutableContainers
                        error:NULL];
                    NSLog(@"[SUCCESS!] Created Tweet with ID: %@", postResponseData[@"id_str"]);
                }
                else {
                    NSLog(@"[ERROR] Server responded: status code %d %@", statusCode,
                        [NSHTTPURLResponse
                        localizedStringForStatusCode:statusCode]);
                }
            }
        else {
            NSLog(@"[ERROR] An error occurred while posting: %@", [error
                localizedDescription]);
        }
    };
}

ACAccount* currentAccount = [self.twitterAccounts objectAtIndex:0];
NSDictionary* message =
 @{@"status": @"My first post using the SLRequest class"};
NSURL* requestURL = [NSURL
    URLWithString:@"http://api.twitter.com/1/statuses/update.json"];

SLRequest* postRequest = [SLRequest
    requestForServiceType:SLServiceTypeTwitter
    requestMethod:SLRequestMethodPOST
    URL:requestURL parameters:message];
postRequest.account = currentAccount;
[postRequest performRequestWithHandler:requestHandler];
}

else
{
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Error"
        message:@"Configure a Twitter account" delegate:self
        cancelButtonTitle:@"OK" otherButtonTitles:nil, nil];
    [alert show];
}

}
- (void)requestAccessToAccountStore

```

```

{
    self.accountStore = [[ACAccountStore alloc] init];
    self.twitterAccountType =
        [self.accountStore accountTypeWithAccountTypeIdentifier:
            ACAccountTypeIdentifierTwitter];
    NSDictionary* options = nil;
    [self.accountStore requestAccessToAccountsWithType:
        self.twitterAccountType options:options
        completion:^(BOOL granted, NSError *error)
    {
        if (granted == YES)
        {
            // do what you need to do here
            dispatch_async(dispatch_get_main_queue(), ^{
                //call loadAccounts here now you now the user has granted access
                [self loadAccounts];
            });
        }
        else
        {
            dispatch_async(dispatch_get_main_queue(), ^{
                [self showAuthorizationError];
            });
        }
    }];
}

-(void)loadAccounts
{
    //check if we have been granted access to the twitter accounts
    if ([self.twitterAccountType accessGranted])
    {
        self.twitterAccounts = [[NSArray alloc]
            initWithArray:[self.accountStore
                accountsWithAccountType:self.twitterAccountType]];
    }
    else
    {
        [self showAuthorizationError];
    }
}

-(void)showAuthorizationError
{
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Error"
        message:@"You are not authorized" delegate:self
        cancelButtonTitle:@"OK" otherButtonTitles:nil, nil];
    [alert show];
}

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

```

*continues*

**LISTING 14-6** (continued)

```
// Dispose of any resources that can be recreated.
}

@end
```

When you launch the application, you are prompted to authorize access to the Accounts database. When you tap the + button, the newTweet: method is invoked and a post is made to your Twitter account.

If you want to post a tweet with an image, you need to make some small modifications to the newTweet: method because the Twitter API states the URL you are posting to is not `http://api.twitter.com/1/statuses/update.json`, but is `https://api.twitter.com/1.1/statuses/update_with_media.json`.

**WARNING** Please note that the URL's provided in the text are URL's to the Application Programming Interface of the social network. These URL's are not accessible by a normal browser since they need to be authenticated.

The addMultiPartData:withName:type:filename: method of the SLRequest class enables you to send the raw bytes of an image as an NSData object to the API.

To send a tweet with an image, implement the newTweet: method as shown in Listing 14-7.

**LISTING 14-7:** newTweet: method posting a tweet with an image

```
- (void)newTweet:(id)sender
{
    SLRequestHandler requestHandler =
    ^(NSData *responseData, NSHTTPURLResponse *urlResponse, NSError *error) {
        if (responseData) {
            NSInteger statusCode = urlResponse.statusCode;
            if (statusCode >= 200 && statusCode < 300) {
                NSDictionary *postresponseData =
                [NSJSONSerialization JSONObjectWithData:responseData
                    options:NSJSONReadingMutableContainers
                    error:NULL];
                NSLog(@"%@", [postresponseData objectForKey:@"id"]);
            } else {
                NSLog(@"%@", [urlResponse localizedDescription]);
            }
        } else {
            NSLog(@"%@", [error localizedDescription]);
        }
    };
}
```

```

};

ACAccount* currentAccount = [self.twitterAccounts objectAtIndex:0];

NSURL* url = [NSURL URLWithString:@"https://api.twitter.com"
    @"1.1/statuses/update_with_media.json"];
NSDictionary* params = @{@"status": @"Wow a plane"};
SLRequest* request = [SLRequest requestForServiceType:SLServiceTypeTwitter
    requestMethod:SLRequestMethodPOST
    URL:url
    parameters:params];
UIImage* plane = [UIImage imageNamed:@"plane.jpg"];
NSData *imageData = UIImageJPEGRepresentation(plane, 1.f);
[request addMultipartData:imageData
    withName:@"media[]"
    type:@"image/jpeg"
    filename:@"image.jpg"];
[request setAccount:currentAccount];
[request performRequestWithHandler:requestHandler];
}

```

**NOTE** Please find here a sample how to perform a post to Sina Weibo, which is very similar as posting to Twitter.

```

- (IBAction)saySomething:(id)sender
{
    if ([SLComposeViewController isAvailableForServiceType:
        SLServiceTypeSinaWeibo]) {
    {
        SLComposeViewController *composeViewController =
        [SLComposeViewController composeViewControllerForServiceType:
            SLServiceTypeSinaWeibo];
        [composeViewController setInitialText:@"现在专业的iOS编程。"];
        [composeViewController addURL:[NSURL URLWithString:
            @"http://www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-
            Programming.productCd-1118661133.html"]];
        __block SLComposeViewController *weakComposeVC =
            composeViewController;
        [composeViewController setCompletionHandler:^(SLComposeView-
            ControllerResult result)
        {
            self.result.text = SLComposeViewControllerResultDone ==
                result ? @"Sent" : @"Canceled";
            [weakComposeVC dismissViewControllerAnimated:YES
                completion:nil];
        }];
        [self presentViewController:composeViewController animated:YES
            completion:nil];
    }
}

```

## Retrieving Tweets

Now that you are capable of posting tweets to a Twitter account, it's time to retrieve the time line of a Twitter account and display it in a UITableView.

Open the YDViewController.xib file using Interface Builder and the Assistant Editor and implement a UITableView to display the retrieved tweets.

Open the YDViewController.h file and create a strong property of type NSMutableArray named tweets, which you'll be using to store the retrieved tweets.

In the YDViewController.m file, subscribe to the UITableViewDataSource and UITableViewDelegate protocols.

Create a method named loadTweets and start again by creating a completion handler. In this completion handler the response data, which is in JSON format, is serialized and assigned to the tweets NSArray. Create an NSMutableDictionary named parms to hold the parameters you need to assign to the SLRequest. Set a value of 10 for the key named count, because you only want to receive the last 10 tweets. Next, set the URL to [https://api.twitter.com/1.1/statuses/home\\_timeline.json](https://api.twitter.com/1.1/statuses/home_timeline.json), which is the Twitter API URL to retrieve the time line of a Twitter account. Finally, create an SLRequest instance named getRequest by calling the requestForServiceType:requestMethod :URL:parameters: method of the SLRequest class. Because you aren't posting data but retrieving data, set the requestMethod to SLRequestMethodGet. Finally, set the account to the current Account instance and call the performRequestWithHandler method of the SLRequest class.

Change the loadAccounts method by calling the loadTweets method after the twitterAccounts array has been initialized as shown in Listing 14-8, where the change is highlighted.

**LISTING 14-8:** The changed loadAccounts method

```
- (void)loadAccounts
{
    //check if we have been granted access to the twitter accounts
    if ([self.twitterAccountType accessGranted])
    {
        self.twitterAccounts = [[NSArray alloc]
            initWithArray:[self.accountStore
                accountsWithAccountType:self.twitterAccountType]];
        [self loadTweets];
    }
    else
    {
        [self showAuthorizationError];
    }
}
```

Finally, implement the required UITableView delegate methods as shown in Listing 14-9.

**LISTING 14-9:** The UITableView delegate methods

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

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)sectionIndex
{
    return [self.tweets count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
    {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                      reuseIdentifier:CellIdentifier];
    }
    //Get the Tweet from the NSArray
    NSDictionary* tweet = [self.tweets objectAtIndex:indexPath.row];
    cell.textLabel.text = tweet[@"text"];
    return cell;
}
```

## INTEGRATING WITH FACEBOOK

If you want your application to integrate with Facebook, you can use the Social framework as you saw earlier using the `SLRequest` class. Although a user might have authorized your application to access his Accounts database, the `requestAccessToAccountsWithType:options:completion:` method of the `ACAccountStore` class will throw an `NSInvalidArgumentException` if the options dictionary is not provided for such account types.

You should configure the options dictionary with the following keys:

- `ACFacebookAppIdKey` containing a valid Facebook App ID.
- `ACFacebookPermissionsKey`, which is an `NSArray` of permissions you are requesting.
- `ACFacebookAudienceKey`, which is required only when posting permissions are requested. You can pass one of the following constants:
  - `ACFacebookAudienceEveryone`: Posts from your app are visible to everyone.
  - `ACFacebookAudienceFriends`: Posts are visible only to friends.
  - `ACFacebookAudienceOnlyMe`: Posts are visible to the user only.

The first step to make your application integrate with Facebook is to set up and configure a Facebook application on <http://developers.facebook.com>. When you visit this page you see a menu item named Apps, and when you click it you see a button named +Create New App. Clicking this button brings up a popup screen in which you enter a name for your application; for example, YDChapter14 as shown in Figure 14-6.

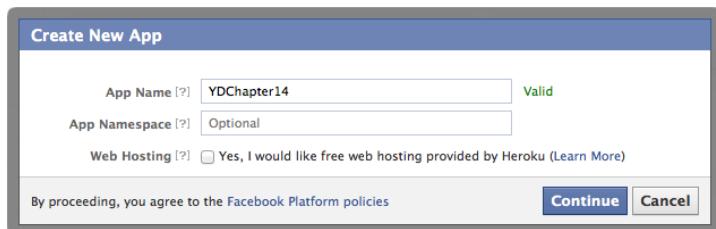


FIGURE 14-6

After passing a Captcha successfully, you see a page in which you can configure your application. Select the Native iOS App and enter the Bundle ID from your application; for example, `net.yourdeveloper.MyFacebook`. Enable the Facebook Login option and click Save Changes. The page will be posted, and at the top of the screen you will see the name of your application together with an App ID and an App Secret, as shown in Figure 14-7.

FIGURE 14-7

Keep this information safe and don't share it with anyone, otherwise someone might use your Facebook application to integrate with Facebook.

Start Xcode and create a new project using the Single View Application Project template, and name it MyFacebook using the options shown in Figure 14-8.

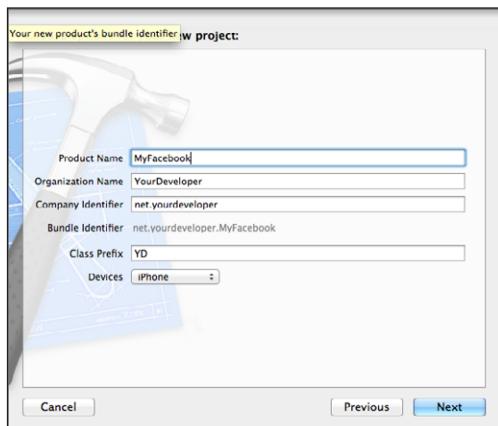


FIGURE 14-8

Add the Accounts and the Social frameworks to your project.

Open the YAppDelegate.h file and create a strong property of type UINavigationController named navController. Open the YAppDelegate.m file and change the application:didFinishWithLaunchingOptions: method to use the navController to become the rootViewController as you have done before.

Open the YDViewController.xib file using Interface Builder and the Assistant Editor and create a simple user interface with a UITableView as shown in Figure 14-9.

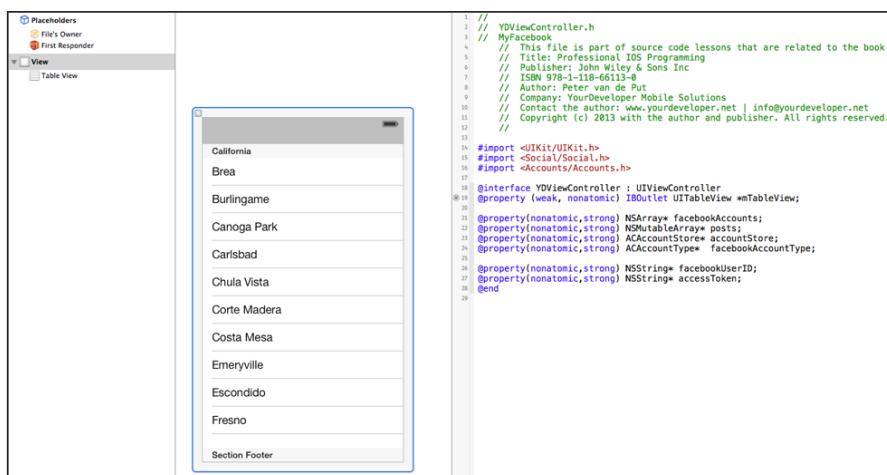


FIGURE 14-9

Open the `YDViewController.h` file and import the Social and Accounts header files. Implement the properties as shown in Listing 14-10.

**LISTING 14-10:** Chapter14/MyFacebook/YDViewController.h

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

@interface YDViewController : UIViewController
@property (weak, nonatomic) IBOutlet UITableView *mTableView;

@property (nonatomic, strong) NSArray* facebookAccounts;
@property (nonatomic, strong) NSMutableArray* posts;
@property (nonatomic, strong) ACAccountStore* accountStore;
@property (nonatomic, strong) ACAccountType* facebookAccountType;

@property (nonatomic, strong) NSString* facebookUserID;
@property (nonatomic, strong) NSString* accessToken;
@end
```

Open the `YDViewController.m` file and subscribe to the `UITableViewDelegate` and `UITableViewDataSource` protocols.

In the `viewDidLoad` method, call the `requestAccessToAccountStore` method and create a `UIBarButtonItem` targeting the `newPost:` method. The `requestToAccountStore` method starts by creating and initializing an instance of `ACAccountStore`. Next you need to create an `NSDictionary` named `options` in which you pass the Facebook-specific options, like the AppID that you received when you created the Facebook application.

**WARNING** *It's important to understand that when you are requesting access to Facebook, you have to ask for a read permission first and then a write permission, in case you want to make a post in a second request. Therefore, the `ACFacebookPermissionKey` is set to `email`, requesting access to the user's e-mail address of his Facebook account.*

When the user has granted access to the Accounts database, the `loadAccounts` method is called on the main queue.

The `loadAccounts` method starts by checking if access has been granted and loads the Facebook accounts in the `facebookAccounts` array, before it calls the `loadWallPosts` method. The `loadWallPosts` method starts by checking if there are Facebook accounts in the `facebookAccounts` array before it sets up an `SLRequestHandler`. In this `SLRequestHandler`, the response data received from the `SLRequest` is serialized in the `postDictionary` `NSDictionary`. The `posts` `NSMutableArray` is created and initialized and each `wallPost` `NSDictionary` is added to the `posts` array. Finally, the `SLRequestHandler` reloads the `mTableView` and calls the `requestAdditionalPermissions` method on the main queue.

The `currentAccount` is determined by selecting the first account from the `facebookAccounts` array. The `facebookUserID` property is set, and the access token is set on the `accessToken` property after retrieving it from the `credentials` of the `currentAccount`.

**WARNING** *The access token is required to pass in the SLRequest when you want to perform a post to Facebook.*

Next, construct the `SLRequest` to retrieve the posts from the user's Facebook wall using a `SLRequestGet` method for the URL `https://graph.facebook.com/me/posts`.

Set the `account` property of your `SLRequest` instance to the `currentAccount` and call the `performRequestWithHandler:` method of the `SLRequest` class, passing your `RequestHandler`.

In the `requestAdditionalPermission` method you again call the `requestAccessToAccountsWithType:` method of the `ACAccountStore` class, passing a different `NSDictionary` named `options`. This time you pass the `publish_stream` value to the `ACFacebookPermissionsKey`, which prompts the user to authorize the application to post to Facebook on behalf of the user.

In the `newPost:` method you again start by creating and initializing an `SLRequestHandler`. Create an `NSDictionary` named `fbPost` and initialize it with at least a `message` key and an `access_token` key. Create and initialize an `NSURL` instance named `requestURL` for the URL `https://graph.facebook.com/[USERID]/feed` where `[USERID]` needs to be replaced with the `facebookUserID` property.

Create and initialize an `SLRequest` instance named `postRequest` and pass the `fbPost` dictionary to the `parameters` property. Finally, call the `performRequestWithHandler:` method of the `SLRequest` class to invoke the request.

As a last step, implement the `UITableViewDelegate` methods, which will present the wallposts in the `UITableView`.

Listing 14-11 is showing only the following methods: `newPost:`, `requestAccessToAccountStore`, and `requestAdditionalPermissions`. The complete implementation of the `YDViewController.m` file can be found in the download of this Chapter.

---

**LISTING 14-11: Chapter14/MyFacebook/YDViewController.m**

---

```

- (void)newPost:(id)sender
{
    //check if there are facebook account
    if ([self.facebookAccounts count] >0)
    {
        SLRequestHandler requestHandler =
        ^(NSData *responseData, NSHTTPURLResponse *urlResponse, NSError *error) {
            if (responseData) {
                NSInteger statusCode = urlResponse.statusCode;
                if (statusCode >= 200 && statusCode < 300) {
                    //SUCCESS
                }
            }
        };
    }
}

```

*continues*

**LISTING 14-11** (continued)

```

        else {
            //Error
        }
    }
    else {
        //Error
    }
};

ACAccount* currentAccount = [self.facebookAccounts objectAtIndex:0];
NSDictionary* fbPost = @{@"access_token":self.accessToken, @"message":
    @"My first FBpost using the SLRequest class"};
NSURL* requestURL = [NSURL
    URLWithString:[NSString stringWithFormat:@"https://graph.facebook.com/%@/feed",self.facebookUserID]];

SLRequest* postRequest = [SLRequest
    requestForServiceType:SLServiceTypeFacebook
    requestMethod:SLRequestMethodPOST
    URL:requestURL parameters:fbPost];
postRequest.account = currentAccount;
[postRequest performRequestWithHandler:requestHandler];
}
else
{
    UIAlertView* alert = [[UIAlertView alloc]
        initWithTitle:@"Error"
        message:@"Configure a Facebook account"
        delegate:self cancelButtonTitle:@"OK"
        otherButtonTitles:nil, nil];
    [alert show];
}

-(void)requestAccessToAccountStore
{
    self.accountStore = [[ACAccountStore alloc] init];
    self.facebookAccountType = [self.accountStore
        accountTypeWithAccountTypeIdentifier:
        ACAccountTypeIdentifierFacebook];
    NSDictionary *options = @{
        ACFacebookAppIdKey: @"151575221693657",
        ACFacebookPermissionsKey: @[@"email"],
        ACFacebookAudienceKey: ACFacebookAudienceFriends
    };

    [self.accountStore requestAccessToAccountsWithType:
        self.facebookAccountType options:options
        completion:^(BOOL granted, NSError *error)
    {
        if (granted == YES)
        {
            // do what you need to do here
    }
}

```

```
dispatch_async(dispatch_get_main_queue(), ^{
    //call loadAccounts here now you now the user has granted access
    [self loadAccounts];
});
}
else
{
    dispatch_async(dispatch_get_main_queue(), ^{
        [self showAuthorizationError];
    });
}
]);
}

-(void)requestAdditionalPermissions
{
    self.accountStore = [[ACAccountStore alloc] init];
    self.facebookAccountType = [self.accountStore
        accountTypeWithAccountTypeIdentifier:
        ACAccountTypeIdentifierFacebook];
    NSDictionary *options = @{
        ACFacebookAppIdKey: @"151575221693657",
        ACFacebookPermissionsKey: @[@"publish_stream"],
        ACFacebookAudienceKey: ACFacebookAudienceFriends
    };

    [self.accountStore requestAccessToAccountsWithType:
        self.facebookAccountType options:options
        completion:^(BOOL granted, NSError *error)
    {
        if (granted == YES)
        {
            // do what you need to do here
            /*
            dispatch_async(dispatch_get_main_queue(), ^{
                //call loadAccounts here now you now the user has granted access
                [self loadAccounts];
            });
            */
        }
        else
        {
            dispatch_async(dispatch_get_main_queue(), ^{
                [self showAuthorizationError];
            });
        }
    }];
}

@end
```

**NOTE** It's important to realize the Facebook API is in no way related to iOS, and therefore it's important to check the <http://developers.facebook.com> website regularly when you are developing an iOS application that integrates with Facebook. It is beyond the scope of this book to completely explore all options of the Facebook API, but with the knowledge you've gained in this chapter you should be able to use all available functions of the Facebook API in your iOS applications.

## CREATING A SINGLE SIGN-IN APPLICATION

In many scenarios you need a user to identify himself before using your application for the first time. The most common scenario is when your application is integrated with a backend system and you need credentials of the user to create an account in your backend system using a web service. You can create a `UIViewController` with registration fields as you learned in Chapter 1, but many users consider this the old fashioned way. Because the majority of iPhone users are active on one or more social media platforms, you can assume they have a Twitter or Facebook account. With what you've learned in this chapter you are capable of providing a modern look and functionality to your application by asking the users to sign in with their Twitter or Facebook account by presenting a user interface similar to the one shown in Figure 14-10.

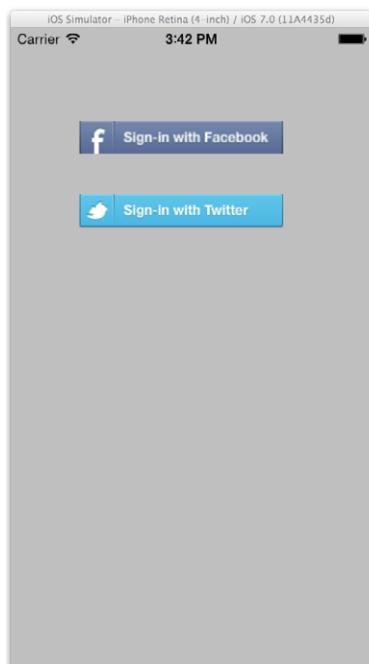


FIGURE 14-10

The images used in this sample are part of the download of this chapter and you are free to use them.

Start Xcode and create a new project using the Single View Application Project template, and name it sso using the options shown in Figure 14-11.

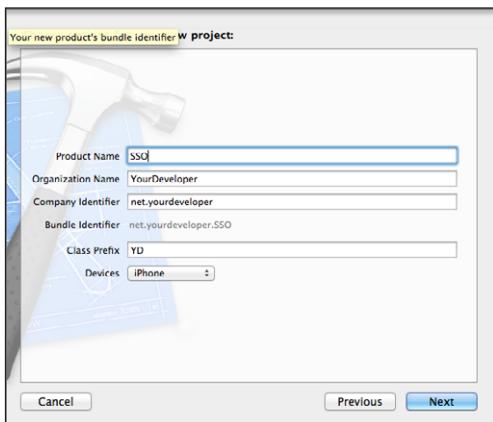


FIGURE 14-11

Import the Accounts framework to your project. Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor and create a simple user interface with two `UIButton` objects, as shown in Figure 14-12.

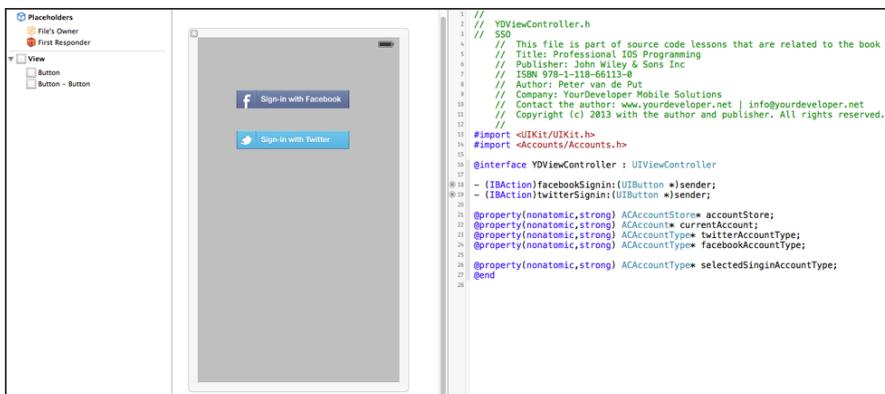


FIGURE 14-12

Open the `YDViewController.h` file and implement the properties as shown in Listing 14-12.

**LISTING 14-12:** Chapter14/SSO/YDViewController.h

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

@interface YDViewController : UIViewController

- (IBAction)facebookSignin:(UIButton *)sender;
- (IBAction)twitterSignin:(UIButton *)sender;

@property(nonatomic,strong) ACAccountStore* accountStore;
@property(nonatomic,strong) ACAccount* currentAccount;
@property(nonatomic,strong) ACAccountType* twitterAccountType;
@property(nonatomic,strong) ACAccountType* facebookAccountType;

@property(nonatomic,strong) ACAccountType* selectedSigninAccountType;
@end
```

Now open the `YDViewController.m` file and implement the `facebookSignin:` method. The `facebookSignin:` method uses the technique for requesting access to the Accounts database as you've done before. When the access has been granted, the `signedIn:` method is called.

The `twitterSignin:` method is comparable to the `facebookSignin:` method, where the difference is in the account type you want to have authorized.

In the `signedIn:` method you simply use the `username` property of the `currentAccount` instance and pass that to your backend system. If you require more information about a user, such as an e-mail address, you need to call additional information using the selected social media API with `SLRequest` objects like you did in the `MyFacebook` sample.

The complete implementation of the `YDViewController.m` file is shown in Listing 14-13.

**LISTING 14-13:** Chapter14/SSO/YDViewController.m

```
#import "YDViewController.h"

@interface YDViewController ()

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

- (IBAction)facebookSignin:(UIButton *)sender
{
```

```

self.accountStore= [[ACAccountStore alloc] init];
self.facebookAccountType = [self.accountStore
    accountTypeWithAccountTypeIdentifier:
    ACAccountTypeIdentifierFacebook];
NSDictionary *options = @{
    ACFacebookAppIdKey: @"+151575221693657",
    ACFacebookPermissionsKey: @[@"email"],
    ACFacebookAudienceKey: ACFacebookAudienceOnlyMe
};

[self.accountStore requestAccessToAccountsWithType:
    self.facebookAccountType options:options
    completion:^(BOOL granted, NSError *error)
{
if (granted == YES)
{
    // do what you need to do here
    dispatch_async(dispatch_get_main_queue(), ^{
        if ([self.facebookAccountType accessGranted])
        {
            NSArray* facebookAccounts = [[NSArray alloc]
                initWithArray:[self.accountStore
                    accountsWithAccountType:self.facebookAccountType]];
            self.currentAccount = [facebookAccounts objectAtIndex:0];
            [self signedIn];
        }
    else
    {
        [self showAuthorizationError];
    }
})
}
else
{
    dispatch_async(dispatch_get_main_queue(), ^{
        [self showAuthorizationError];
    });
}
}];

- (IBAction)twitterSignin:(UIButton *)sender
{
    self.accountStore = [[ACAccountStore alloc] init];
    self.twitterAccountType = [self.accountStore
        accountTypeWithAccountTypeIdentifier:
        ACAccountTypeIdentifierTwitter];
    [self.accountStore requestAccessToAccountsWithType:self.twitterAccountType
continues

```

**LISTING 14-13 (continued)**

```

        options:nil completion:^(BOOL granted, NSError *error)
    {
        if (granted == YES)
        {
            // do what you need to do here
            dispatch_async(dispatch_get_main_queue(), ^{
                if ([self.twitterAccountType accessGranted])
                {
                    NSArray* twitterAccounts = [[NSArray alloc]
                        initWithArray:[self.accountStore
                            accountsWithAccountType:self.twitterAccountType]];
                    self.currentAccount = [twitterAccounts objectAtIndex:0];
                    [self signedIn];
                }
                else
                {
                    [self showAuthorizationError];
                }
            });
        }
        else
        {
            dispatch_async(dispatch_get_main_queue(), ^{
                [self showAuthorizationError];
            });
        }
    }];
}

-(void)signedIn
{
    NSLog(@"%@", self.currentAccount.username);
    //you can use self.currentAccount.username to pass to your backend system
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Welcome"
        message:@"You have been signed-in" delegate:self
        cancelButtonTitle:@"OK" otherButtonTitles:nil, nil];
    [alert show];
}
-(void)showAuthorizationError
{
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Error"
        message:@"You are not authorized"
        delegate:self cancelButtonTitle:@"OK"
        otherButtonTitles:nil, nil];
    [alert show];
}
-(void)didReceiveMemoryWarning
{
}

```

```
[super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

If you plan to use this technology in your applications more than once, it's a good idea to add this view controller, with a proper name, to the Personal Library project you created in Chapter 1.

## SUMMARY

In this chapter you learned how to integrate your application with social media platforms like Twitter and Facebook. You are now capable of requesting access to the Accounts database and posting messages on Twitter and Facebook using the built-in `SLComposeViewController` programmatically. You also learned how to use the `SLRequest` class to communicate with the Twitter and Facebook APIs, which enables you to use all available functions of these social media platform APIs.

In Part IV of this book you learn how to take your application to production, analyze the usage of your application, and monetize it.



# PART IV

## Taking Your Application to Production

---

- ▶ **CHAPTER 15:** Analyzing Your Application
- ▶ **CHAPTER 16:** Monetize Your App
- ▶ **CHAPTER 17:** Understanding iTunes Connect
- ▶ **CHAPTER 18:** Building and Distribution



# 15

## Analyzing Your Application

### **WHAT'S IN THIS CHAPTER?**

---

- Analyzing your application from a technical perspective
- Analyzing your application from a commercial perspective

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 15 download and individually named according to the names throughout the chapter.

After all the hard work you've put into developing your iOS application, you want it to be a success. If you have developed the application for a paying customer, you probably are now in the phase where you can offer him a test version for evaluation purposes to see if your effort meets his expectations. However, if you have developed an application for yourself with the aim to *monetize* it, you also need to make sure the application is evaluated and tested before taking it into production. Chapter 16 is all about monetizing your application.

This chapter shows you how to perform a technical analysis before taking an application into production. It also shows you how to implement a commercial analysis as a process to be executed after the application has been taken into production.

### **PERFORMING A TECHNICAL ANALYSIS**

When your application has finally been realized and is available in the App Store, you need to get users to download it and start using it on their device. Studies have revealed that when users download and install an application on their smartphone and launch it, they want it to operate immediately and smoothly. When an application crashes during the first-time usage, in most cases users will remove the application from their device. In addition, when the application is using extensive storage space or memory, it is reason for users to terminate and remove the application.

All the money you might have spent in marketing your application to get a user to download and install it is lost. Even worse, you might receive a negative review.

The key technical reasons for applications to perform poorly or behave in a user-unfriendly manner are as follows:

- Blocking the main thread
- Memory leaks
- Using synchronized HTTP requests
- Extensive bandwidth usage
- Battery drainage
- Bad user interface

This section takes a closer look at these topics so you can learn how to identify them and, where possible, avoid them.

## Application Crashes

Application crashes are the number one reason why users stop using your application. Crashes can happen for many different reasons, and most fall directly under the responsibility of the programmer. In Chapter 1 you learned to create a crash handler, which I recommend using in all of your applications for two reasons:

- In case of an application crash, you can present a message to the user to inform him.
- You can collect the information from the crash log and use a web service to send it to you for analysis.

I use redmine ([www.redmine.org](http://www.redmine.org)) as my main management system. It is an open source platform to manage projects and tasks and integrates with a *subversion* (also known as *svn*) system for source code management. In my applications, I've added a method to the crash handler you saw in Chapter 1, and I send the collected crash log directly to redmine by calling a web service and creating a task of type *crash* in my system. If one of my applications crashes on a user's device, I'll be notified within a minute with the information of the device it has been running on, the version of the application, and the crash log. This gives me all the information I need to find the bug, fix it, and publish a new release of the application.

So far you've learned all the techniques and technologies to implement a similar process that fits in your organization.

## Blocking the Main Thread

A common mistake still made by many developers is blocking the main thread, which results in a non-responsive application. You should *never* perform heavy or long-running operations on the main thread of the application because UIKit does all of its work there. The easiest way to avoid this mistake is to move your operation to a background queue so it will not interfere with the main thread.

Create a `dispatch_queue_t` instance by calling the `dispatch_get_global_queue` function, which returns a global concurrent queue of a given priority level. Call the `dispatch_asynch` function to submit the block to the dispatch queue. A sample implementation is shown in Listing 15-1.

#### LISTING 15-1: Create a background queue

```
dispatch_queue_t bgQueue = dispatch_get_global_queue
    (DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0);
dispatch_async(bgQueue, ^{
//perform your operation
});
```

Instead of calling the `dispatch_get_global_queue` function, you can also create your own queue by calling the `dispatch_queue_create` function with a name for the queue. A sample implementation is shown in Listing 15-2.

#### LISTING 15-2: Create a named queue

```
dispatch_queue_t bgQueue = dispatch_queue_create("net.yourdeveloper.app.queueusername", nil);
dispatch_async(bgQueue, ^{
//perform your operation
});
```

## Memory Leaks

A memory leak occurs when the application incorrectly manages memory allocations. As a result, the object expected to be in memory can't be accessed.

The main cause of memory leaks is when you manage the memory yourself in your application, and your allocating and releasing is not balanced. This might result in an application crash when you try to access an object that has already been released.

The introduction of Automatic Reference Count (ARC) in iOS 5.0 has been a blessing for many programmers who were having a hard time in managing memory correctly in their applications. Automatic Reference Counting implements automatic memory management for Objective-C objects and blocks, freeing the programmer from the need to explicitly insert retains and releases. It does not provide a cycle collector; users must explicitly manage the lifetime of their objects, breaking cycles manually or with weak or unsafe references. When two objects keep a reference to each other and are retained, it creates a retain cycle since both objects try to retain each other, making it impossible to release.

ARC may be explicitly enabled with the compiler flag `-fobjc-arc`. It may also be explicitly disabled with the compiler flag `-fno-objc-arc`.

I was skeptical about ARC until I was working on a project I had to take over from another developer. The programming was poorly organized, and the application seemed to have random crashes. Because of the complex functionality of the application and the poor programming, it took a lot of time to find all the memory leaks. At that moment I decided to give ARC a try and redeveloped the

application. I was able to copy and paste most of the code and because of the use of ARC, all the release and retain statements had to be removed. When launching the application, it worked and the memory leaks were gone. Since then, I use ARC for all of my projects and still, today, it saves me a lot of time.

It is important to realize that ARC is a feature of the Objective-C compiler, and therefore the entire ARC logic of releasing and retaining happens when you build your app. ARC is not a runtime feature, except for one small part: the weak pointer system.

The only thing that ARC does is insert retains and releases into your code when it compiles it, exactly where you would have, or at least should have, put them yourself.

## Using Synchronized HTTP Requests

In Chapter 7 you learned how to make network calls to websites, RESTful services, and SOAP services. As you learned, two main objects are involved in calling an HTTP service: the `NSURLRequest` object and the `NSURLConnection` object.

The `NSURLRequest` object represents a URL load request independent of protocol and URL scheme. The main constructor accepts an `NSURL` object containing the URL to load, and optionally the cache policy for this request.

The `NSURLConnection` object is used to perform loading the `NSURLRequest`. The object can be invoked either synchronously or asynchronously, but the Apple developer guidelines very strongly discourage the use of synchronous connections because your application thread will block until the response is received. This means that the complete user interface is blocked; if the connection cannot be established, your application thread only comes back alive after the timeout period has expired, which can be up to 900 seconds.

Use the `URLRequest` class you created in Chapter 7 for your network communications to avoid the application having to wait for the response of the request and becoming unresponsive.

## Extensive Bandwidth Usage

The immense success of smartphones and tablets and the enormous amount of free applications has resulted in mobile operators changing their terms for mobile contracts.

Up until 2012, most mobile operators offered unlimited bandwidth with their prepaid and subscription plans. The explosive growth of the smartphone market consumed a lot of bandwidth of the operator networks, causing a negative effect on their earnings. As a result, most operators now offer only packages with a certain amount of megabytes (MB) in a certain timeframe, such as 500 MB/month. When users reach the agreed amount, they have to pay for each additional MB at a relatively high rate.

If your application uses a lot bandwidth, resulting in users having to pay for additional MBs, they won't be happy and might even blame your application.

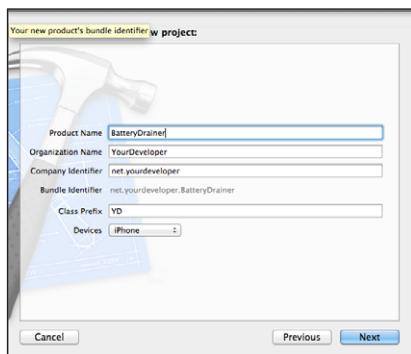
Especially when your application is communicating with a web service, you have to be careful to avoid reloading the complete dataset each time the user launches your application; you might consider storing the received remote information using Core Data. Another important improvement is to check if the user's device is connected via a WiFi connection or a cellular connection like 3G or 4G.

In most cases, a WiFi connection is free—such as at home, work, or school, where the cellular connection is always related to the user's operator plan.

Apple has developed a sample application named `Reachability` that demonstrates how to use the `SystemConfiguration` framework to monitor the network state of the device.

You can download the `Reachability` application directly from <http://developer.apple.com/library/ios/#samplecode/Reachability>. It contains a `Reachability.h` and a `Reachability.m` file that you are free to use in your own applications. The `Reachability` class is not ARC-compliant, which means that if you use this class in one of your ARC projects, you must set the `-fno-objc-arc` compiler flag for the `Reachability.m` file in your project.

To demonstrate, start Xcode and create a new project using the Single View Application Project template, and name it `BatteryDrainer` using the options shown in Figure 15-1.



**FIGURE 15-1**

Download the Apple `Reachability` sample project to obtain a copy of the `Reachability.h` and `Reachability.m` file. Once obtained, copy these two files to your project and add the `SystemConfiguration` and `CFNetwork` frameworks. Set the `-fno-objc-arc` compiler flag for the `Reachability.m` file in your project.

Open the `YAppDelegate.h` file and import the `Reachability` header file. Create a strong property of type `Reachability` named `reachability` and a public method named `connectedViaWIFI`, as shown in Listing 15-3.

#### LISTING 15-3: Chapter15/BatteryDrainer/YAppDelegate.h

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

#import "Reachability.h"
@class YDViewController;
```

*continues*

**LISTING 15-3 (continued)**

```

@interface YDAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@property (strong, nonatomic) YDViewController *viewController;

@property (strong, nonatomic) Reachability* reachability;

- (BOOL)connectedViaWIFI;
@end

```

Open the `YDAppDelegate.m` file and create and initialize the `reachability` instance. Call the `startNotifier` method of the `Reachability` class. Implement the `connectionViaWIFI` method by returning the `currentReachabilityStatus` equal to the `ReachableViaWiFi` constant of your `reachability` instance. The main part of the `YDAppDelegate.m` implementation is shown in Listing 15-4.

**LISTING 15-4: Chapter15/BatteryDrainer/YDAppDelegate.m**

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.reachability = [Reachability reachabilityForInternetConnection];
    [self.reachability startNotifier];
    [self.reachability currentReachabilityStatus];

    // Override point for customization after application launch.
    self.viewController = [[YDViewController alloc]
        initWithNibName:@"YDViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

- (BOOL)connectedViaWIFI
{
    return [self.reachability currentReachabilityStatus]== ReachableViaWiFi;
}

```

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor and create a simple user interface with a `UILabel` object to display the network connection status, as shown in Figure 15-2.

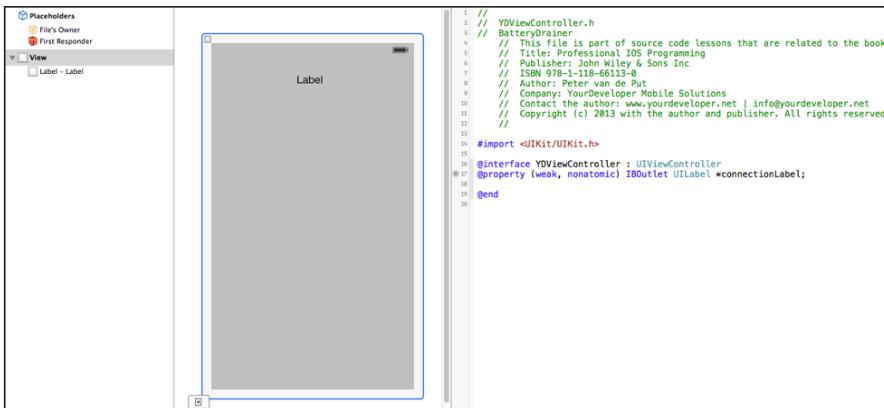


FIGURE 15-2

Open the `YDViewController.m` file and import the `YDAAppDelegate` header file. In the `viewDidLoad` method, create an instance of the `YDAAppDelegate` class named `appDelegate`. When the network status changes, it will broadcast a notification with the name `kReachabilityChangedNotification`. Create an observer for this notification targeting a method named `networkStatusChanged`.

To test the network status you can simply call the `connectedViaWIFI` method of your `appDelegate` instance and use the return value to implement your functional logic. The complete `YDViewController.m` file is shown in Listing 15-5.

#### LISTING 15-5: Chapter15/BatteryDrainer/YDViewController.m

```

#import "YDViewController.h"
#import "YDAAppDelegate.h"
@interface YDViewController : UIViewController

@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.

    YDAAppDelegate* appDelegate = (YDAAppDelegate *) [[UIApplication
        sharedApplication] delegate];

    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(networkStatusChanged:)
        name:kReachabilityChangedNotification
        object:appDelegate.reachability];
}

```

*continues*

**LISTING 15-5 (continued)**

```

if ([appDelegate connectedViaWIFI])
    self.connectionLabel.text = @"Connected via WIFI";
else
    self.connectionLabel.text = @"NOT Connected via WIFI";
}

- (void)networkStatusChanged:(NSNotification*)notification {
    //you know the network status has changed so perform your action here
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

An example of the way I use it in many applications using web services is when the user is connected via WiFi; I do a complete refresh of all the data by calling the web service and updating the data in Core Data. However, when the connection is not a WiFi connection, I call a different method on the web service API that will serve me only with changes since the last connection of this user. This saves a lot of bandwidth usage.

## Battery Drainage

As an iOS programmer you have to make sure your application doesn't drain the battery. One of the main causes affecting the battery lifetime is calling the `startUpdatingLocation` method of the `CLLocationManager` class and never calling the `stopUpdatingLocation` method.

When you download an application from the App Store, in many cases it will prompt you to authorize access to your location. This request is coming from the `CLLocationManager`, which is very often used for purposes it's not intended for.

Many developers create and initialize a `CLLocationManager` instance in their `AppDelegate` even if the application itself doesn't require it, so why do they do it?

As you learn in the “Performing a Commercial Analysis” section of this chapter, you might be interested in the location of a user to determine in which countries your application is used. With this information, you can localize your application in a certain language.

When you need this insight, simply create and initialize an instance of the `CLLocationManager` class, set the delegate, and call the `startUpdateLocation` method in the `application:didFinishWithLaunchingOptions:` method.

Implement the `locationManager:didUpdateToLocations:` delegate method and store the `[locations lastObject]` value in a `CLLocation` property, which you can use for your analytical purposes. Open the `BatteryDrainer` project and add the `CoreLocation` framework to the project. Open the `YAppDelegate.h` file and change it as shown in Listing 15-6.

**LISTING 15-6:** Chapter15/BatteryDrainer/YDAppDelegate.h

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

#import "Reachability.h"

#import <CoreLocation/CoreLocation.h>
@class YDViewController;

@interface YDAppDelegate : UIResponder <UIApplicationDelegate,CLLocationManagerDelegate>

@property (strong, nonatomic) UIWindow *window;

@property (strong, nonatomic) YDViewController *viewController;

@property(nonatomic,strong) CLLocationManager* locmanager;
@property(nonatomic,strong) CLLocation *userlocation;

@property (strong, nonatomic) Reachability* reachability;

- (BOOL)connectedViaWIFI;
@end
```

Now open the `YDAppDelegate.m` file and implement the `CLLocationManager` logic as explained. The implementation is shown in Listing 15-7.

**LISTING 15-7:** Chapter15/BatteryDrainer/YDAppDelegate.m

```
#import "YDAppDelegate.h"

#import "YDViewController.h"

@implementation YDAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    self.locmanager = [[CLLocationManager alloc] init];
    self.locmanager.delegate=self;
    [self.locmanager startUpdatingLocation];

    self.reachability = [Reachability reachabilityForInternetConnection];
    [self.reachability startNotifier];
    [self.reachability currentReachabilityStatus];

    // Override point for customization after application launch.
    self.viewController = [[YDViewController alloc]
                           initWithNibName:@"YDViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
```

*continues*

**LISTING 15-7 (continued)**

```
[self.window makeKeyAndVisible];
return YES;
}

- (BOOL)connectedViaWIFI
{
    return [self.reachability currentReachabilityStatus]== ReachableViaWiFi;
}

#pragma mark CoreLocation
- (void)locationManager:(CLLocationManager *)manager
    didUpdateLocations:(NSArray *)locations
{
    self.userlocation=[locations lastObject];
    [self.locmanager stopUpdatingLocation];
}
```

## Bad User Interface

Most iOS users expect an application to be intuitive and expressly designed for the device they are using, which might be an iPhone, iPod Touch, or an iPad. For this reason you should design your application for the device screen of the user. You can find an overview of the different screen sizes you have to take into account in Appendix B. Users also expect an application to respond to gestures that are familiar from other applications, such as swiping a table row to reveal a Delete button. Apple has provided a document named the iOS Human Interface Guidelines, which contains guidelines and principles that help you design a professional user interface.

You can find the iOS Human Interface Guidelines at <https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html>.

Finally, it's imperative that you use high-quality artwork to enrich your application. Most programmers are not graphic designers; so to make your application successful, consider working with a graphic designer to get an astonishing user interface.

## PERFORMING A COMMERCIAL ANALYSIS

Once your application is available in the App Store and you want to monetize it, commercial analysis is an important methodology to increase your revenues.

When you log in to the Apple Member Center via <https://developer.apple.com>, you have access to iTunes Connect, the area where you manage your applications. When you enter this part of the Member Center, you can click the Sales and Trends link to view and download your sales statistics.

The information provided is too limited to allow you to do a serious commercial analysis, however, because it basically only displays the number of downloads per market for each of your applications. You can also view the In-App Purchases per market, but the information is available only for the past 13 weeks and is discarded after that. You can download the data and process it in Excel, but you are still missing valuable information to perform a commercial analysis.

For this reason, several analytical platforms are available that enable you to perform some serious commercial analysis. It's up to you to select your analytics provider based on your specific requirements, but I always make use of the company Flurry ([www.flurry.com](http://www.flurry.com)) because it is one of the largest players in the field of gathering analytics data and it is free to use. It's estimated that about 50 percent of the iOS applications available in the App Store are using Flurry Analytics.

## Introducing Flurry Analytics

I want to emphasize that I'm in no way related to Flurry nor am I receiving any benefits by mentioning it in this book. It's not in the scope of this book to describe all the possible features of Flurry Analytics, so I'll restrict myself simply to explaining how to sign up for a Flurry account, download and install the Flurry Analytics SDK, and how to implement it. When you go to the Flurry website on [www.flurry.com](http://www.flurry.com) you can register yourself to create an account. Once you've verified your account, you can log in to your account and click Add New Application from the menu to create your application in the Flurry system. Follow the instructions on the screen to create and configure your application. Once your application is stored, you will see a screen telling you your unique application key and a link to download the latest SDK.

## Implementation of the SDK

Once you have downloaded the latest version of the Flurry SDK, navigate to the downloaded archive and within the archive navigate to the Flurry folder, which contains two files, the `Flurry.h` header file containing the public interface to a static library, and the library file named `libFlurry.a`.

Start Xcode and open the project in which you want to implement the Flurry Analytics SDK. Now drag and drop the `Flurry.h` and `libFlurry.a` files into your project.

Open your `YDAppDelegate.m` file and import the Flurry header file. In the `application:didFinishLaunchingWithOptions:` method, simply implement the following line:

```
[Flurry startSession:@"YOUR_API_KEY"] ;
```

replacing `YOUR_API_KEY` with your unique application key.

That's it. Each time your application is launched, statistics information is sent to Flurry, which you can then analyze on the website.

If you want to perform serious commercial analysis on your applications and you want to use Flurry, you should study the information available on its website.

## SUMMARY

In this chapter you learned how to perform technical and commercial analyses of your application. The technical analysis is important to ensure that users are happy and continue to use your application. The commercial analysis will help you find the information you need to improve your application, remove functions that are never used, or localize the application in multiple languages in order to monetize it.

In the next chapter you learn how to monetize your application—in other words, how to make money!

# 16

## Monetize Your App

### **WHAT'S IN THIS CHAPTER?**

---

- Understanding In-App Purchase
- Developing an In-App Purchase helper class
- Implementing advertisements in your application

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 16 download and individually named according to the names throughout the chapter.

### **INTRODUCTION TO MONETIZING**

After all the time you've put into building and testing your application, you may want to monetize your efforts. You can do this in a number of ways, including:

- Paid application
- Advertising
- In-App Purchases
- Subscriptions
- Lead generation
- Affiliate sales

## Paid Application

Publishing your application as a paid application in the App Store is simple—you set your price tier and start marketing your application so users will buy it. Keep in mind that Apple will take 30 percent of your revenue, and customers who've bought your application might expect free updates and support for as long as they use the application.

The price you can set for your application depends strongly on the functionality it offers as well as the price that is set for similar or competitive applications.

## Advertising

You can work with an advertising platform like Apple's iAd or Google's AdMob to generate revenue by displaying advertisements in your application. You learn how to implement these platforms in the section "Monetizing with Advertisements." Most advertising platforms don't pay for displaying advertisements, but you will be paid if a user taps on the advertisement and is redirected to the advertisement details.

You can also sell space to companies that might find advertising in your application beneficial.

Hundreds of advertising platforms are available, and it's up to you to decide which one works best—it might be different for each application.

## In-App Purchases

An In-App Purchase is a way to up-sell functionality of your application. Your application might be free, but some functionality will be unlocked when a user upgrades to a pro version using an In-App Purchase.

Various kinds of In-App Purchases are available and are explained in this chapter, including how you can implement them. As with paid applications, Apple also takes a 30 percent cut of your revenue.

**NOTE** *In-App Purchase is the only mechanism by which an application can allow users to purchase content that is consumed within the application. Thus an application can't use a payment service like PayPal as an alternative to In-App Purchase if the item being purchased is consumed on the device.*

## Subscriptions

Subscription-based applications include, for example, magazine applications for which a user needs to purchase a subscription to receive the new issue or content. For digital magazine applications, Apple introduced Newsstand in iOS 5.0. You can find more information on Newsstand at <https://developer.apple.com/newsstand>.

## Lead Generation

A nice example of lead generation is one of my own applications named FX, which is a currency converter used by thousands of people in more than 60 countries. You can download it from <https://itunes.apple.com/in/app/fx-converter/id479159482?mt=8>. The commercial analysis using the Flurry Analytics framework (explained in Chapter 15) revealed that a large number of users are living in the United Kingdom, and with additional market research revealed that quite a few of them are looking to buy a property in France to spend their retirement. Once they purchase a property in France, or any other country where they don't use the £ as a currency, they need to convert a serious amount of Pound Sterling to Euros.

Using the FX application, a user can request a quotation for a currency conversion. That information is shared with several professional currency conversion companies, which will pay a certain fee if that user becomes a customer.

## Affiliate Sales

You can generate sales for products or services by subscribing to an affiliate program like Apple's iTunes. You can find more information about the affiliate program at <http://www.apple.com/itunes/affiliates/resources/documentation/app-store-affiliate-program.html>.

# DEVELOPING IN-APP PURCHASES

When you want to implement In-App Purchase to your application, it's important to understand the In-App Purchase process, the different types of product types and their specific behaviors and requirements as well as how to verify the payment and supply the purchased content to the user.

## Introduction to In-App Purchase

Store Kit is the framework that provides you with classes that allow an application payment from a user to purchase additional content. You can find the Store Kit framework reference guide at [https://developer.apple.com/library/ios/#documentation/StoreKit/Reference/StoreKit\\_Collection/\\_index.html](https://developer.apple.com/library/ios/#documentation/StoreKit/Reference/StoreKit_Collection/_index.html).

The In-App Purchase programming guide is located at <https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/Introduction/Introduction.html>.

The Store Kit framework communicates with the App Store on behalf of your application and provides localized information about the products you want to offer. You present these products to users, enabling them purchase one or more of them. Once a user makes a selection, your application uses Store Kit to collect payment from that user.

Basically, you can sell four different kinds of products:

- **Subscriptions:** For example, a weekly analysis report presented in your application.
- **Services:** For example, a translation of a document.
- **Functionality:** For example, a pro version that unlocks functionality not available in the free version or that stops displaying advertisements.
- **Content:** For example, a game level, photos, artwork, and other digital content.

## Registering Products

Before you can sell a product using In-App Purchases, you need to register it with the App Store through iTunes Connect, which is available via the member center of the developer portal.

To register a product you need to enter a name, a description, and pricing information, as well as some other metadata that will be used by the App Store.

A product is uniquely identified by a *product identifier*, for which it's common practice to use a reverse DNS name like net.yourdeveloper with your bundle identifier. For example, the product identifier for the "Upgrade to Pro" product might be net.yourdeveloper.appname.upgradeToPro.

## Choosing the Product Type

You can select from different product types, each with its own specific characteristics. The following product types and the explanations are taken from the developer portal.

### Consumable

A consumable In-App Purchase must be purchased each time the user needs it. As the name implies, it's consumed once and then it is no longer available.

### Non-Consumable

Non-consumable In-App Purchases must be purchased only once by users. After a non-consumable product is purchased, it is provided to all devices associated with that user's iTunes account. Store Kit provides built-in support to restore non-consumable products on multiple devices.

### Auto-Renewable Subscriptions

Auto-renewable subscriptions are delivered to all of a user's devices in the same way as non-consumable products. However, auto-renewable subscriptions differ in other ways. When you create an auto-renewable subscription in iTunes Connect, you choose the duration of the subscription. The App Store automatically renews the subscription each time its term expires. If the user chooses to not allow the subscription to be renewed, the user's access to it is revoked after the subscription expires. Your application is responsible for validating whether a subscription is currently active and can also receive an updated receipt for the most recent transaction.

## Free Subscription

Free subscriptions are a way for you to put free subscription content in Newsstand. Once a user signs up for a free subscription, the content is available on all devices associated with the user's Apple ID. Free subscriptions do not expire and can be offered only in Newsstand-enabled apps.

## Non-Renewing Subscription

A non-renewing subscription is a mechanism for creating products with a limited duration. Non-renewing subscriptions differ from auto-renewable subscriptions in a few key ways:

- The term of the subscription is not declared when you create the product in iTunes Connect; your application is responsible for providing this information to the user. In most cases, you would include the term of the subscription in the description of your product.
- Non-renewing subscriptions can be purchased multiple times (like a consumable product) and are not automatically renewed by the App Store. You are responsible for implementing the renewal process inside your application. Specifically, your application must recognize when the subscription has expired and prompt the user to purchase the product again.
- You are required to deliver non-renewing subscriptions to all devices owned by the user. Non-renewing subscriptions are not automatically synchronized to all devices by Store Kit; you must implement this infrastructure yourself. For example, most subscriptions are provided by an external server; your server would need to implement a mechanism to identify users and associate subscription purchases with the user who purchased them.

## Understanding the In-App Purchase Process

The Store Kit framework does not cover all the steps in an In-App Purchase. Its main functionality is to interact with the App Store.

The In-App Purchase process requires you to implement the following steps:

1. Retrieve a list of product identifiers that are available from the application bundle.
2. Send a request to the App Store using the Store Kit framework to retrieve detailed information on each product identifier, such as name, description, price, and product type.
3. The App Store returns the requested information to the application.
4. The application presents a user interface with the available products like a normal e-commerce store, where the user can select one or more products to purchase.
5. Once the user initiates the actual purchase, the application sends a payment request to the App Store using the Store Kit framework.
6. The App Store processes the payment and returns the transaction information to the application.
7. The application processes the received payment information and, if payment has been successful, it delivers the product to the application.

With this process your product identifiers are part of the application bundle. Although this is easy to implement, it's difficult to maintain because if you want to add new products available for your application, you need to add those new products to your application bundle and upload a new version to the App Store. This will then go through the normal review process.

As an alternative to having to upload a new version of your application to be reviewed and released, Apple advises to store your list of product identifiers on a server and in the first step of the In-App Purchase process, you contact your server to obtain the list of product identifiers. In this scenario you can add new products without having to upload a new version of your application to be reviewed and released.

## Implementing an In-App Purchase

In this section you learn how to implement a complete In-App Purchase process for an application that has several product identifiers available from the application bundle.

Start Xcode and create a new project using the Single View Application Project template, and name it `MyStore` using the options shown in Figure 16-1.

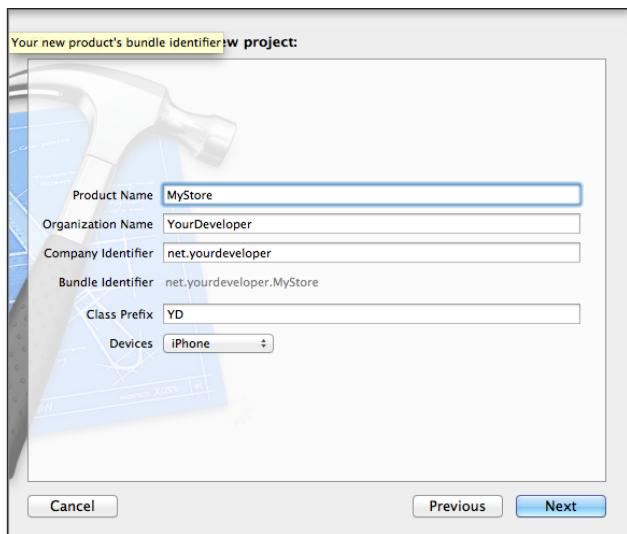


FIGURE 16-1

Add the Store Kit framework to your project. In this application you'll be presenting a store with several products.

Let's assume that your application has an option to purchase a pro version that doesn't present any advertisements, which the standard application does. Later in this chapter, in the section "Monetizing with Advertisements," you learn how to implement the advertisements, but for now you'll just display a surrogate image as an advertisement.

Open the `YDViewController.xib` file using Interface Builder and the Assistant Editor and create a user interface with a `UITableView` as shown in Figure 16-2.

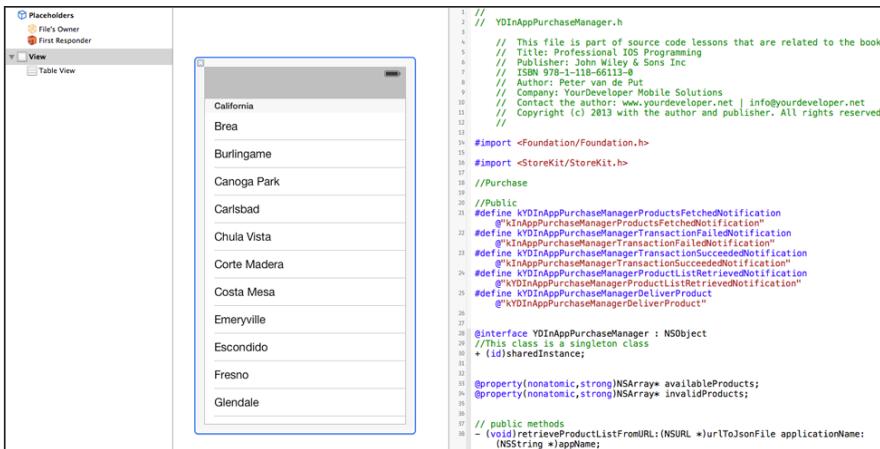


FIGURE 16-2

Notice that the `UITableView` has a `Y` position of 50 so you can display the `surrogateAdvertisementView` above it. This `surrogateAdvertisementView` will later be replaced with a real advertisement.

## Create Products in iTunes Connect

You have to start by creating your products in iTunes Connect. The [http://developer.apple.com/library/mac/#documentation/LanguagesUtilities/Conceptual/iTunesConnect\\_Guide/13\\_ManagingIn-AppPurchases/ManagingIn-AppPurchases.html](http://developer.apple.com/library/mac/#documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/13_ManagingIn-AppPurchases/ManagingIn-AppPurchases.html) link will bring you to the iTunes Connect Developer Guide, which explains in detail how to create and configure your products.

For this demonstration project I've configured three products in iTunes Connect, as shown in Table 16-1.

TABLE 16-1: Sample products in iTunes connect.

PRODUCT IDENTIFIER	PRODUCT TYPE
net.yourdeveloper.MyStore.cons1	Consumable
net.yourdeveloper.MyStore.cons2	Consumable
net.yourdeveloper.MyStore.upgrade	Non-Consumable

To make life easier, create an In-App Purchase helper class named `YDInAppPurchaseManager`, which you can add to the Personal Library from Chapter 1 and reuse in all your applications for which you want to use In-App Purchase functionality.

Using Xcode, select File  $\Rightarrow$  New  $\Rightarrow$  File from the menu, select the Objective-C class template, and click Next. In the Options dialog box, name the class `YDInAppPurchaseManager` and subclass it from `NSObject` as shown in Figure 16-3. Click Next.

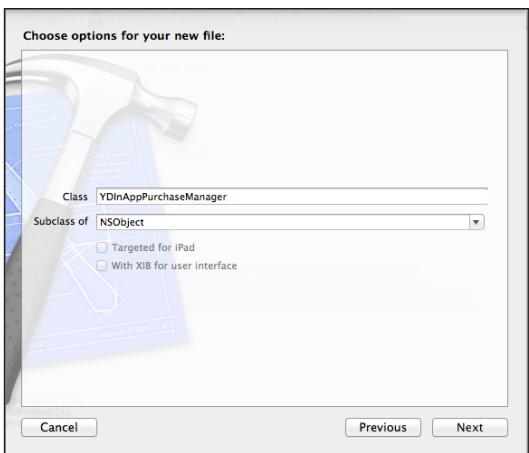


FIGURE 16-3

In the following screen, click Create.

The `YDInAppPurchaseManager` class will be designed as a singleton class. Singleton classes are an important concept to understand because they exhibit an extremely useful design pattern. This idea is used throughout the iPhone SDK; for example, `UIApplication` has a method called `sharedApplication` that, when called from anywhere, returns the `UIApplication` instance that relates to the currently running application.

In the `YDInAppPurchaseManager.h` file, create the `sharedInstance` method like this:

```
+ (id)sharedInstance;
```

Implement this method in the `YDInAppPurchaseManager.m` file. This method starts by creating a static variable named `sharedManager`, which is then initialized only once. The only-once creation is realized by using the `dispatch_once` function from Grand Central Dispatch, as shown in Listing 16-1.

#### LISTING 16-1: The `sharedInstance` method implementation

```
+ (id)sharedInstance {
    static YDInAppPurchaseManager *sharedManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedManager = [[self alloc] init];
    });
    return sharedManager;
}
```

You can now call the methods of the `YDInAppPurchaseManager` by calling `[ [YDInAppPurchaseManager sharedInstance] METHOD_NAME ]`, where you replace `METHOD_NAME` with one of the implemented methods.

Open the `YDInAppPurchaseManager`, import the `StoreKit` header file, and define some constants, which will be used for setting up notifications.

Create two strong `NSArray` properties: one named `availableProducts`, which will contain the available products sellable via this application; and one named `invalidProducts`, which will contain products that are not configured properly or aren't available in the App Store.

Next, create the definitions for the public methods. The complete `YDInAppPurchaseManager.h` file is shown in Listing 16-2.

#### LISTING 16-2: Chapter16/MyStore/YDInAppPurchaseManager.h

```
#import <Foundation/Foundation.h>
#import <StoreKit/StoreKit.h>

//Purchase

//Public
#define kYDInAppPurchaseManagerProductsFetchedNotification
    @"kInAppPurchaseManagerProductsFetchedNotification"
#define kYDInAppPurchaseManagerTransactionFailedNotification
    @"kInAppPurchaseManagerTransactionFailedNotification"
#define kYDInAppPurchaseManagerTransactionSucceededNotification
    @"kInAppPurchaseManagerTransactionSucceededNotification"
#define kYDInAppPurchaseManagerProductListRetrievedNotification
    @"kYDInAppPurchaseManagerProductListRetrievedNotification"
#define kYDInAppPurchaseManagerDeliverProduct
    @"kYDInAppPurchaseManagerDeliverProduct"

@interface YDInAppPurchaseManager : NSObject
//This class is a singleton class
+ (id)sharedInstance;

@property(nonatomic,strong)NSArray* availableProducts;
@property(nonatomic,strong)NSArray* invalidProducts;

// public methods
- (void)retrieveProductListFromURL:(NSURL *)urlToJsonFile
    applicationName:(NSString *)appName;
- (BOOL)productPurchased:(NSString *)productIdentifier;
- (BOOL)canMakePurchases;
- (void)purchaseProduct:(SKProduct*)productToPurchase;
@end
```

Add the `URLRequest` class you created in Chapter 7 to your project.

For the validation of the transaction receipt, you need the ability to create a base64-encoded string from an `NSData` object.

In Xcode, select File  $\Rightarrow$  New  $\Rightarrow$  File from the menu and select the Objective-C category as shown in Figure 16-4. Click Next.

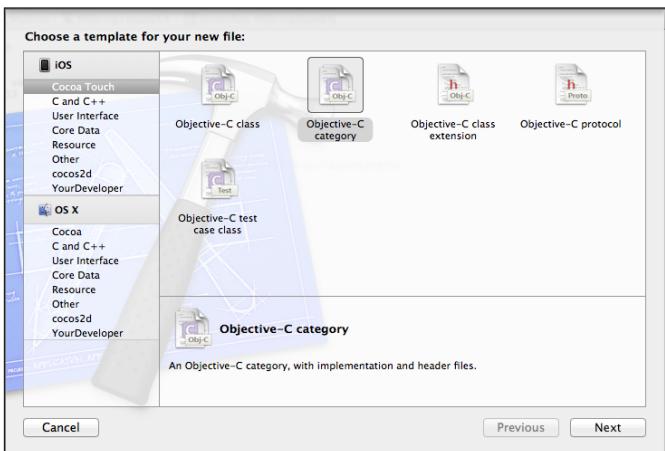


FIGURE 16-4

On the next screen, name the category `base64` and select `NSString` from the Category On dropdown list as shown in Figure 16-5.

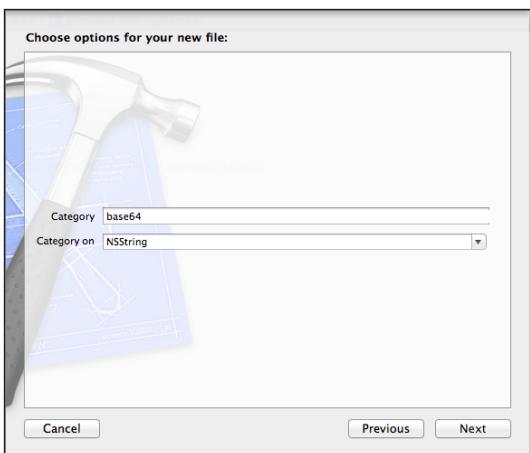


FIGURE 16-5

Click Next and create the file in your project folder.

Open the `NSString+base64.h` file and implement the code as shown in Listing 16-3.

**LISTING 16-3: Chapter16/MyStore/NSString+base64.h**

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

@interface NSString (base64)
+ (NSString *) base64StringFromData:(NSData *)paramData length:(NSInteger)paramLength;
@end
```

Open the `NSString+base64.m` file and implement the code as shown in Listing 16-4.

**LISTING 16-4: Chapter16/MyStore/NSString+base64.m**

```
#import "NSString+base64.h"

@implementation NSString (base64)
static char Base64EncodingTable[64] = {
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
    'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
    'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
    'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/'
};

+ (NSString *) base64StringFromData:(NSData *)paramData length:(NSInteger)paramLength {

    NSInteger ixtext = 0, dataLength = 0;
    long remainingCharacters;
    unsigned char inputCharacters[3] = {0, 0, 0}, outputCharacters[4] = {0, 0, 0, 0};
    short counter, charsonline = 0, charactersToCopy = 0;
    const unsigned char *rawBytes;

    NSMutableString *result;

    dataLength = [paramData length];
    if (dataLength < 1){
        return [NSString string];
    }
    result = [NSMutableString stringWithCapacity: dataLength];
    rawBytes = [paramData bytes];
    ixtext = 0;

    while (YES) {
        remainingCharacters = dataLength - ixtext;
        if (remainingCharacters <= 0)
            break;
        for (counter = 0; counter < 3; counter++) {
            NSInteger index = ixtext + counter;
            if (index < dataLength)
                inputCharacters[counter] = rawBytes[index];
            else
                inputCharacters[counter] = 0;
        }
        outputCharacters[0] = (inputCharacters[0] & 0xFC) >> 2;
```

*continues*

**LISTING 16-4** (continued)

```

outputCharacters[1] =
    ((inputCharacters[0] & 0x03) << 4) | ((inputCharacters[1] & 0xF0) >> 4);
outputCharacters[2] =
    ((inputCharacters[1] & 0x0F) << 2) | ((inputCharacters[2] & 0xC0) >> 6);
outputCharacters[3] = inputCharacters[2] & 0x3F;
charactersToCopy = 4;
switch (remainingCharacters) {
    case 1:
        charactersToCopy = 2;
        break;
    case 2:
        charactersToCopy = 3;
        break;
}
for (counter = 0; counter < charactersToCopy; counter++) {
    [result appendString: [NSString stringWithFormat:
        @"%c", Base64EncodingTable[outputCharacters[counter]]]];
}
for (counter = charactersToCopy; counter < 4; counter++) {
    [result appendString: @"="];
}
ixtext += 3;
charsonline += 4;

if ((paramLength > 0) && (charsonline >= paramLength)) {
    charsonline = 0;
}
}
return result;
}
@end

```

Open the `YDInAppPurchaseManager.m` file and import the `NSURLRequest` and the `NSString+base64` header files.

To validate the transaction receipts, you need to access a URL on the `apple.com` domain. There are different URLs for the production and the sandbox environments. For easy access, just create them here as constants as shown in Listing 16-5.

**LISTING 16-5:** Creating the constants

```

const NSString* kSKTransactionReceiptVerifierURL =
    @"https://buy.itunes.apple.com/verifyReceipt";
const NSString* kSKSandboxTransctionReceiptVerifierURL =
    @"https://sandbox.itunes.apple.com/verifyReceipt";

```

Subscribe to the `SKProductRequestDelegate` and the `SKPaymentTransactionObserver` protocols. Create a private variable named `productRequest` of type `SKProductRequest` and one named `productList` of type `NSMutableArray` in the private interface declaration, as shown in Listing 16-6.

#### LISTING 16-6: Private interface declaration

```
@interface YDInAppPurchaseManager() <SKProductsRequestDelegate,
                                         SKPaymentTransactionObserver>
{
    SKProductsRequest *productsRequest;
    NSMutableArray* productList;
}

@end
```

Create the `sharedInstance` method as explained earlier and shown in Listing 16-7.

#### LISTING 16-7: sharedInstance method

```
+ (id)sharedInstance {
    static YDInAppPurchaseManager *sharedManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedManager = [[self alloc] init];
    });
    return sharedManager;
}
```

## Retrieving the Product List

As you have seen before you can retrieve the product list in two ways. You can create a list of product identifiers in your application bundle or you can store the list on a server.

When integrating the list in the bundle, you need to build and upload a new version of your application once the product list changes; however, when you store it on a server you are free to change the product list whenever you want. Because it doesn't affect the application, there is no need to build a new version and submit it to the App Store.

You can develop a complete backend system in which you manage your application and its In-App Purchase product lists, or you can use a text editor and create a JSON file with a structure similar to that shown in Listing 16-8.

#### LISTING 16-8: Sample JSON file

```
{
    "applications": {
        "application": [
            {
                "id": "com.yourcompany.yourapp"
            }
        ]
    }
}
```

*continues*

**LISTING 16-8 (continued)**

```

    "name": "MyStore",
    "products": [
        "product": [
            {
                "product_identifier": "net.yourdeveloper.MyStore.cons1",
                "product_name": "Consumable 1",
                "product_type": "Consumable"
            },
            {
                "product_identifier": "net.yourdeveloper.MyStore.cons2",
                "product_name": "Consumable 2",
                "product_type": "Consumable"
            },
            {
                "product_identifier": "net.yourdeveloper.MyStore.upgrade",
                "product_name": "Upgrade to pro",
                "product_type": "Non-Consumable"
            }
        ]
    }
},
{
    "name": "FX",
    "products": [
        "product": [
            {
                "product_identifier": "net.yourdeveloper.FX.upgrade",
                "product_name": "Upgrade to Pro",
                "product_type": "Non consumable"
            }
        ]
    ]
}
]
}

```

With a structure like this you can support multiple applications with multiple products using a single JSON file. You can create it in any text editor and it doesn't require a back end. The only requirement is that it's accessible via a public URL and the web server is configured for the MIME type JSON to serve the contents of the file to the request.

The JSON file is available via the URL <http://developer.yourdeveloper.net/products.json>. You are free to download it and make the modifications required for your own applications and products and store it on your own server.

Implement the `retrieveProductListFromURL:applicationName:` method. It starts by initializing the local `productList` and creates and initializes the request to retrieve the information from the passed URL. The received data, which is the JSON response, is serialized into an `NSDictionary`. The `NSDictionary` is parsed, and the products that match the `applicationName` you've passed are added to the `productList` array. When all entries from the `NSDictionary` are processed, the `requestProductDetails` method is called. The method implementation is shown in Listing 16-9.

**LISTING 16-9:** The retrieveProductListFromURL:applicationName: method

```

- (void)retrieveProductListFromURL:(NSURL *)urlToJsonFile
    applicationName:(NSString *)appName;
{
    if (productList)
        productList=nil;
    productList=[[NSMutableArray alloc] init];
    NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:urlToJsonFile

        cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
        timeoutInterval:60];

    [request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
    URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
    [urlRequest startWithCompletion:^(URLRequest *request, NSData *data, BOOL success) {
        if (success)
        {
            NSError* error=nil;
            NSDictionary* resultDict =
                [NSJSONSerialization JSONObjectWithData:data
                    options:kNilOptions error:&error];
            NSDictionary* applications =
                (NSDictionary *)[resultDict objectForKey:@"applications"];
            for (NSDictionary* applicationDict in (NSDictionary* )
                [applications objectForKey:@"application"])
            {
                NSString* thisAppName = [applicationDict objectForKey:@"name"];
                //check if this is the appname we are looking for
                if ([thisAppName isEqualToString:appName])
                {
                    NSDictionary* appProducts =
                        (NSDictionary*)[applicationDict objectForKey:@"products"];
                    for (NSDictionary* product in (NSDictionary*)
                        [appProducts objectForKey:@"product"])
                    {
                        NSString* productIdentifier =
                            [product objectForKey:@"product_identifier"];
                        [productList addObject:productIdentifier];
                    }
                }
            }
            [self requestProductDetails];
        }
    else
    {
        NSLog(@"error %@", [[NSString alloc] initWithData:data
            encoding:NSUTF8StringEncoding]);
    }
}];
}

```

## Requesting Product Detail Information

After you've created the product list containing your product identifiers, create and initialize your `SKProductRequest` instance variable named `productsRequest` using the `initWithProductIdentifiers` method of the `SKProductRequest` class. Set the delegate to `self` and call the `start` method of the `SKProductRequest` class.

The `SKProductRequest` class is used to retrieve localized information about the list of products you've passed in the `initWithProductIdentifiers:` method. When the request completes, the `productRequest:didReceiveReponse:` delegate method is called, which you will implement next. The `requestProductDetails` method is shown in Listing 16-10.

**LISTING 16-10:** The `requestProductDetails` method

```
- (void)requestProductDetails
{
    NSSet *productIdentifiers = [NSSet setWithArray:productList];
    productsRequest = [[SKProductsRequest alloc]
                       initWithProductIdentifiers:productIdentifiers];
    productsRequest.delegate = self;
    [productsRequest start];
}
```

## Receiving Product Detail Information

Implement the `productRequest:didReceiveReponse:` delegate method of the `SKProductRequest` class. This delegate method is called when the `SKProductRequest start` method is finished, and it will return with a response of type `SKProductResponse`.

The `SKProductResponse` class has two public read-only properties of type `NSArray`. One is the `products` array containing the products that are available for sale, and the other is `invalidProductIdentifiers`, which contains product identifiers that are invalid. A product identifier might be invalid for the following reasons:

- ▶ Not released for sale in iTunes Connect
- ▶ Not complete or correctly configured

You assign the two response arrays to your `YDInAppPurchaseManager` properties, and post a notification named `KYDInAppPurchaseManagerProductsFetchedNotification` to inform the observers that the product details are available and can be displayed in the user interface. The `productsRequest:didReceiveResponse:` method is shown in Listing 16-11.

**LISTING 16-11:** The `productsRequest:didReceiveReponse` method

```
- (void)productsRequest:(SKProductsRequest *)request
didReceiveResponse:(SKProductsResponse *)response
{
    self.availableProducts = response.products;
```

```

    self.invalidProducts= response.invalidProductIdentifiers;
    //notify the products have been fetched
    [[NSNotificationCenter defaultCenter]
        postNotificationName:kYDInAppPurchaseManagerProductsFetchedNotification
        object:self userInfo:nil];
}

```

## Present the Store

Open the `YDViewController.h` file and import the `YDInAppPurchaseManager` header file. Create a strong property of type `UIView` named `surrogateAdvertisementView`, which will be used to simulate an advertisement that will be replaced with a real advertisement in the section “Monetizing with Advertisements.” The `YDViewController.h` implementation is shown in Listing 16-12.

### LISTING 16-12: Chapter16/MyStore/YDViewController.h

```

#import <UIKit/UIKit.h>

#import "YDInAppPurchaseManager.h"
@interface YDViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITableView *mTableView;
@property(nonatomic,strong) UIView* surrogateAdvertisementView;
@end

```

Open the `YDViewController.m` file and subscribe to the `UITableViewDataSource` and `UITableViewDelegate` protocols.

In the `viewDidLoad` method, create the `surrogateAdvertisementView` as a simple `UIView` with a blue background to simulate an advertisement at the top of the screen.

Finally, in the `viewDidLoad` method call the `initializeInAppPurchaseManager` method.

## Initialize the `YDInAppPurchaseManager`

Create the method `initializeInAppPurchaseManager`, which will start by calling the `canMakePurchases` method of the `YDInAppPurchaseManager` to see if the application is capable of making a purchase. Set up the observers to process the notifications posted by the `YDInAppPurchaseManager` and create an `NSURL` object with the URL for your JSON product list file. Finally, call the `retrieveProductListFromURL:applicationName:` method of the `YDInAppPurchaseManager` class. The implementation is shown in Listing 16-13.

### LISTING 16-13: The `initializeInAppPurchaseManager` method

```

- (void)initializeInAppPurchaseManager
{
    if ([[YDInAppPurchaseManager sharedInstance] canMakePurchases])
    {

```

*continues*

**LISTING 16-13** (continued)

```

//add notification called when products have been fetched
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(productListReceived:)
    name:kYDInAppPurchaseManagerProductsFetchedNotification object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(productPurchaseSucceeded:)
    name:kYDInAppPurchaseManagerTransactionSucceededNotification
    object:nil];

NSURL* url = [NSURL
URLWithString:@"http://developer.yourdeveloper.net/products.json"];
[[YDInAppPurchaseManager sharedInstance]
retrieveProductListFromURL:url applicationName:@"MyStore"];

}

else
{
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Error"
        message:@"you can't make purchases" delegate:self
        cancelButtonTitle:@"OK" otherButtonTitles:nil, nil];
    [alert show];
}

}

```

The `productsLoaded:` method is called once the `kYDInAppPurchaseManagerProductsFetchedNotification` notification is posted to inform you that the products have been loaded. You can simply call the `reloadData` method of your `UITableView` instance as shown in Listing 16-14.

**LISTING 16-14: The `productsLoaded:` method**

```

- (void)productListReceived:(NSNotification *)notification
{
    [self.mTableView reloadData];
}

```

## Display the Available Products

To display the available products in your `UITableView` instance you need to implement, at minimum, the mandatory `delegate` methods as you've done many times before. The `numberOfSections` `InTableView:` method returns `1` because your `UITableView` will have only one section. The `table view:numberOfRowsInSection:` method will return the value of the `availableProducts` property of the `YDInAppPurchaseManager`. This property has been set by the `productRequest:didReceive Response:` method, as shown in Listing 16-11.

To display each available product with a localized description and a localized price, implement the `tableView:cellForRowAtIndexPath:` method as follows:

1. Create a reusable `UITableViewCell`.
2. Get the `SKProduct` from the `availableProducts` array of the `YDInAppPurchaseManager` for the `indexPath.row` index.
3. Set the `LocalizedTitle` property of the retrieved `SKProduct` to the `cell.textLabel.text` property.
4. Create, initialize, and configure an `NSNumberFormatter` named `priceFormatter`.
5. Set the formatted localized price to the `cell.detailTextLabel.text` property.
6. Perform a check to see if the product has been purchased already by calling the `productPurchased:` method of the `YDInAppPurchaseManager` by passing the `productIdentifier` property of the selected `SKProduct`.
7. Display a checkmark if the product is already purchased, and if not, create a `UIButton` with title `Buy` and set the `indexPath.row` value to its `tag` property.
8. Finally, return the cell object.

The three `UITableView` delegate methods are shown in Listing 16-15.

**LISTING 16-15: The UITableView delegate methods**

```
- (void)productListReceived:(NSNotification *)notification
{
    [self.mTableView reloadData];
}

#pragma mark UITableView delegates
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger) tableView:(UITableView *) table numberOfRowsInSection:(NSInteger)section
{
    return [[[YDInAppPurchaseManager sharedInstance] availableProducts] count];
}

- (UITableViewCell *) tableView:(UITableView *) tableView
    cellForRowAtIndexPath:(NSIndexPath *) indexPath
{
    UITableViewCell *cell = (UITableViewCell *)[tableView
```

*continues*

**LISTING 16-15 (continued)**

```
dequeueReusableCellWithIdentifier:@"MyCellIdentifier"];
if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
        reuseIdentifier:@"MyCellIdentifier"];
}
cell.selectionStyle = UITableViewCellStyleNone;
SKProduct* product = [[[YDInAppPurchaseManager sharedInstance] availableProducts]
    objectAtIndex:indexPath.row];
cell.textLabel.text = product.localizedTitle ;
// Create an NSNumberFormatter
NSNumberFormatter* priceFormatter;

priceFormatter = [[NSNumberFormatter alloc] init];
[priceFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
[priceFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
[priceFormatter setLocale:product.priceLocale];

cell.detailTextLabel.text = [priceFormatter stringFromNumber:product.price];

if ([[YDInAppPurchaseManager sharedInstance]
    productPurchased:product.productIdentifier]) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
    cell.accessoryView = nil;
} else {
    UIButton* buyButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    buyButton.frame = CGRectMake(0, 0, 72, 37);
    [buyButton setTitle:@"Buy" forState:UIControlStateNormal];
    buyButton.tag = indexPath.row;
    [buyButton addTarget:self action:@selector(buyButtonTapped:)
        forControlEvents:UIControlEventTouchUpInside];
    cell.accessoryType = UITableViewCellAccessoryNone;
    cell.accessoryView = buyButton;
}

return cell;
}
```

The result is a user interface that will look similar to the one shown in Figure 16-6.

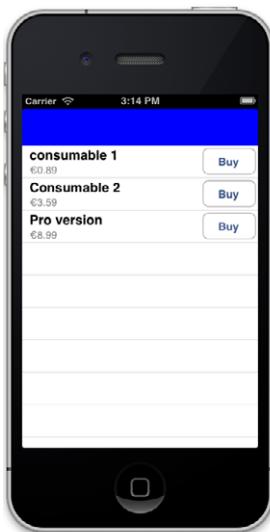


FIGURE 16-6

## Buy a Product

Now that your user interface is displaying a table with available products and a Buy button for each product, you need to implement the `buyButtonTapped:` method, which is invoked if the user taps one of the visible Buy buttons.

This method creates a `UIButton` named `buyButton` by casting the `sender` value. Next, it creates an `SKProduct` instance named `product` from the `availableProducts` array of the `YDInAppPurchaseManager` class by using the `objectAtIndex:` method. It then calls the `purchaseProduct:` method of the `YDInAppPurchaseManager` class, as shown in Listing 16-16.

### LISTING 16-16: The `buyButtonTapped:` method

```
- (void)buyButtonTapped:(id)sender
{
    UIButton* buyButton = (UIButton *)sender;
    SKProduct* product = [[[YDInAppPurchaseManager sharedInstance] availableProducts]
                           objectAtIndex:buyButton.tag];
    [[YDInAppPurchaseManager sharedInstance] purchaseProduct:product];
}
```

## Sending a Payment Request

Open the `YDInAppPurchaseManager` and create the `purchaseProduct:` method. This method calls the `defaultQueue addTransactionObserver:` method of the default `SKPaymentQueue`. It creates and initializes a variable named `payment` of type `SKPayment` by calling the `paymentWithProduct:` method of the `SKPayment` class. The payment instance is added to the default `SKPayment` queue by calling the `addPayment:` method, as shown in Listing 16-17.

### LISTING 16-17: The `purchaseProduct:` method

```
- (void)purchaseProduct:(SKProduct*)productToPurchase
{
    [[SKPaymentQueue defaultQueue] addTransactionObserver:self];
    SKPayment* payment = [SKPayment paymentWithProduct:productToPurchase];
    [[SKPaymentQueue defaultQueue] addPayment:payment];
}
```

You've added a `TransactionObserver` for the `SKPaymentQueue`, and now you implement the observer itself by implementing the `paymentQueue:updatedTransactions:` method as shown in Listing 16-18. Based on the transaction state, one of the following methods is called:

- `completeTransaction:` if the transaction has been completed
- `failedTransaction:` if the transaction fails or if the user taps Cancel
- `restoreTransaction:` if the transaction needs to be restored

### LISTING 16-18: The `paymentQueue:updatedTransactions:` method

```
#pragma mark SKPaymentTransactionObserver methods

// called when the transaction status is updated
- (void)paymentQueue:(SKPaymentQueue *)queue updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction *transaction in transactions)
    {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
                break;
            default:
                break;
        }
    }
}
```

```

    }

- (void)completeTransaction:(SKPaymentTransaction *)transaction
{
    [self finishTransaction:transaction wasSuccessful:YES];
}

- (void)restoreTransaction:(SKPaymentTransaction *)transaction
{
    [self finishTransaction:transaction wasSuccessful:YES];
}

- (void)failedTransaction:(SKPaymentTransaction *)transaction
{
    if (transaction.error.code != SKErrorPaymentCancelled)
    {
        // error!
        [self finishTransaction:transaction wasSuccessful:NO];
    }
    else
    {
        // this is fine, the user just cancelled, so don't notify
        [[SKPaymentQueue defaultQueue] finishTransaction:transaction];
    }
}

```

Both the `completeTransaction:` and the `restoreTransaction:` methods call the `finishTransaction:wasSuccessful` method, which you implement next.

## Processing the Transaction Receipt

The `finishTransaction:wasSuccessful` method is called once the payment transaction has finalized. If the result of the transaction was not successful, a `kYDInAppPurchaseManagerTransactionFailedNotification` notification is posted, for which you've set up an observer in the `YDViewController.m` file and implemented a method to handle the response.

You start by calling the `finishTransaction:` method on the default `SKPaymentQueue`.

## Verifying the Transaction Receipt

Your application now should perform the additional step to verify that the payment you received from Store Kit came from Apple. To verify the receipt, implement the following steps:

1. Retrieve the receipt data from the `transactionReceipt` property and encode it as a `base64` string using the `NSString+base64` category.
2. Create a JSON object with a single key named `receipt-data` and the `base64`-encoded string as the value.
3. Post the JSON object using an HTTP POST request to the verification URL using the `URLRequest` class.

4. The received response, which is again a JSON object with two keys—status and receipt—should have the value 0 for the status key in case of a valid receipt. Any value other than 0 means the receipt is invalid.
5. Once you've verified the receipt is valid, post a notification named `KYDInAppPurchaseManagerTransactionSucceededNotification` to inform the observer that the payment has succeeded and has been verified.
6. Finally, post a notification named `KYDInAppPurchaseManagerDeliverProductNotification` to inform the observer that the payment has succeeded and has been verified.

Note there are two URLs: <https://buy.itunes.apple.com/verifyReceipt> for production and <https://sandbox.itunes.apple.com/verifyReceipt> for the sandbox environment. The `finishTransaction:wasSuccessful:` method is shown in Listing 16-19.

#### LISTING 16-19: The `finishTransaction:wasSuccessful:` method

```
- (void)finishTransaction:(SKPaymentTransaction *)transaction
                  wasSuccessful:(BOOL)wasSuccessful
{
    // remove the transaction from the payment queue.
    [[SKPaymentQueue defaultQueue] finishTransaction:transaction];

    NSDictionary *userInfo = [NSDictionary dictionaryWithObjectsAndKeys:
                             transaction, @"transaction" , nil];
    if (wasSuccessful)
    {
        //Verify the transaction receipt if it really came from Apple
        NSError *jsonSerializationError = nil;

        NSString *transactionReceiptAsString = [NSString
                                                base64StringFromData:transaction.transactionReceipt
                                                length:[transaction.transactionReceipt length]];
        NSDictionary *receiptDictionary = [[NSDictionary alloc]
                                           initWithObjectsAndKeys:transactionReceiptAsString,
                                           @"receipt-data", nil];
        id receiptDictionaryAsData = [NSJSONSerialization
                                      dataWithJSONObject:receiptDictionary
                                      options:NSJSONWritingPrettyPrinted
                                      error:&jsonSerializationError];

        NSURL *sandboxStoreURL = [[NSURL alloc]
                                   initWithString:kSKSandboxTransctionReceiptVerifierURL];
        NSURL *storeURL = [[NSURL alloc] initWithString: kSKTransactionReceiptVerifierURL];
        NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:sandboxStoreURL
                                                               cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
                                                               timeoutInterval:60];
        [request setHTTPMethod:@"POST"];
        [request setHTTPBody:receiptDictionaryAsData];
        [request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
        URLRequest *urlRequest = [[URLRequest alloc] initWithRequest:request];
```

## Delivering the Product

The notification observer you configured will be invoked by the `finishTransaction:wasSuccessful:` method. As a last step you need to open the `YDViewController.m` file and implement the following four methods to handle the received notifications:

- productPurchaseSucceeded:
  - productPurchaseFailed:
  - deliverProduct:
  - upgradeToProVersion

The `productPurchaseSucceeded:` method displays a `UIAlertView` to inform the user that the transaction has succeeded. The `mTableView` is reloaded to change the Buy button to a checkmark, and you can perform a test to see if the `productIdentifier` of the purchased product is a product you have defined to unlock certain features, such as an Upgrade to Pro version. In this example the `productIdentifier` `net.yourdeveloper.MyStore.upgrade` indicated an Upgrade to Pro version product, which will remove advertisements.

Implement the `upgradeToProVersion` method, which simply removes the `surrogateAdvertisementView` from the `superView` and repositions the `UITableView` to a full screen.

The notification methods are shown in Listing 16-20.

#### LISTING 16-20: Notification methods

```

- (void)productPurchaseSucceeded:(NSNotification *)notification
{
    [self.mTableView reloadData];
    SKPaymentTransaction * transaction = (SKPaymentTransaction *)[notification.userInfo
        objectForKey:@"transaction"];
    UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Thanks!"
        message:@"for your purchase."
        delegate:nil
        cancelButtonTitle:nil
        otherButtonTitles:@"OK", nil];

    [alert show];
    //Test if it's our upgrade product
    if ([transaction.payment.productIdentifier
        isEqualToString:@"net.yourdeveloper.MyStore.upgrade"])
    {
        [self upgradeToProVersion];
    }
}

- (void)productPurchaseFailed:(NSNotification *)notification
{
    SKPaymentTransaction * transaction = (SKPaymentTransaction *)[notification.userInfo
        objectForKey:@"transaction"];
    if (transaction.error.code != SKErrorPaymentCancelled) {
        UIAlertView* alert = [[UIAlertView alloc] initWithTitle:@"Error!"

            message:transaction.error.localizedDescription
            delegate:nil
            cancelButtonTitle:nil
            otherButtonTitles:@"OK", nil];

        [alert show];
    }
}

- (void)deliverProduct:(NSNotification *)notification
{
    SKPaymentTransaction * transaction = (SKPaymentTransaction *)[notification.userInfo
        objectForKey:@"transaction"];
    //Test if it's our upgrade product
    if ([transaction.payment.productIdentifier
        isEqualToString:@"net.yourdeveloper.MyStore.upgrade"])
    {
        [self upgradeToProVersion];
    }
}

```

```

        else
        {
            //Deliver the product e.g. opening a level in a game or download a PDF
        }
    }
- (void)upgradeToProVersion
{
    //remove the surrogateView from the superview
    [self.surrogateAdvertisementView removeFromSuperview];
    self.mTableView.frame = [[UIScreen mainScreen] bounds];
    [self.mTableView reloadData];
}

```

You have now implemented the complete cycle of process steps required to display In-App Purchase products from an external source, display them in a user interface with a Buy button, perform the payment transaction, verify the payment receipt, and inform the user with the results.

The `YDInAppPurchaseManager` class should definitely be part of the Private Library project you developed in Chapter 1 if you aim to use the In-App Purchase functionality in your applications.

## MONETIZING WITH ADVERTISEMENTS

Hundreds of advertising platforms are available and it's not within the scope of this book to describe all of them. I'll describe the implementation logic of two advertising platforms:

- **iAd:** The advertising platform from Apple.
- **AdMob:** The advertising platform from Google.

The reason only these two platforms are covered in this book is simple. The iAd platform is Apple's platform and this book is about iOS technology, so the iAd platform is within the scope. AdMob is the advertising platform from Google. It is one of the largest platforms available and has a dominant market position.

## Introducing the iAd Framework

The iAd framework does all the necessary work to download advertisements from the iAd network, which you can present on the user interface of your application.

To use the iAd framework and display advertisements from the iAd network, you must agree to the iAd network license agreement, which is available in the member center of the developer portal.

iTunes Connect is where you configure whether or not your application supports iAds.

You can find the iAd framework reference at [http://developer.apple.com/library/ios/#documentation/UserExperience/Reference/iAd\\_ReferenceCollection/\\_index.html](http://developer.apple.com/library/ios/#documentation/UserExperience/Reference/iAd_ReferenceCollection/_index.html).

The iAd programming guide is available at [http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/iAd\\_Guide](http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/iAd_Guide).

Two types of advertisements are available:

- **Banner views:** These use a portion of the screen to display a banner Ad.
- **Full-screen advertisements:** These are available only on the iPad and display a page of content served by the iAd network.

When a user taps the presented advertisement, it will typically cover the entire screen while the application continues running. As a result, the user cannot see the user interface of your application and you might want to postpone certain functionalities until the advertisement is no longer visible.

You also have the option to cancel the displayed advertisement when your application requires the user's attention.

## Implementing iAd Advertising

Start Xcode and create a new project using the Single View Application Project template, and name it `MyiAdDemo` using the options shown in Figure 16-7.

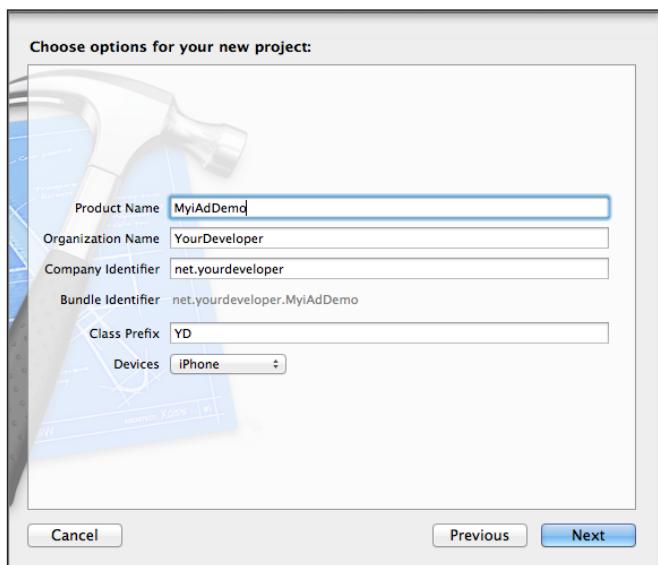


FIGURE 16-7

Add the iAd framework to your project, open the `YDViewController.m` file, and import the iAd header file.

Subscribe to the `ADBANNERVIEWDELEGATE` protocol and create a private variable named `adBannerview` of type `ADBANNERVIEW` and a `BOOL` named `bannerIsVisible`.

In the `viewDidLoad` method, create and initialize the `adBannerView` with a zero frame and call the `setAutoresizingMask:` of the `adBannerView`. Set the `delegate` property of the `adBannerView` to `self` and add the `adBannerview` as a subview to the current view.

If you launch your application now, an advertisement will display if the network can serve one. Because the network will never have a 100 percent fill rate (the fill rate is the ratio of advertisements served in relation to the number of requests made), a white frame might display instead of an actual advertisement.

Because you have subscribed to the `ADBannerViewDelegate` protocol and set the delegate to `self`, you need to implement the `bannerViewDidLoadAd:` delegate method, which is invoked once an advertisement is loaded; and the `bannerView:didFailToReceiveAdWithError:` delegate method, which is invoked if the network can't serve an advertisement.

Finally, you need to implement the `willRotateToInterfaceOrientation:duration:` method so the advertisement will display correctly when the user rotates the device. The complete implementation of the `YDViewController.m` is shown in Listing 16-21.

**LISTING 16-21: Chapter16/MyAdDemo/YDViewController.m**

```
#import "YDViewController.h"
#import <iAd/iAd.h>
@interface YDViewController ()<ADBannerViewDelegate>
{
    ADBannerView* adBannerView;
    BOOL bannerIsVisible;
}
@end

@implementation YDViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    adBannerView = [[ADBannerView alloc] initWithFrame:CGRectMake(0, 20, self.view.frame.size.width, 50)];
    [adBannerView setAutoresizingMask:UIViewAutoresizingFlexibleWidth];
    adBannerView.delegate=self;
    [self.view addSubview:adBannerView];
}

- (void)bannerViewDidLoadAd:(ADBannerView *)banner
{
    bannerIsVisible=YES;
}
- (void)bannerView:(ADBannerView *)banner didFailToReceiveAdWithError:(NSError *)error
{
    bannerIsVisible=NO;
}

- (void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)
    toInterfaceOrientation duration:(NSTimeInterval)duration
{
    if (UIInterfaceOrientationIsLandscape(toInterfaceOrientation))
```

*continues*

**LISTING 16-21 (continued)**

```

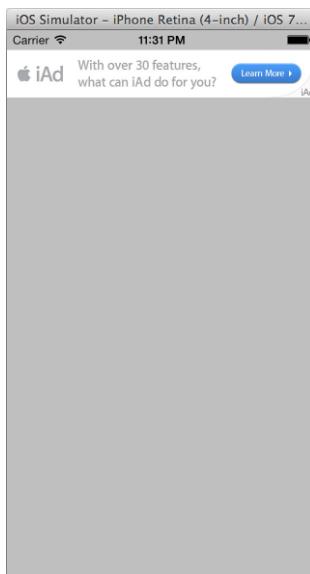
        [adBannerView setAutoresizingMask:UIViewAutoresizingFlexibleWidth];
    else
        [adBannerView setAutoresizingMask:UIViewAutoresizingFlexibleWidth];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end

```

When you launch your application, it will look as shown in Figure 16-8 and Figure 16-9. Figure 16-8 shows a portrait view of the application while figure 16-9 shows a landscape view of the application.



**FIGURE 16-8**



**FIGURE 16-9**

## Implementing the AdMob Network

AdMob is the mobile advertising network from Google available on [www.admob.com](http://www.admob.com). You can create an account for free to become a publisher and you can start immediately.

Once you've created an account, you can create your application by selecting Sites and Apps from the drop-down navigation menu and clicking Add Site/App.

Select the app type, which will be an iPhone App or an iPad App, and complete the details screen. When you click the Save button, the screen displays a button to download the SDK. Download the SDK and click the Go to Sites/Apps button, which presents you with the list of Sites/Apps available under your account.

Navigate to the application you've just created, and when you hover over that line, you will see two buttons: a Reporting button and a Manage Settings button. Click the Manage Settings button and you will see a Publisher ID under the application name. You need this Publisher ID to implement the AdMob SDK.

## Displaying AdMob Advertisements

To display the AdMob advertisements, open the `Mystore` project you created earlier and implement the display of AdMob advertisements as a replacement for the `surrogateAdvertisementView`.

After you've downloaded the AdMob SDK and opened the `Mystore` project in Xcode, drag and drop the following files into your project:

- `libGoogleAdMobAd.a`
- `GADAdMobExtras.h`
- `GADAdNetworkExtras.h`
- `GADBannerViewDelegate.h`
- `GADInterstitial.h`
- `GADInterstitialDelegate.h`
- `GADRequest.h`
- `GADRequestError.h`
- `GADAdSize.h`
- `GADBannerView.h`

You need to add the following frameworks to your project to work with AdMob:

- `AdSupport`
- `SystemConfiguration`
- `MessageUI`
- `AudioToolbox`
- `CFNetwork`
- `StoreKit` (which you already have added)

Finally, add `-ObjC` under Other Linker Flags in the Build Settings tab of your application as shown in Figure 16-10.

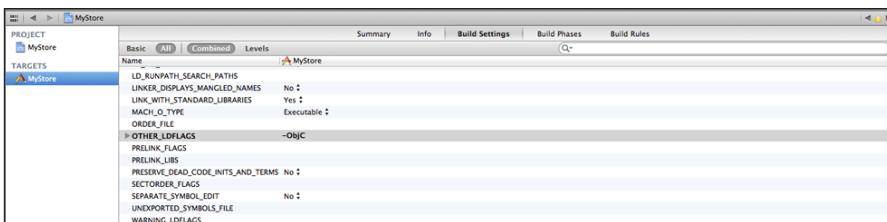


FIGURE 16-10

Open the `YDViewController.m` file and import the `GADBannerView` header file.

In the private interface, declare a variable named `bannerView` of type `GADBannerView`.

Change the `viewDidLoad` method as shown in Listing 16-22.

#### LISTING 16-22: The viewDidLoad method

```
- (void)viewDidLoad
{
    [super viewDidLoad];
/*
self.surrogateAdvertisementView=[[UIView alloc] initWithFrame:
                                CGRectMake(0,0,320,50)];
self.surrogateAdvertisementView.backgroundColor=[UIColor blueColor];
[self.view addSubview:self.surrogateAdvertisementView];
*/
[self initializeInAppPurchaseManager];
[self displayAdvertisement];
}
```

Create a new method named `displayAdvertisement`, which is responsible for displaying the AdMob advertisement.

It starts by checking whether the user purchased the Upgrade to Pro product, which would remove the advertising. It does this by calling the `productPurchased:` method of the `YDInAppPurchaseManager` by passing the product identifier of the Upgrade to Pro product. If the upgrade has not been purchased, the `bannerView` is created and initialized and added as a subview of the current view.

Finally, the `loadRequest` method of the `GADBannerView` class is called, which is responsible for the communication with the advertising network. The complete `displayAdvertisement` method is shown in Listing 16-23.

#### LISTING 16-23: The displayAdvertisement method

```
- (void)displayAdvertisement
{
    if (![[YDInAppPurchaseManager sharedInstance]
          productPurchased:@"net.yourdeveloper.MyStore.upgrade"])
    {
```

```

// Create the GADBannerview
bannerView = [[GADBannerview alloc] initWithFrame:
              CGRectMake(0,0,
                         GAD_SIZE_320x50.width,
                         GAD_SIZE_320x50.height)];
// Specify your AdMob Publisher ID.
bannerView.adUnitID = @"a14f0070e2b8015";
// Let the runtime know which UIViewController to restore after taking
// the user wherever the ad goes and add it to the view hierarchy.
bannerView.rootViewController = self;
[self.view addSubview:bannerView];

GADRequest *request = [GADRequest request];
request.testing=YES; // or NO
// Initiate a generic request to load it with an ad.
[bannerView loadRequest:[GADRequest request]];
}

else
{
    self.mTableView.frame = [[UIScreen mainScreen] bounds];
}
}

```

The `upgradeToProVersion` method needs a small change. Instead of removing the surrogate `AdvertisementView` from the `superView`, remove the `bannerView` from the `superView` as shown in Listing 16-24.

LISTING 16-24: Changed upgradeToProVersion method

```
- (void)upgradeToProVersion
{
    //remove the surrogateView from the superview
    //[[self.surrogateAdvertisementView removeFromSuperview];
    [bannerView removeFromSuperview];
    self.mTableView.frame = [[UIScreen mainScreen] bounds];
    [self.mTableView reloadData];
}
```

The complete functional `Mystore` project, including the AdMob SDK, is available in the download of this chapter.

## SUMMARY

In this chapter you learned different ways to monetize your application. You learned to implement In-App Purchases in your application using the Store Kit framework, and how to implement advertisements using the iAd and AdMob advertising networks.

In the next chapter you learn how to create certificates and provisioning profiles, which you need to sign, build, and distribute your applications.



# 17

## Understanding iTunes Connect

### WHAT'S IN THIS CHAPTER?

---

- Using iTunes Connect
- Understanding and using certificates and profiles
- Understanding the difference between development and production certificates and profiles

### WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 17 download and individually named according to the names throughout the chapter.

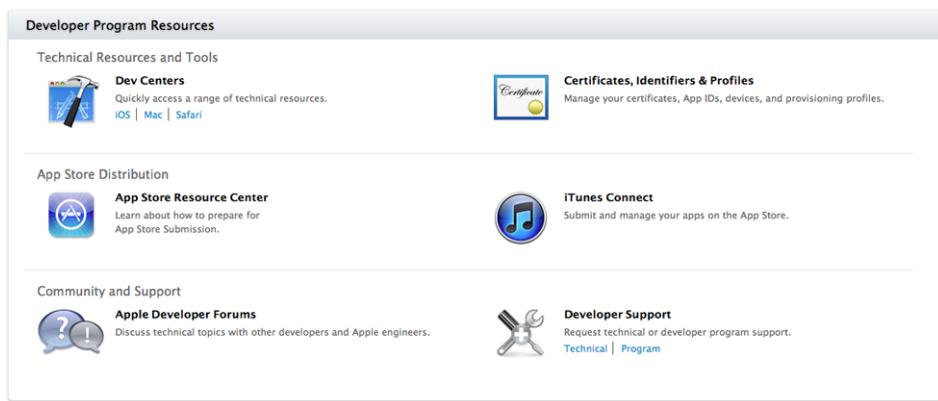
During the development process you have been able to run your application on the simulator as well as on one of the devices you've connected directly to your Mac. Before you can publish your application to the App Store, however, you need to have it tested by your client or a selection of users.

**WARNING** *Before you give others access to a version of your application for testing, it's very important that you test the application on a real device. A lot of functionalities aren't available on the simulator, such as dialing a phone number or displaying a UIImagePickerController. You must also realize that when you run your application on the simulator, its available memory is almost equal to the available memory of your Mac, and the same applies for the available storage capacity. This means that it's almost impossible to hit the hardware boundaries when running an application in the simulator.*

## IOS DEVELOPER MEMBER CENTER

Once you've enrolled in an Apple Developer Program, you have access to the Developer Member Center.

Visit the <https://developer.apple.com> website and log in with your credentials. After you have successfully logged in, you will see a screen similar to the one shown in Figure 17-1.



**FIGURE 17-1**

The Certificates, Identifiers and Profiles link brings you to the place where you can manage your certificates, identifiers, devices, and provisioning profiles.

The iTunes Connect link opens iTunes Connect, where you can manage your application in the App Store. Using iTunes Connect is explained in Chapter 18.

## Obtaining a Developer Certificate

You can test your application on the simulator, but it's important to do your final tests on a real device. To run an application on your device, the device must be connected to your Mac, it must be enabled for development, and it must be recognized by Apple. You can achieve this by creating a development certificate that identifies you, the application, and the device.

Log in to the Member Center and click the Certificates, Identifiers and Profiles link. The screen displays your existing certificates, and you can add a new certificate by clicking the + button.

On the page that appears, select iOS App Development and click the Continue button as shown in Figure 17-2.

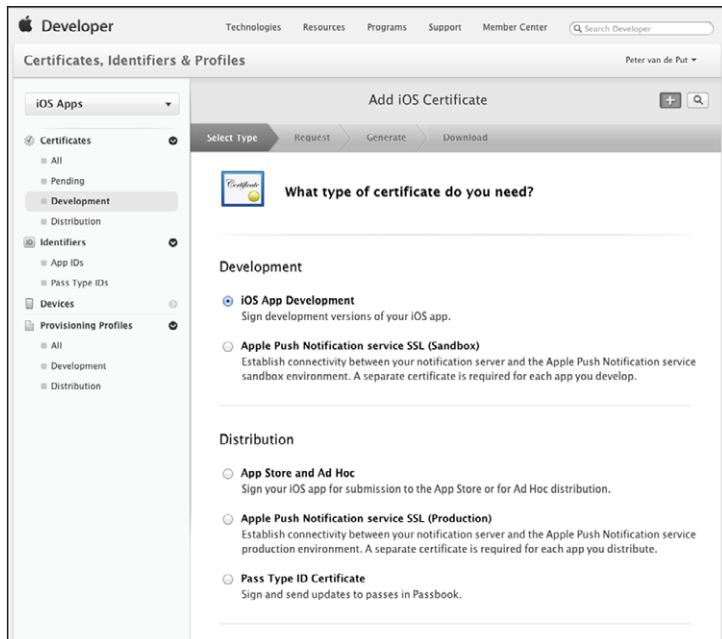


FIGURE 17-2

The next page explains how to create a Certificate Signing Request, also known as CSR. Read the instructions to create the CSR file as shown Figure 17-3.

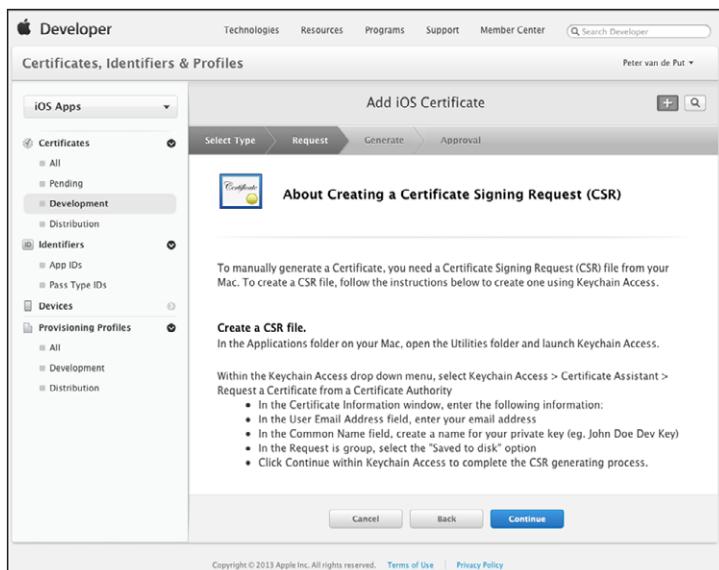


FIGURE 17-3

Click the Continue button and enter your details as shown in Figure 17-4.

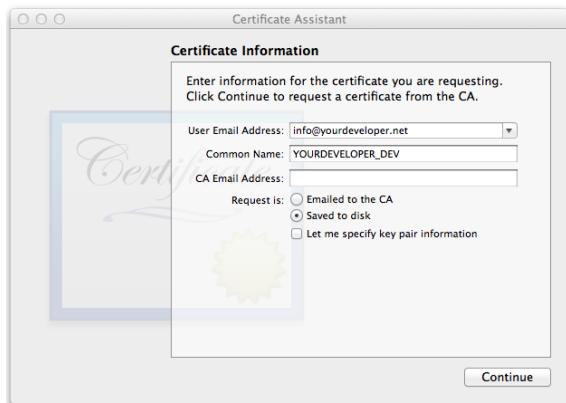


FIGURE 17-4

Once you've entered your details, save the file as shown in Figure 17-5.

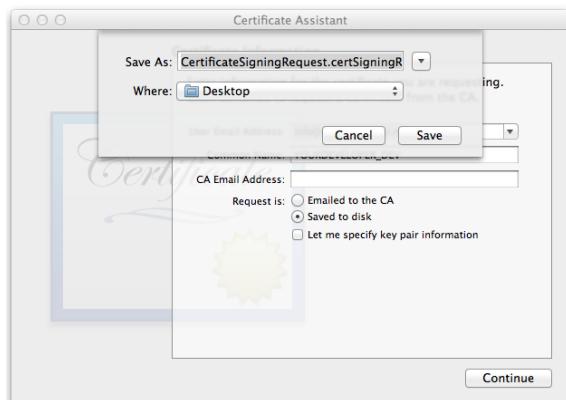


FIGURE 17-5

Clicking the Save button brings up the screen in Figure 17-6.

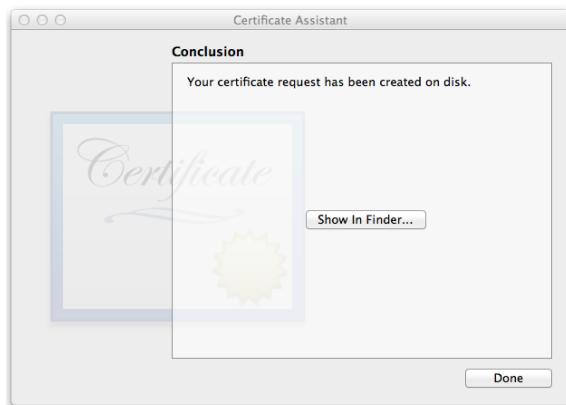


FIGURE 17-6

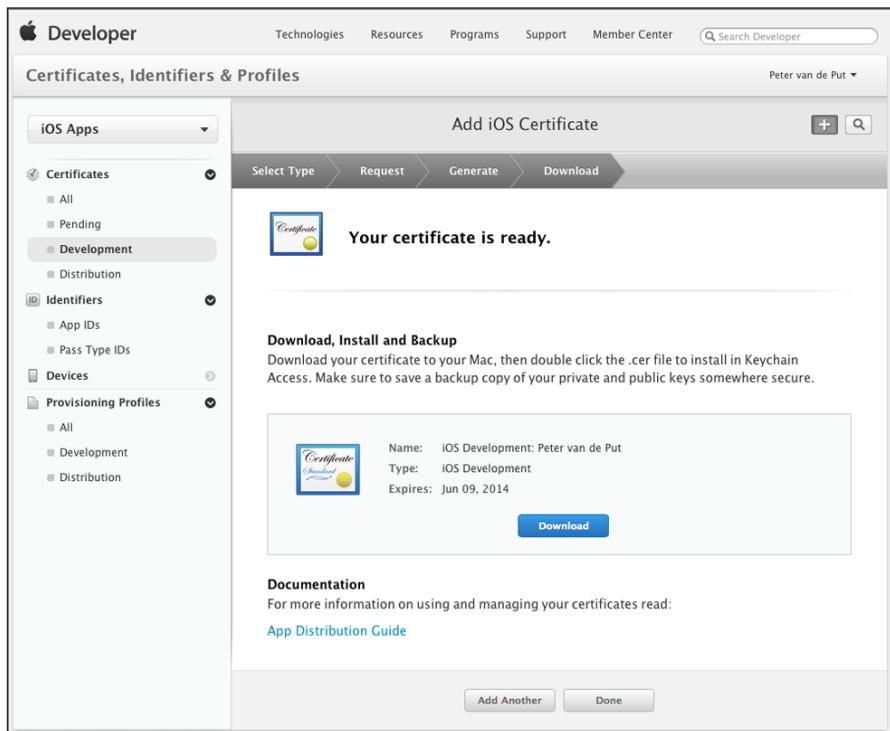
Click the Done button to finalize this step.

Figure 17-7 shows the web page where you can upload your CSR file. Select your file and click the Generate button to create your developer certificate.

A screenshot of the Apple Developer website's 'Certificates, Identifiers &amp; Profiles' section. On the left, there is a sidebar with categories: 'iOS Apps' (selected), 'Certificates' (selected), 'Identifiers', 'Devices', and 'Provisioning Profiles'. Under 'Certificates', 'Development' is selected. The main content area is titled 'Add iOS Certificate' and shows a progress bar: 'Select Type' (done), 'Request' (done), 'Generate' (in progress, indicated by a grey arrow), and 'Download' (not yet reached). Below the progress bar, there is a section titled 'Generate your certificate.' with a sub-instruction: 'With the creation of your CSR, Keychain Access simultaneously generated a public and private key pair. Your private key is stored on your Mac in the login Keychain by default and can be viewed in the Keychain Access application under the "Keys" category. Your requested certificate will be the public half of your key pair.' There is also a 'Upload CSR file.' section with a 'Choose File...' button and a file input field containing 'CertificateSigningRequest.certSigningRequest'. At the bottom of the page are 'Cancel', 'Back', and 'Generate' buttons. The footer of the page includes copyright information: 'Copyright © 2013 Apple Inc. All rights reserved.' and links to 'Terms of Use' and 'Privacy Policy'.

FIGURE 17-7

Once it's ready, you are notified and can download the certificate, as shown in Figure 17-8.



**FIGURE 17-8**

Click the Download button to download your developer certificate. The certificate will be downloaded into the Downloads folder.

Now you can double-click the certificate in Finder to install the certificate in your keychain, or you can import the certificate in the Keychain Access application manually using the Import option. Which method you choose depends on how you have secured your Mac and configured your keychain.

## Managing Devices

A device is, as the name implies, a physical device, which can be an iPod Touch, an iPhone, or an iPad. You need to register devices in the developer portal if you want to build Ad Hoc distributions, which are explained in Chapter 18.

To find the unique identifier of your device, start Xcode and select Window  $\Rightarrow$  Organizer from the menu. The Library pane on the left lists your devices, and when you click one (for example, your iPhone), you see the details of the device on the right. The Identifier field is the unique identifier for this device, as shown in Figure 17-9.

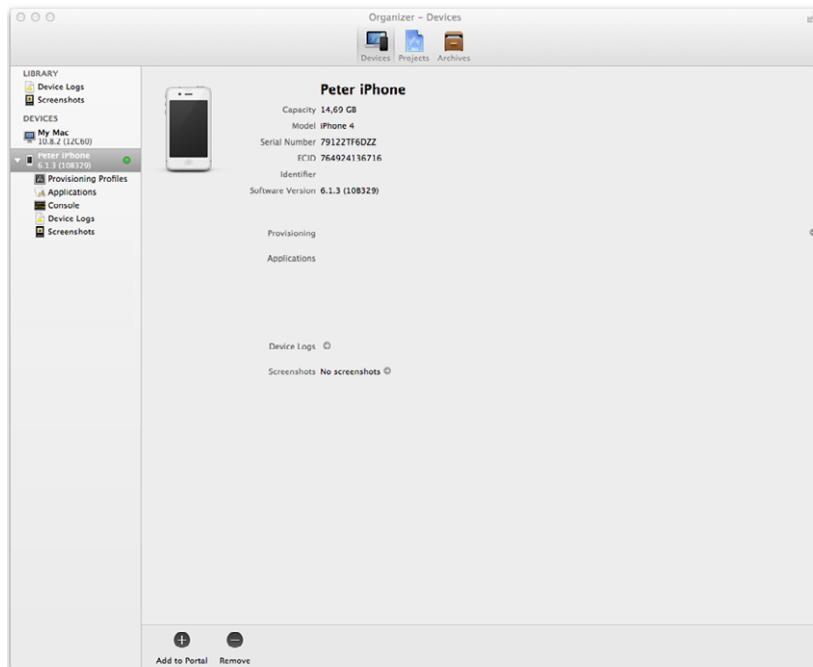


FIGURE 17-9

Back on the Certificates, Identifiers and Profiles page in the Member Center, click the Devices link under iOS Apps, as shown in Figure 17-10.

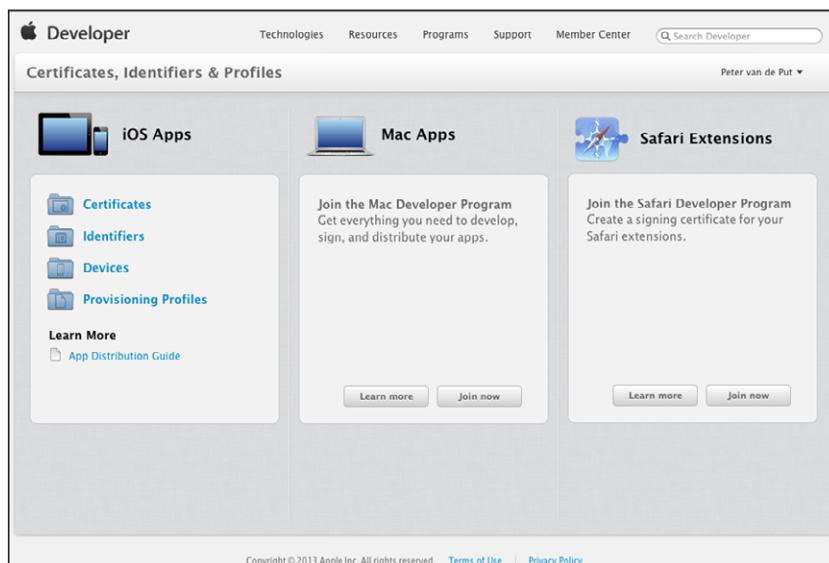


FIGURE 17-10

If you have already registered some devices under this developer account, you can see them here as shown in Figure 17-11.

The screenshot shows the 'Certificates, Identifiers & Profiles' section of the Apple Developer portal. On the left, there's a sidebar with filters for 'iOS Apps', 'Certificates' (showing 'All' selected), 'Identifiers' (showing 'All' selected), 'Devices' (showing 'All' selected), and 'Provisioning Profiles'. The main area is titled 'iOS Devices' and displays a table with columns 'Name' and 'UDID'. A message at the top says 'You can register 97 additional devices.' There are three rows of device entries, each with a redacted name and a redacted UDID.

FIGURE 17-11

To register a device, click the + button and enter a Name and a UDID for a device. In the UDID field, enter the unique identifier for the device you found earlier as shown in Figure 17-12.

The dialog box is titled 'Register Device' and contains instructions: 'Name your device and enter its Unique Device Identifier (UDID)'. It has two input fields: 'Name:' containing 'PeteriPhone 5S' and 'UDID:' which is currently empty. Below these fields is a section titled 'Register Multiple Devices' with instructions: 'Upload a file containing the devices you wish to register. Please note that a maximum of 100 devices can be included in your file and it may take a few minutes to process.' A link 'Download sample files' is provided. At the bottom are 'Cancel' and 'Continue' buttons.

FIGURE 17-12

You also can register multiple devices at once by uploading a file containing the devices you want to register. A link with a sample file to download is available on this page.

**WARNING** You can register a maximum of 100 devices under a normal developer account. You can disable devices, but you can't delete them. If the limitation to register 100 devices is a problem, you can contact Apple to purchase space for additional devices.

## Managing Apps

When you want to publish your application to the App Store you need to create an application identifier as a preparation for creating the provisioning profile(s).

Two areas in the Member Center are involved in creating your application. You create your application ID and configure it under Certificates, Identifiers and Profiles, and in the iTunes Connect area you need to configure the application with its metadata, price, and artwork as it will show in the App Store.

Under Identifiers, click the App IDs node and the right pane will display your registered iOS App IDs, as shown in Figure 17-13.

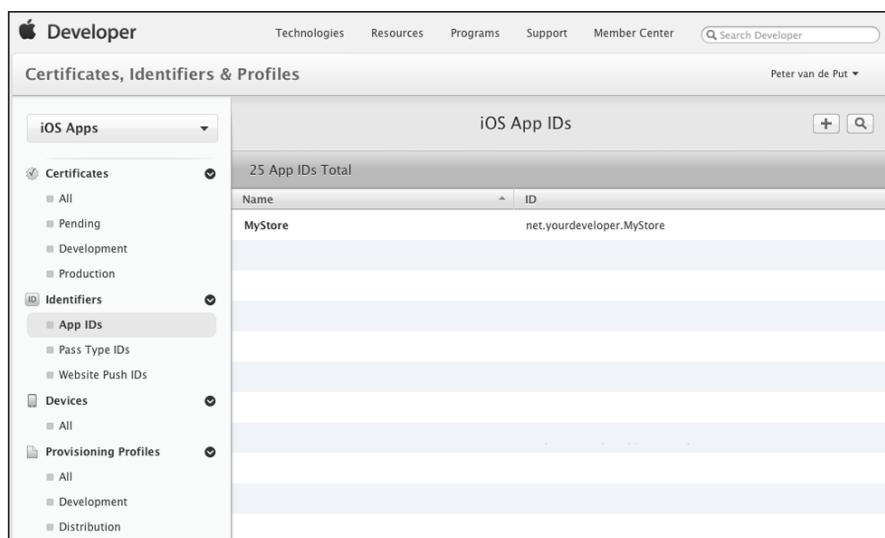


FIGURE 17-13

Click the + button to create a new application, and you are presented with a page to enter your application details.

You need to enter an App ID Description (for example, Dummy), which is only used internally and is not visible in the App Store.

The App ID Prefix field presents you with a list of values; normally you would select the one with (Team ID) at the end.

If you plan to use application services such as Game Center, In-App Purchases, Data Protection, iCloud, Inter-App Audio, Passbook, or Push Notifications, you have to define an Explicit App ID in the Bundle ID field.

The standard recommendation is to use a reverse-domain name style string like `net.yourdeveloper.Dummy`. Under the heading App Services, select the options that are applicable for your application; in this case this would only be In-App Purchases. Once you've filled in all the fields and selected the application services, click the Continue button. An example is shown in Figure 17-14.

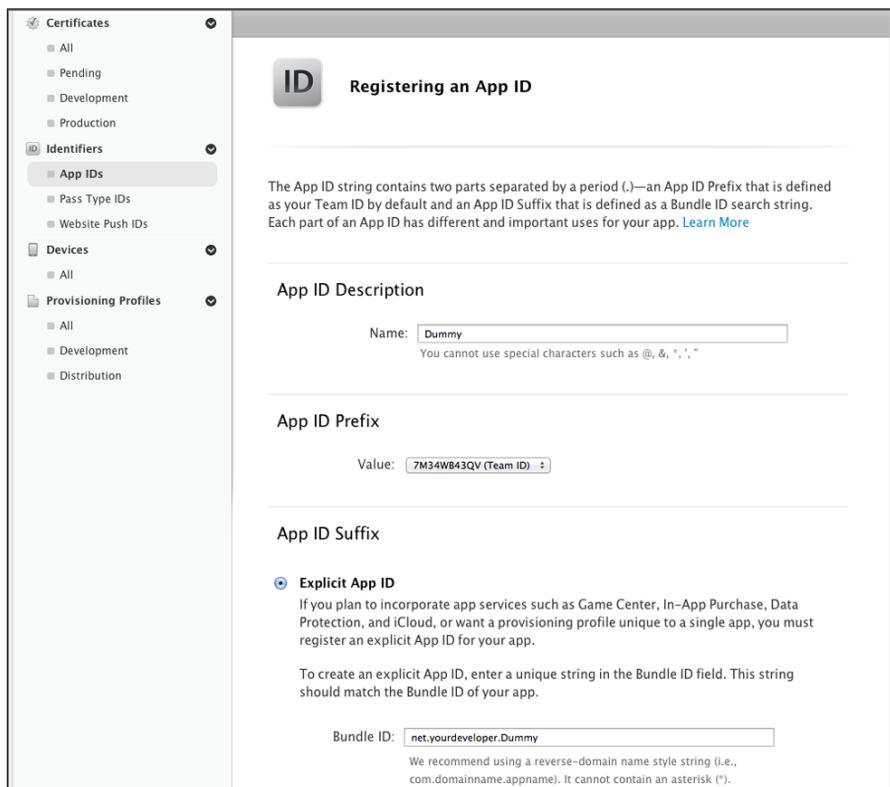


FIGURE 17-14

You'll see a confirmation page summarizing the application information you have configured, as shown in Figure 17-15.

Verify the information and click the Submit button to create your application. Once the system has verified all the information you've submitted, it presents a confirmation screen informing you that the registration has completed, as shown in Figure 17-16.

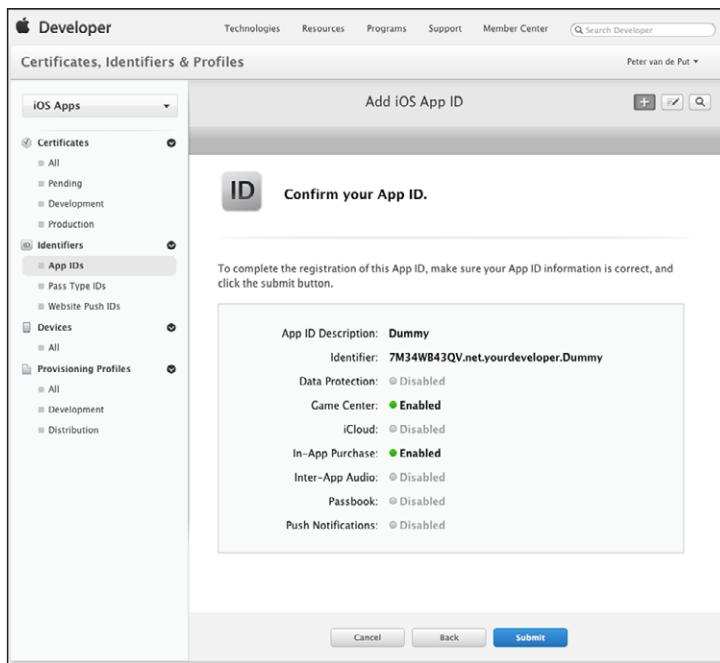


FIGURE 17-15

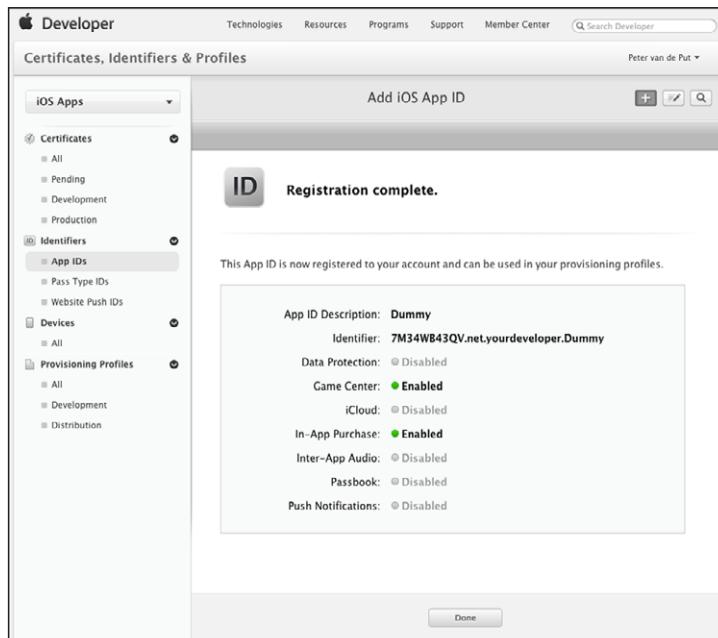
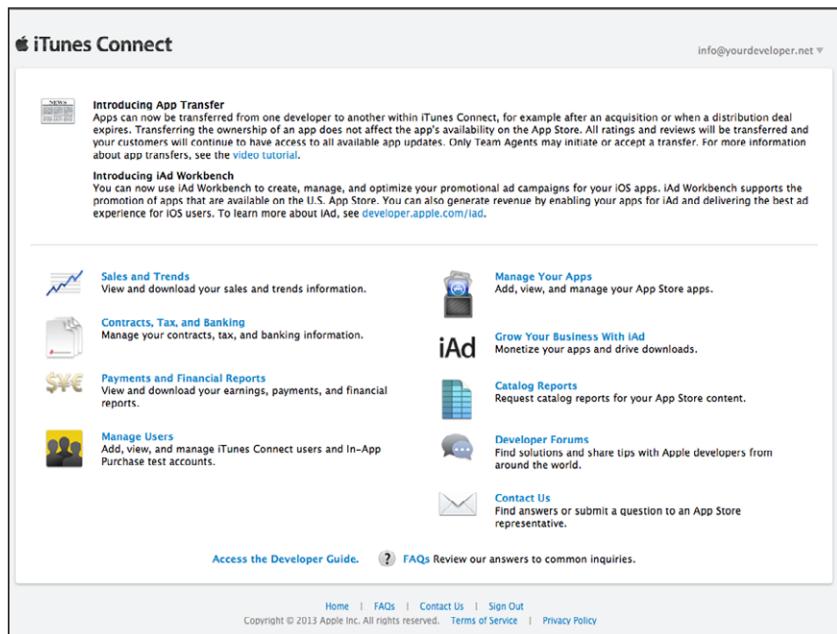


FIGURE 17-16

Your application has now been registered and you can prepare the submission to the App Store.

Go back to the homepage of the Member Center and click the iTunes Connect link to submit and manage your applications in the App Store. Figure 17-17 shows the homepage of iTunes Connect.



**FIGURE 17-17**

In the right column, click the Manage Your Apps link; this presents a new page in which all your applications are shown.

To add your new application, click the Add New App button located at the top-left corner of the page. On the next page, enter the App Information. Select the default language from the drop-down list and enter the App Name as it will appear in the App Store. The SKU number is an article number for your reference. From the Bundle ID drop-down list, select the Bundle ID you just created (`net.yourdeveloper.Dummy`) as shown in Figure 17-18.

Click the Continue button and select the availability date and price tier for your application. If your application is free, select Free from the Price Tier drop-down list, otherwise select the price tier you want to apply for this application. A sample configuration is shown in Figure 17-19.

**iTunes Connect**

### App Information

Enter the following information about your app.

Default Language	English	(?)
App Name	My Dummy Application	(?)
SKU Number	MyDumm	(?)
Bundle ID	Dummy - net.yourdeveloper.Dummy	(?)

The name of your app as it will appear on the App Store. Note that this name cannot be longer than 255 bytes.

You can register a new Bundle ID [here](#).

**Note:** Note that the Bundle ID cannot be changed if the first version of your app has been approved or if you have enabled Game Center or the iAd Network.

Does your app have specific device requirements? [Learn more](#)

**Cancel** **Continue**

Home | FAQs | Contact Us | Sign Out  
Copyright © 2013 Apple Inc. All rights reserved. Terms of Service | Privacy Policy

FIGURE 17-18

**iTunes Connect**

### My Dummy Application

Select the availability date and price tier for your app.

Availability Date	06/Jun	13	2013	(?)
Price Tier	Free	(?)		

[View Pricing Matrix](#) (?)

Discount for Educational Institutions (?)  
 Custom B2B App (?)

Unless you select **specific stores**, your app will be for sale in all App Stores worldwide.

**Go Back** **Continue**

Home | FAQs | Contact Us | Sign Out  
Copyright © 2013 Apple Inc. All rights reserved. Terms of Service | Privacy Policy

FIGURE 17-19

Click the Continue button and enter the version information in the presented screen.

The version number of your application should follow typical software versioning conventions—for example, 1.0 or 1.0.0—and should be equal to the software version you've set in your Xcode project.

In the Copyright field, enter the current year followed by the name of the person or legal entity that owns the exclusive rights for the application. Select the Primary Category and optionally the Secondary Category for this application. A sample configuration is shown in Figure 17-20.



FIGURE 17-20

Scroll down to the Rating section and tick the appropriate boxes for your application. If you need more information about rating your application, you can click the App Rating Details link.

Under Metadata, enter an application description, which will be displayed in the App Store. Enter keywords that will help users find your application as well as a support URL. Optionally, you can enter a Marketing and a Privacy Policy URL. A sample configuration is shown in Figure 17-21.

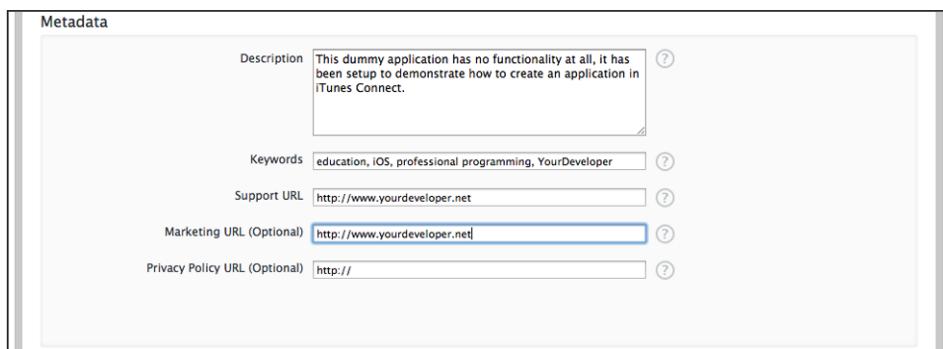


FIGURE 17-21

Under the App Review Information section, enter the contact details of the person who should be contacted in case the App Review team has any questions or needs additional information. The optional Review Notes field enables you to explain, for example, how the App Review team can test the application, and you can also describe its functionality. A sample configuration is shown in Figure 17-22.

Under the Uploads section you can upload the artwork for your application. You can find the artwork dimensions and specifications by clicking the question mark. A sample selection is shown in Figure 17-23, in which I've uploaded some transparent images.

**App Review Information**

**Contact Information** [?](#)

First Name

Last Name

Email Address

Phone Number   
Include your country code

**Review Notes (Optional)** [?](#)

If your application requires for examples some specific application for the review team, you can enter this here.  
For example when your application needs a login with a username and a password you can indicate this here and in the optional section under here  
you can enter the Username and the Password applicable.

**Demo Account Information (Optional)** [?](#)

Username

Password

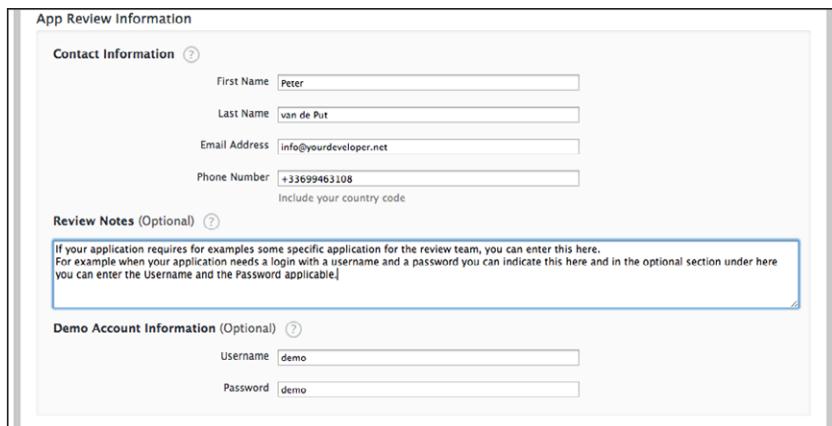


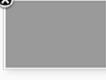
FIGURE 17-22

**Uploads**

**Large App Icon** [?](#)

 [Choose File](#)

**3.5-Inch Retina Display Screenshots** [?](#)

 [Choose File](#)

**4-Inch Retina Display Screenshots** [?](#)

 [Choose File](#)

**iPad Screenshots** [?](#)

[Choose File](#)

**Routing App Coverage File (Optional)** [?](#)

[Choose File](#)

[Go Back](#) [Save](#)

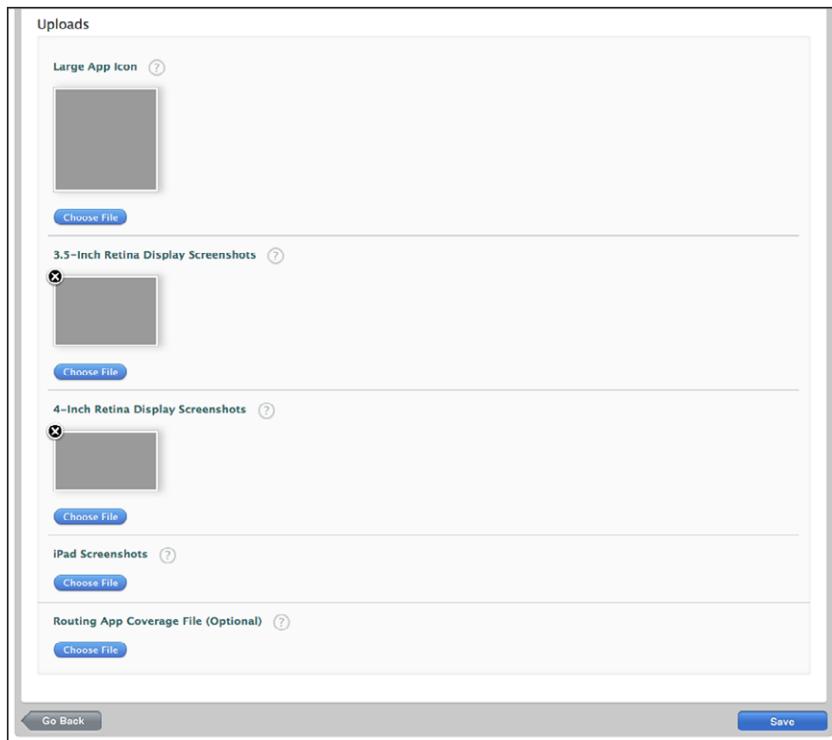


FIGURE 17-23

Finally, click the Save button to create your application, and its status will be Prepare for Upload. The available statuses are explained in Chapter 18.

## Creating a Development Provisioning Profile

Before you can build and distribute your application you need to create a provisioning profile. A development provisioning profile is a collection of digital entities that uniquely relates developer(s) and device(s) to an authorized iOS development team and enables the device(s) for testing. A development provisioning profile must be installed on each device on which you want to run your application code.

Each provisioning profile contains a collection of iOS development certificates, unique device identifiers, and a unique App ID. If a device is not included in the development provisioning profile, it can't be used for testing the application.

A single device can contain multiple development provisioning profiles. In short, this means that a development provisioning profile is a combination of the following:

- The App ID, which is the unique identifier of the application.
- A certificate or multiple certificates that will be used to code sign the application.
- A device or multiple devices on which the application will be installed.

To create a development provisioning profile, navigate to the Certificates, Identifiers and Profiles section in the Member Center and select Development under the Provisioning Profiles node. You will see a list of available iOS provisioning profiles. Click the + button to create a new development provision profile. The Add iOS Provisioning Profile screen appears, in which you select what type of provisioning profile you need. Select iOS App Development as shown in Figure 17-24.

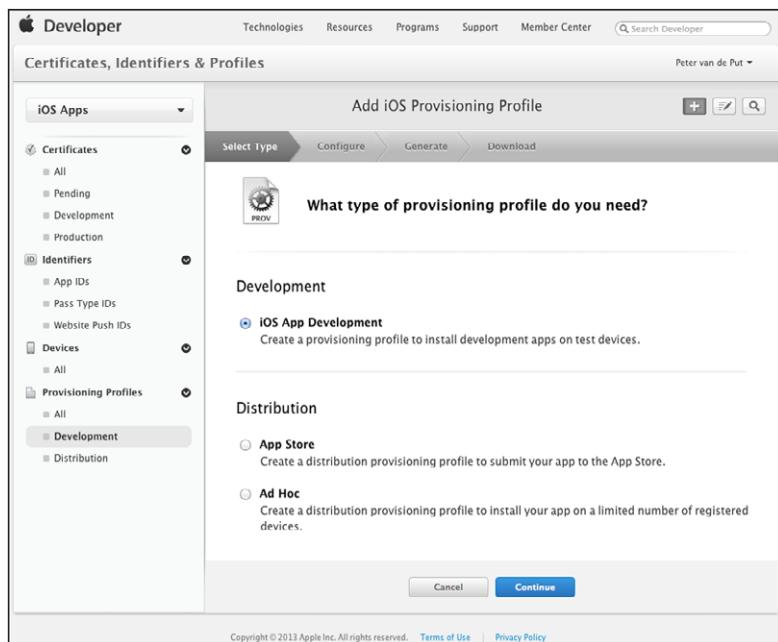
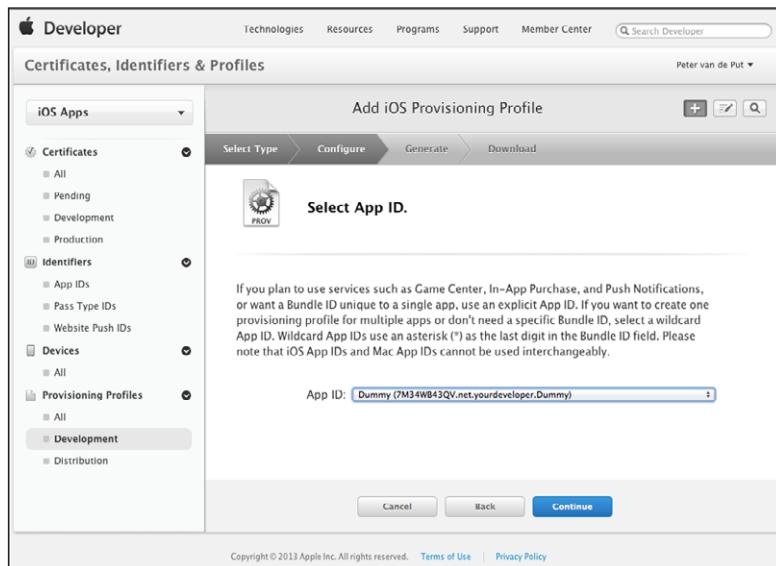


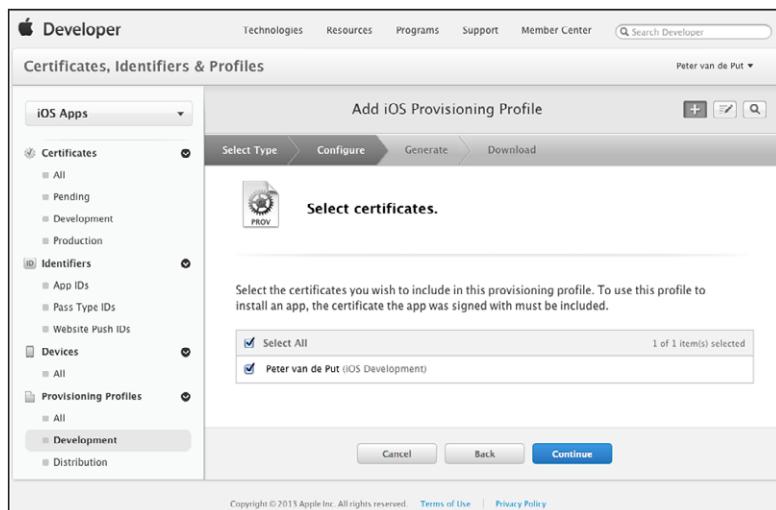
FIGURE 17-24

Click the Continue button, and in the next screen choose the App ID for which you want to create a provisioning profile by selecting it from the drop-down list as shown in Figure 17-25.



**FIGURE 17-25**

Click the Continue button to select the certificate(s) you want to include in this provisioning profile as shown in Figure 17-26.



**FIGURE 17-26**

Click the Continue button to select the device(s) you want to include in this provisioning profile as shown in Figure 17-27.

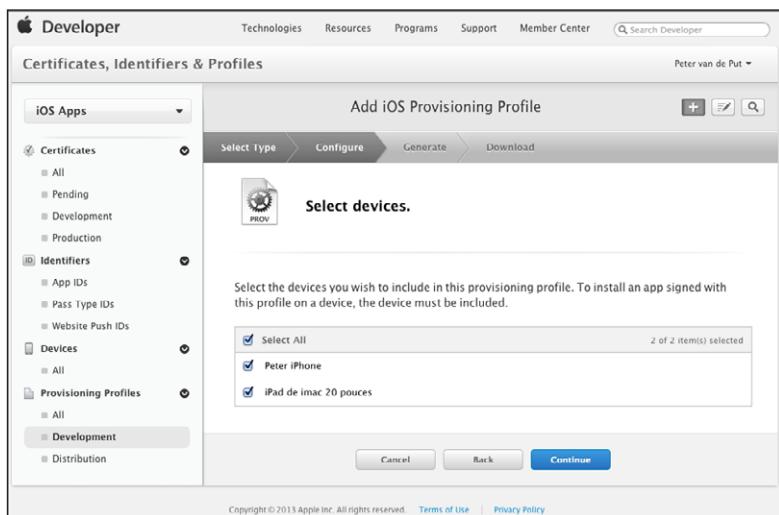


FIGURE 17-27

Click the Continue button to name the provisioning profile and generate it. The Profile Name you enter will be used to identify the profile within the Member Center. Enter a profile name (for example, Dummy\_dev\_profile) as shown in Figure 17-28, and click the Generate button.

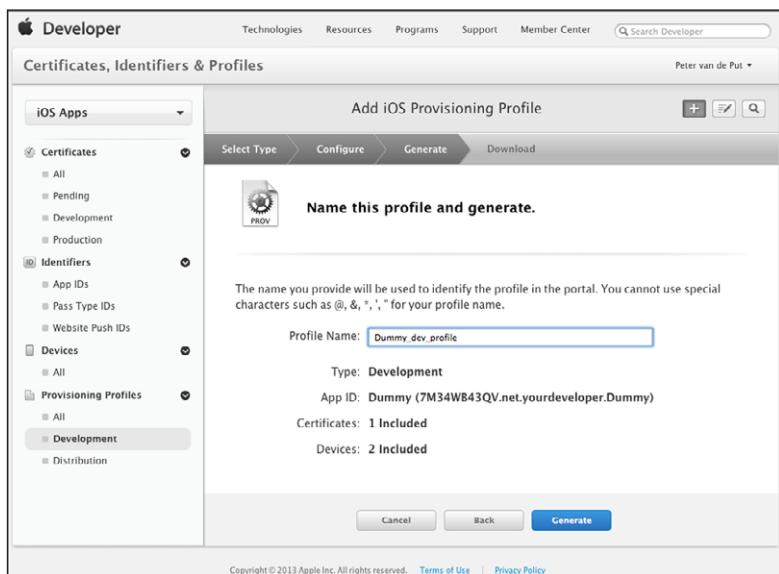
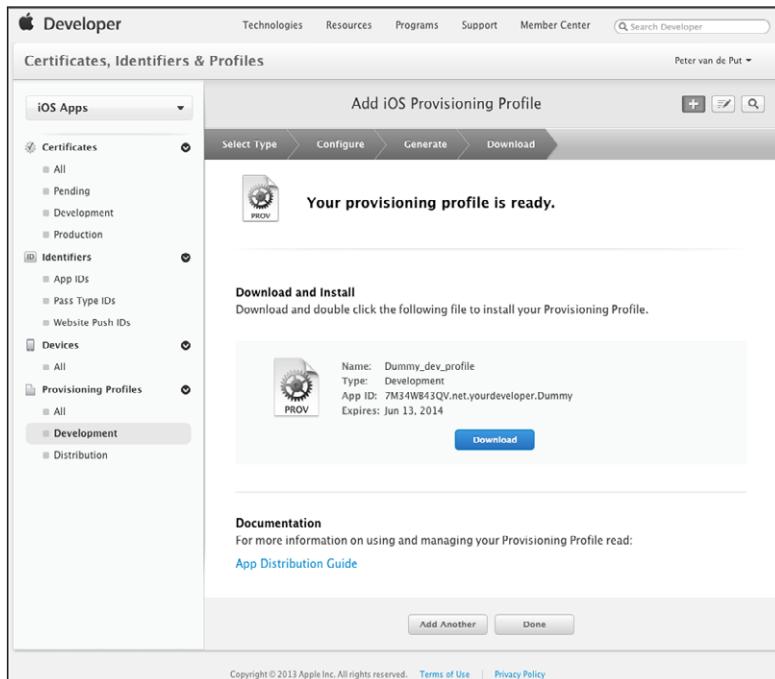


FIGURE 17-28

Once the profile is generated, you are presented with a screen from which you can download the profile to your Mac, as shown in Figure 17-29. Click the Done button once the profile is downloaded.



**FIGURE 17-29**

In Chapter 18 you learn how to use the generated and downloaded development provisioning profile to sign and build your application.

## Creating a Distribution Provisioning Profile

The difference between a distribution provisioning profile and a development provisioning profile is that a distribution provisioning profile enables you to distribute the application either to the App Store or using the Ad Hoc distribution mechanism.

### Creating an Ad Hoc Distribution Provisioning Profile

An Ad Hoc distribution provisioning profile is used to install your application on a limited number of registered devices; for example, to distribute the application to your client's device for testing and evaluation.

The steps you have to follow are almost the same as when creating a development provisioning profile. The only difference is in the selection of the certificate, because now you need to select a production certificate instead of a development certificate.

## Creating an App Store Distribution Provisioning Profile

An App Store distribution provisioning profile is required to sign, build, and distribute your application to the App Store, as you will learn to do in Chapter 18. Creating the App Store distribution provisioning profile requires fewer steps than the Ad Hoc creation of the Ad Hoc distribution provisioning profile. Navigate to the Distribution link under the Provisioning Profiles node, click it, and your distribution profiles will show. Click the + button to bring up the Add iOS Provisioning Profile page. Under the Distribution header, select App Store as shown in Figure 17-30.

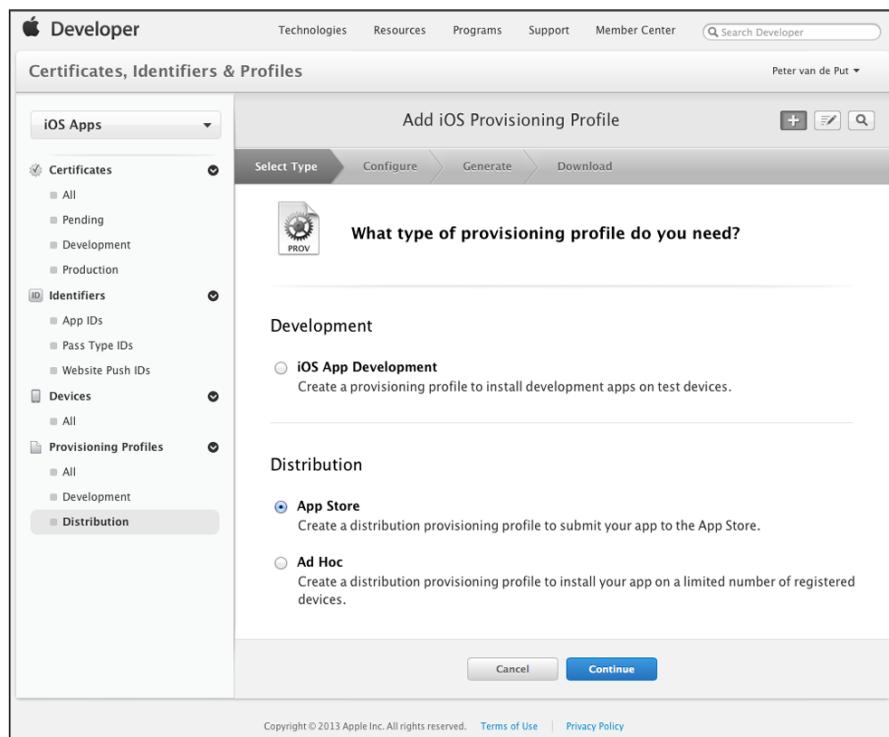


FIGURE 17-30

Click the Continue button, and in the next screen select the App ID as shown in Figure 17-31.

Click the Continue button and select the distribution certification you want to include in the provisioning profile you are about to create, as shown in Figure 17-32.

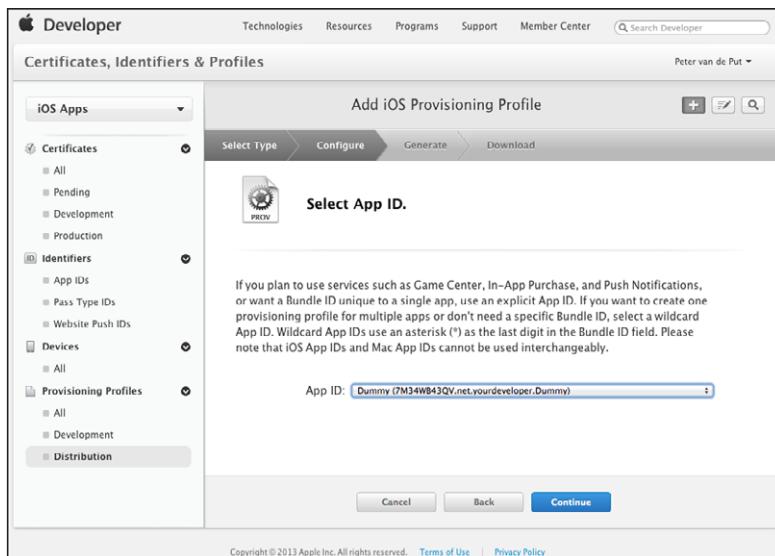


FIGURE 17-31

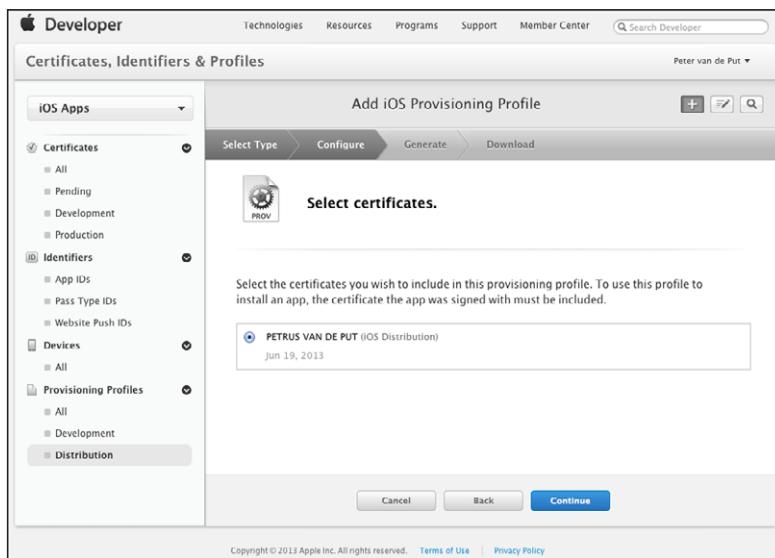


FIGURE 17-32

Click the Continue button, and in the next page enter a profile name and click the Generate button as shown in Figure 17-33.

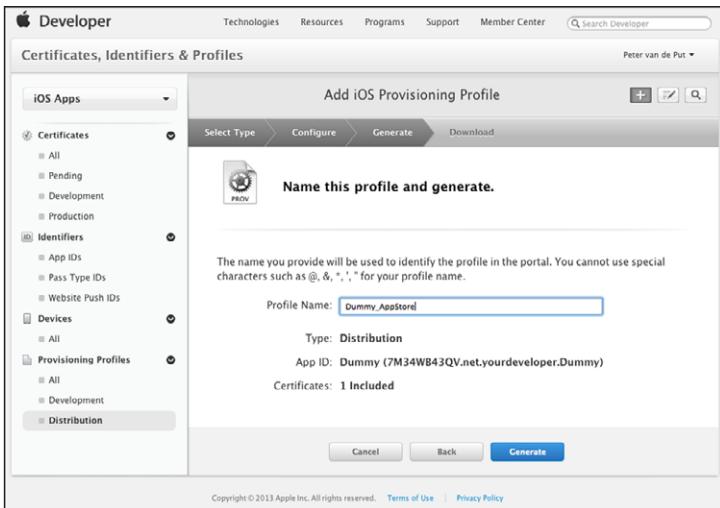


FIGURE 17-33

The App Store distribution provisioning profile is generated, and once it's ready you see the page where you can download and install the profile, as shown in Figure 17-34.

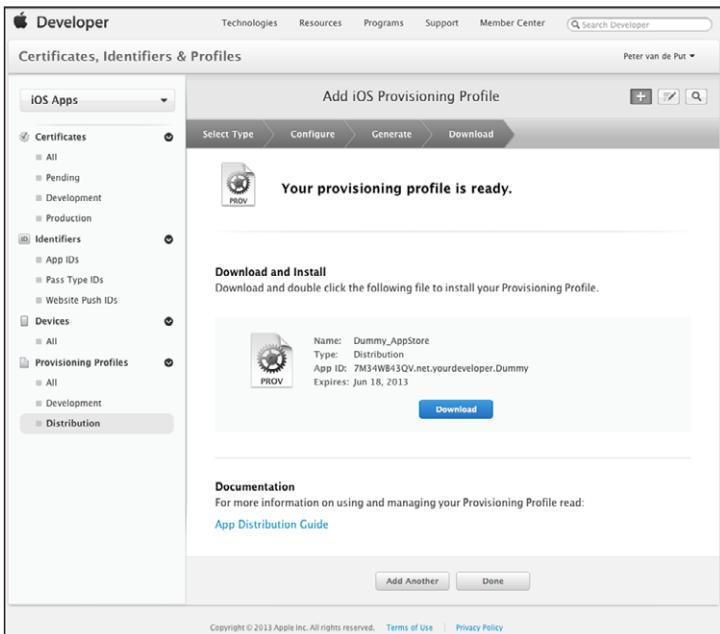


FIGURE 17-34

Click the Download button to download your profile. Once it's downloaded, click the Done button.

## SUMMARY

In this chapter you learned which elements are involved in creating development and distribution provisioning profiles. The developer certificate combined with the application identifier, and optionally combined with a series of device identifiers, provide you with a provisioning profile. You need a valid provisioning profile to be able to either build an application for Ad Hoc distribution or for publishing to the App Store.

In the next chapter you learn how to use these provisioning profiles to sign, build, and distribute your applications.



# 18

## Building and Distribution

### **WHAT'S IN THIS CHAPTER?**

---

- Understanding the App Store Review process
- Building your application for Ad Hoc distribution
- Signing, building, and distributing your application to the App Store

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/proiosprog](http://www.wrox.com/go/proiosprog) on the Download Code tab. The code is in the Chapter 18 download and individually named according to the names throughout the chapter.

### **APP STORE REVIEW**

When you want to publish an application in the App Store, you are bound to the terms of the Program License Agreement, the Human Interface Guidelines, and any other contracts in place between you as a developer and Apple.

You can find the Human Interface Guidelines at <http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/>. This document describes the guidelines and principles to follow when creating a user interface for your iOS application.

When you want to submit your new application to the App Store, you use iTunes Connect to add your application and set up the required metadata. You also configure In-App purchases in iTunes Connect if you haven't done so already during the development process.

## Understanding the Review Guidelines

Apple has defined a document named the *App Store Review Guidelines*, in which it states very clearly which criteria apply to having your application published in the App Store. You can find these guidelines at <https://developer.apple.com/appstore/resources/approval/guidelines.html>.

As an experienced iOS developer, you are probably familiar with these guidelines; you might even have some applications in the App Store already. Keep in mind that this is a living document, so each time you publish an application you should perform a check yourself to see if your application will be accepted in the review process.

Apple starts the review process when you publish an application or an update to the App Store. If your application is rejected, in most cases it's because one or more elements of the Review Guidelines are not respected. The e-mail you receive from Apple will refer to the article in the review guidelines.

## Understanding the Review Process

When you submit your application to the App Store, it enters the queue of applications waiting for review and its status changes to “Waiting for Approval” in iTunes Connect. Recall that if your application requires a username and a password, you need to add test or demo credentials in the Review Notes section with some explanation for the reviewer. Back in 2009, an application review could take weeks, but in 2013 the average application review takes about five working days.

When your application is at the beginning of the queue its status changes to “In Review,” indicating that the review process has started. When the review process is finalized, you are notified with a message indicating the result, and you can see the status of your application using iTunes Connect. Table 18-1 shows which application statuses exist and what they mean.

TABLE 18-1: iTunes Connect Application Statuses

STATUS	DESCRIPTION
Prepare for Upload	This appears as the first status for your app. This status means that you should enter or edit metadata, screenshots, pricing, In-App purchases, Game Center, iAd network settings, and so on, to prepare your app for upload to the App Store.
Waiting for Upload	Once you've completed entering your metadata and indicated that you are ready to submit your binary, but you have not finished uploading your binary through Application Loader the status will indicate “Waiting for Upload”. Your app must be Waiting for Upload for you to be able to deliver your binary through Application Loader.

Status	Description
Waiting for Review	<p>After you submit a new app or update and before the app is reviewed by Apple the status will indicate "Waiting for Review". This status means that your app has been added to the app review queue, but has not yet started the review process. Because it takes time to review binaries, keep in mind that this state does not indicate that your app is currently being reviewed.</p> <p>While your app is waiting for review, you can:</p> <ul style="list-style-type: none"> <li>➤ Reject your binary to remove it from the Apple review queue</li> <li>➤ Edit certain app information</li> </ul>
In Review	<p>When the Apple review team is reviewing your app the status will indicate "In Review".</p>
Pending Contract	<p>When your app has been reviewed and is ready for sale, but your contracts are not yet in effect this will be indicated by the status "Pending Contract". You can check the progress of your contracts in the Contracts, Tax and Banking module.</p>
Waiting for Export Compliance	<p>When your application is using encryption technology and your CCATS file is in review with Export Compliance the status will indicate "Waiting for Export Compliance".</p>
Upload Received	<p>When your binary has been received through Application Loader, but has not yet completed processing into the iTunes Connect system. If your app has been in the Upload Received state for more than 24 hours, you should contact iTunes Connect Support through the iTunes Connect Contact Us module.</p>
Pending Developer Release	<p>Once your app version has been approved by Apple and you have chosen to set your version release control. When you are ready to release your app to the App Store, click the Release This Version button on the app's Version Details page within Manage Your Applications.</p>
Processing for App Store	<p>When your binary is being processed and will be ready for sale within 24 hours this will be indicated with the status "Processing for App Store".</p>
Pending Apple Release	<p>When your app version is set for an iOS or OSX version that is not yet released to the public this will be indicated by the status "Pending Apple Release".</p>

*continues*

**TABLE 18-1** (*continued*)

STATUS	DESCRIPTION
Ready for Sale	Once the binary has been approved and the app is posted to the App Store the status will indicate "Ready for Sale".
Rejected	If the binary has not passed review this will be indicated by the status "Rejected". You receive a communication from Apple in the Resolution Center regarding the reason for the rejection.
Metadata Rejected	If specific metadata items aside from your binary have not passed review it will be indicated by the status "Metadata Rejected". To resolve the issue, edit the metadata in iTunes Connect, and your existing binary is then reused for the review process. You receive a communication from Apple in the Resolution Center regarding the reason for the metadata rejection.
Removed from Sale	If your app has been removed from the App Store it will be indicated by the status "Removed from Sale".
Developer Rejected	If you've rejected the binary from the review process. Choosing the Developer Rejected status removes your app from the review queue. After you resubmit your binary, the app review process starts over from the beginning.
Developer Removed from Sale	This status indicates that you've removed the app from the App Store.
Invalid Binary	This appears when your binary has been received through Application Loader but did not meet all requirements for upload. You receive an e-mail detailing the issue with your binary and showing how to resolve it. To resend the resolved binary, go into iTunes Connect and click Ready to Upload Binary again. This action sets your app back to the Waiting for Upload state so that you can resend the binary through Application Loader.
Missing Screenshot	Available for iOS apps only. When your app is missing a required screenshot for iPhone and iPod Touch or iPad for your default language app or for your added localizations it will be indicated by the status "Missing Screenshot". At least one screenshot is required for both iPhone and iPod Touch, and for iPad if you are submitting a universal app.  Click the number next to the status to view a list of the territories in which a screenshot is missing.

## Understanding Rejections

If your application is rejected you are notified with a message explaining why, and very often it contains information pointing you in the direction of how to resolve the issue.

**WARNING** *The App Review rejection message falls under the Non-Disclosure Agreement you signed with Apple, and therefore you are not allowed to release the contents of a rejection.*

You should follow the instructions in the message. Sometimes you need to make changes to your application and resubmit it to the App Store for review. If you don't agree with the rejection, you can follow the link in the message directing you to the complaint process.

## Avoiding Common Pitfalls

Several publications have listed the top-10 reasons why Apple rejects an application. Although it's impossible to verify these publications because the review process and rejection results fall under the Non-Disclosure Agreement, some of the more common reasons for rejection include the following:

- The application crashes.
- Not respecting trademarks. This includes using words like Apple, iOS, iPhone, and iPad in your application metadata.
- Using the words "beta," "trial," or "preview" in your metadata.
- Long loading time during first launch. Your application should launch within 15 seconds; otherwise, the operating system will kill it and it will be rejected. You should use background queues wisely to ensure the main thread is never blocked and the user interface is working smoothly and quickly after the application is launched.
- Using non-Apple payment systems. All content sold from within an application should be sold using In-App Purchase and be handled through the user's iTunes account.
- Promoting an Android or Windows Mobile version in the application or its metadata. You should promote that on your website.
- Not following the iOS data storage guidelines. With the introduction of iOS 5.1, data that can be regenerated should not be able to be backed up. You can find the iOS data storage guidelines at <https://developer.apple.com/icloud/documentation/data-storage/>.
- The application crashes when a user denies permissions. You've learned that the user is prompted to authorize your application to access the Address Book, the Calendar, Reminders, Bluetooth, Twitter, or Facebook. If the user denies access, your application should still function.

## BUILDING FOR AD HOC DISTRIBUTION

To build your application for Ad Hoc distribution you must have a valid Ad Hoc distribution provisioning profile installed. You learned how to create the Ad Hoc distribution provisioning profile in Chapter 17 and how to download it and install it in your keychain access.

## Building Your Application

Building your application for Ad Hoc distribution starts with code signing. Signing an application allows the operating system to identify who signed the application, and to verify that the application has not been changed since it was signed. When you code sign an application during a build, Xcode seals your application and identifies it as yours based on the provisioning profile.

To build your application for Ad Hoc distribution, follow these steps:

1. Select your project's target, and under Build Settings scroll down to the Code Signing section.
2. Select the Ad Hoc distribution provisioning profile you created earlier, as shown in Figure 18-1.

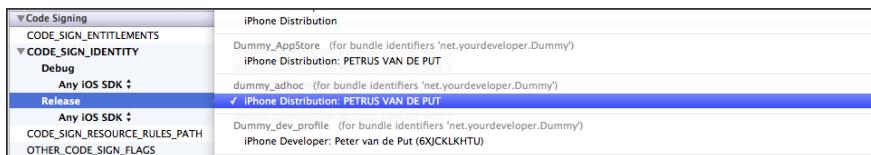


FIGURE 18-1

3. Select iOS Device as the target, as shown in Figure 18-2.

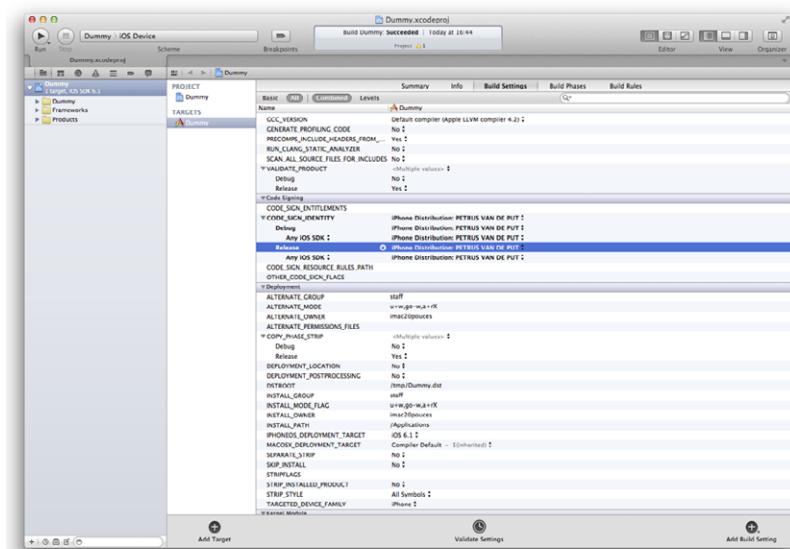


FIGURE 18-2

4. Select Build For ➔ Archiving from the Product menu to build an archive, as shown in Figure 18-3.

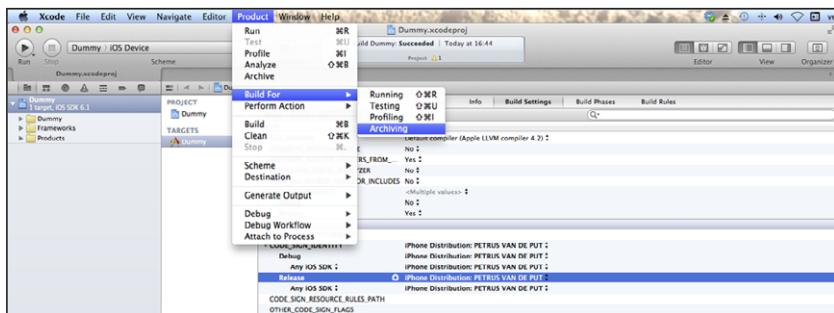


FIGURE 18-3

Xcode will now sign and build your application using the selected Ad Hoc distribution provisioning profile. Depending on the size of your application, this can take from 30 seconds to a few minutes for applications containing a large number of classes and Interface Builder files.

If the compilation was successful, Xcode opens the Organizer window to display the archives, as shown in Figure 18-4.

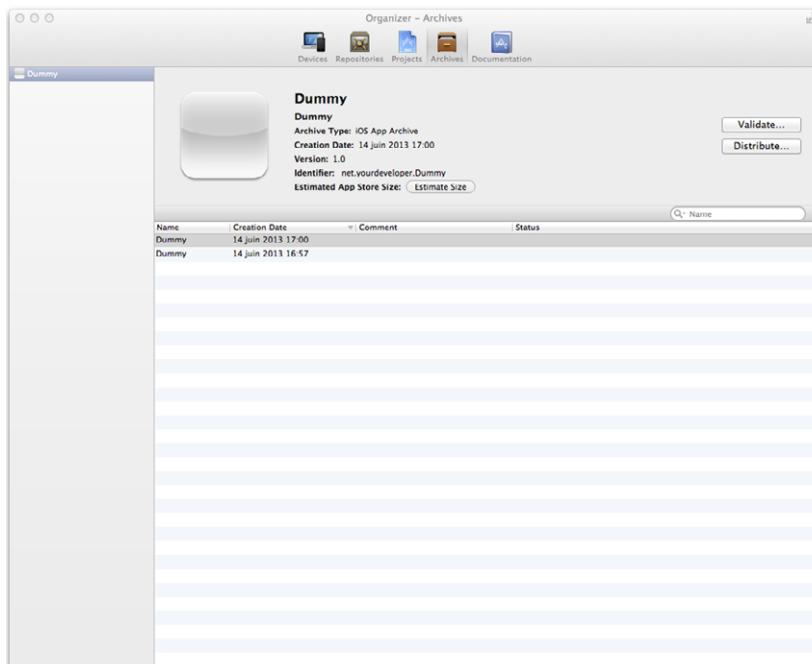


FIGURE 18-4

## Distribute for Testing

Once you have built your application for Ad Hoc distribution, you need to distribute it to your test users. In the Organizer window, select the created archive and click the Distribute button. This opens a new window in which you select the method of distribution. Select Save for Enterprise or Ad-Hoc Deployment, as shown in Figure 18-5.



FIGURE 18-5

Click Next and choose an identity to code sign the compiled application. Select the same Ad Hoc distribution provisioning profile you used to build the archive, as shown in Figure 18-6.

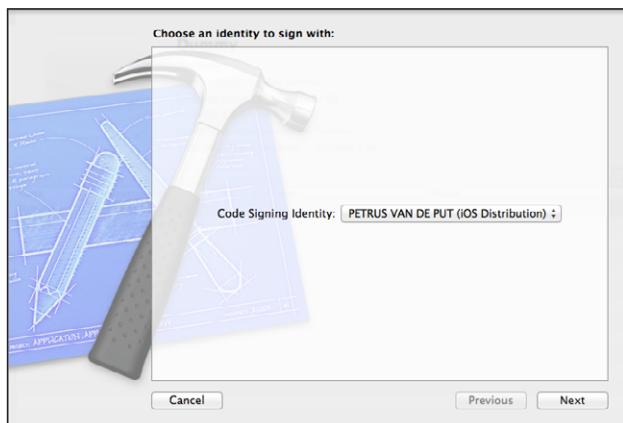
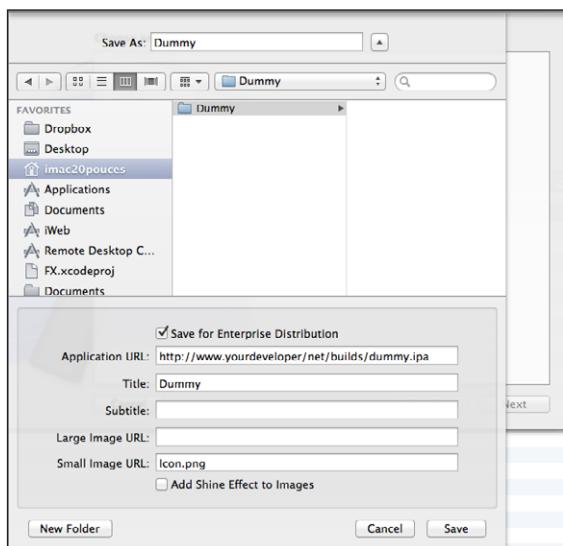


FIGURE 18-6

Click Next to open the window shown in Figure 18-7. Navigate to a folder in which to save the compiled version and select the Save for Enterprise Distribution box. Fill in the fields under this box as follows:

- **Application URL:** The URL where your tester/client can download the version from his iOS device.
- **Title:** Use the same name as the target name.
- **Subtitle:** You can enter what you want here, this subtitle is not used anywhere.
- **Large Image URL:** Icon@2x.png
- **Small Image URL:** Icon.png

Click Save to save the compiled and signed application in the selected folder.



**FIGURE 18-7**

When you use Finder and navigate to the selected folder, you will find two files that have been saved:

- Dummy.ipa
- Dummy.plist

Copy your Icon.png and Icon@2x.png files from your project's folder to the folder where you saved the application.

An HTML file is required to display a specific hyperlink to download the application. This index.html file must contain a link to a specially formatted URL starting with `itms-services://?action=download-manifest&url=`. When clicked on an iOS device, this link opens the manifest file, which is

in the `app.plist`, and verifies if the device's identifier is part of the Ad Hoc distribution provisioning profile. If so, it downloads and installs the application from this location. A sample `index.html` file is shown in Listing 18-1.

**LISTING 18-1:** Chapter18/index.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Ad Hoc application downloads</title>
</head>
<body>
<ul>
<li>
<a href="itms-services://?action=download-
 manifest&url=http://www.yourdomain.com/apps/APP_NAME/app.plist">
 Click here to download and install the APP_NAME application.</a>
</li>
</ul>
</body>
</html>
```

Upload the following files to a server location that is accessible for all your test users with the URL you entered as the Application URL:

- `Icon.png`
- `Icon@2x.png`
- `index.html`
- `App.plist`
- `App.ipa`

If you use an iOS device whose device identifier is part of the Ad Hoc distribution provisioning profile, and navigate to the Application URL, the `index.html` page will be served with the hyperlink to download the application. If you click the link, the application is downloaded and installed on the device and can be used for testing and evaluation purposes.

## BUILDING FOR APP STORE DISTRIBUTION

Signing your application and building it for App Store distribution is very similar to building the application for Ad Hoc distribution.

Navigate to <https://developer.apple.com> and sign in to the Member Center. Click the iTunes Connect link and log in to iTunes Connect. Click the Manage Your Apps link to display your list of applications.

**NOTE** In iTunes Connect the word “binary” is used to refer to the compiled version of your application. When you see a button labeled Ready to Upload Binary, you can read it as “Ready to upload my application to the App Store.”

Click the icon of the application you want to upload to the App Store to display its metadata, as shown in Figure 18-8.

The screenshot shows the iTunes Connect interface for managing an application named "My Dummy Application". The top navigation bar includes links for "View in App Store", "Rights and Pricing", "Transfer App", "Manage In-App Purchases", "Manage Game Center", "Set Up Ad Network", "Newsstand Status", and "Delete Application". The main content area is titled "App Information" and contains sections for "Identifiers" and "Links". Under "Identifiers", details are listed: SKU (MyDumm), Bundle ID (net.yourdeveloper.Dummy), Apple ID (661625624), Type (iOS App), and Default Language (English). The "Links" section includes a "View in App Store" link. Below this is a "Versions" section for the "Current Version" (1.0.0), which shows a status of "Prepare for Upload" and a creation date of June 13, 2013. A "View Details" button is located at the bottom left of this section. At the bottom right is a "Done" button. The footer contains links for Home, FAQs, Contact Us, Sign Out, and legal notices for Terms of Service and Privacy Policy.

FIGURE 18-8

Click the View Details button to reveal the details of your application, as shown in Figure 18-9.

This screenshot shows the "Version Information" section for the "My Dummy Application (1.0.0)" page. It displays the following details: Version 1.0.0, Copyright 2013 YourDeveloper, Primary Category Education, Secondary Category Book, Rating 4+, and Status Developer Rejected. To the right, there is a "Links" sidebar with options for Version Summary, Binary Details, and Status History. A prominent blue button at the top right says "Ready to Upload Binary". The top navigation bar is identical to Figure 18-8, including the "View in App Store" link.

FIGURE 18-9

Your application status is probably Ready for Upload. Click the Ready to Upload Binary button in the top-right corner to inform iTunes Connect that you are about to upload your application. The Export Compliance page appears, in which you have to answer a question about whether your application is using cryptography. Answer the question(s) and click Save to display an information page telling you that you're ready to upload your binary, as shown in Figure 18-10.

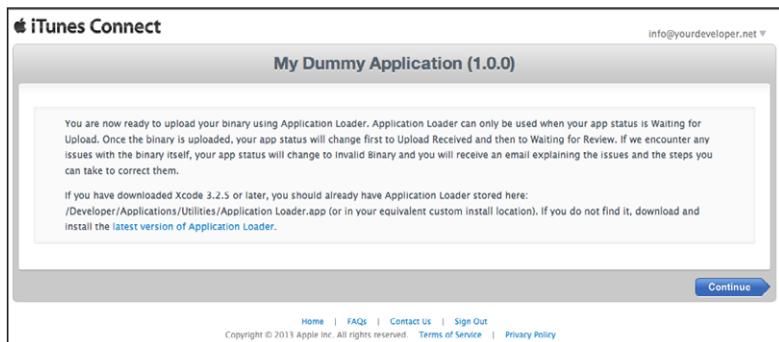


FIGURE 18-10

Click Continue to end this step. You will see your application's detail page and notice that the status has changed to Waiting for Upload, as shown in Figure 18-11.



FIGURE 18-11

Go back to Xcode and select the App Store distribution provisioning profile from the drop-down list, as shown in Figure 18-12.



FIGURE 18-12

Select Build For ➔ Archiving from the Product menu to build an archive. Open the Xcode Organizer and navigate to the archive you just created.

Select the archive and click the Distribute button. The next window prompts you to select the method of distribution. This time select Submit to the iOS App Store as shown in Figure 18-13.



FIGURE 18-13

Click Next, and you are prompted to log in to iTunes Connect. Enter your credentials and click the Next button. You are now prompted to select an Application record and a Code Signing Identity from the presented drop-down lists, as shown in Figure 18-14.

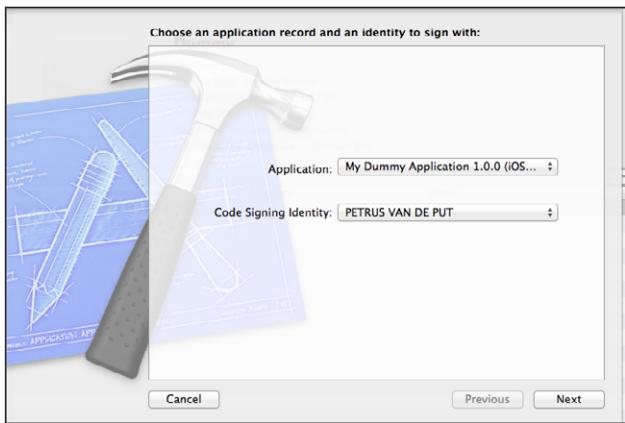


FIGURE 18-14

Make the correct choices and click Next. The application will now be uploaded to the App Store for reviewing, and during this process you will see the screen shown in Figure 18-15.

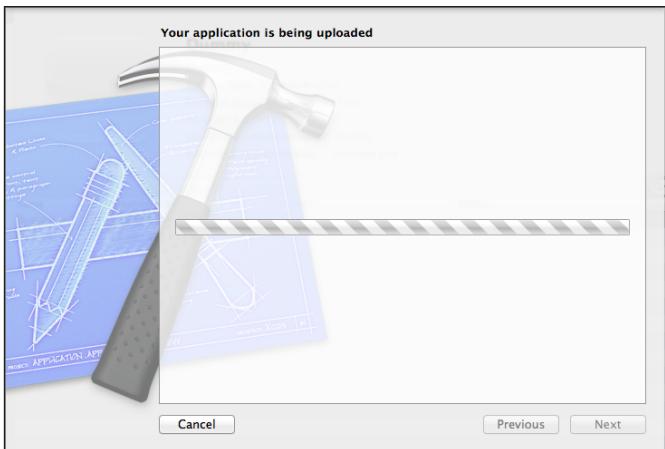


FIGURE 18-15

If you have followed all these steps carefully, the result will be a confirmation that your application has been uploaded, as shown in Figure 18-16.

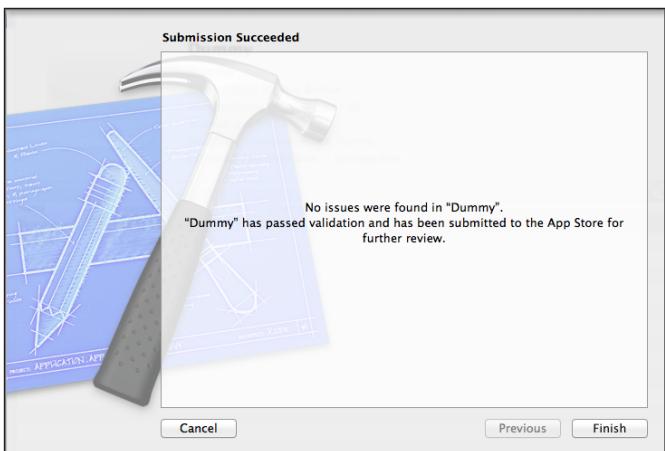
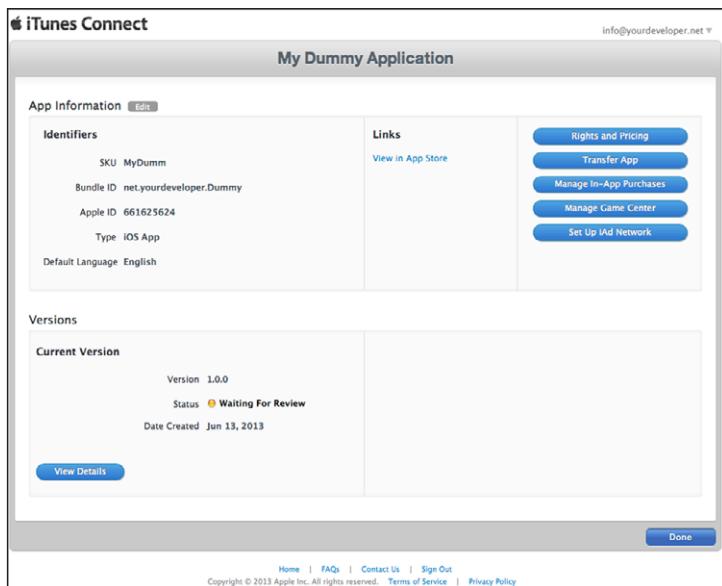


FIGURE 18-16

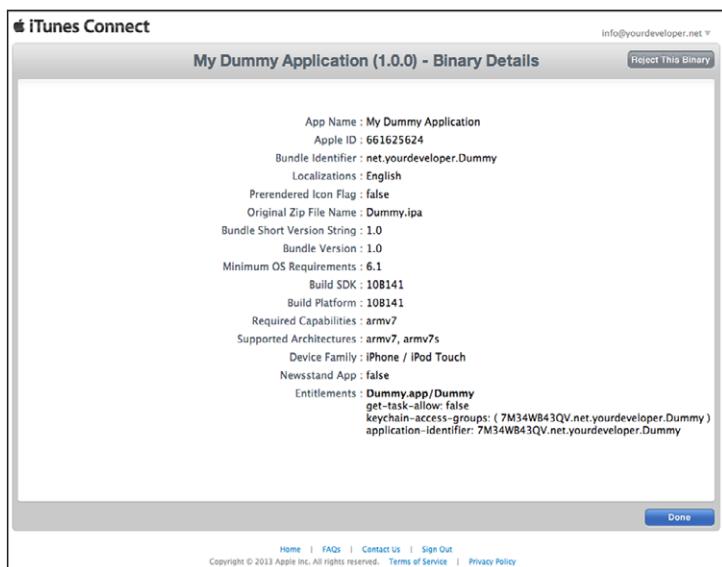
As a confirmation you can check your application's status in iTunes Connect. You will see it has changed to Waiting for Review, as shown in Figure 18-17.



**FIGURE 18-17**

If you find a problem after you've uploaded your application and its status is Waiting for Review, you can take it back.

In the application details screen under the heading links there is a hyperlink named Binary Details. If you click it, your binary details are displayed. In the top-right corner there is a button labeled Reject This Binary, as shown in Figure 18-18.



**FIGURE 18-18**

If you click that button, a confirmation box appears as shown in Figure 18-19.

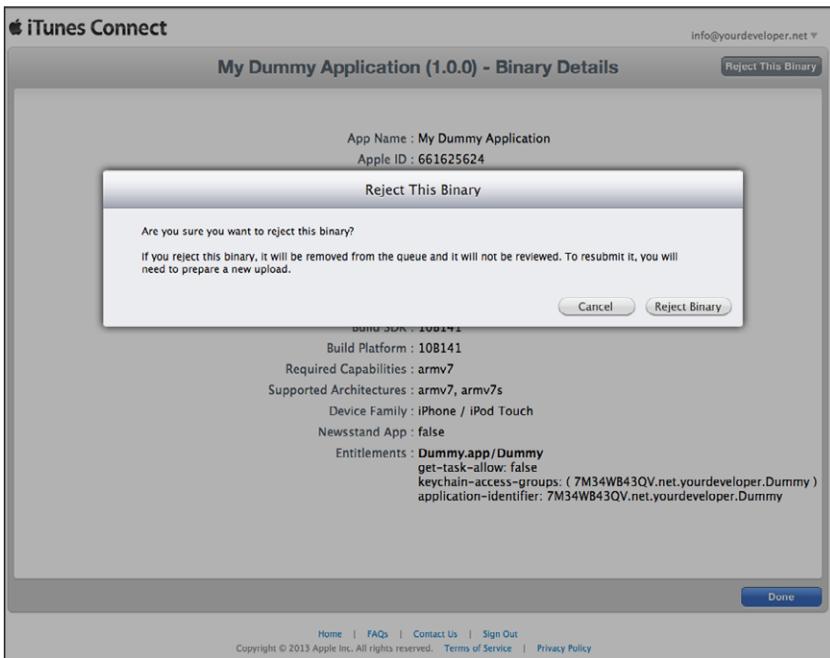


FIGURE 18-19

If you click the Reject Binary button, your upload will be canceled and the status of your application will change to Developer Rejected, as shown in Figure 18-20. The Ready to Upload Binary button appears again, and you'll need to click this and repeat the preceding steps before you can upload a new binary.



FIGURE 18-20

## SUMMARY

In this final chapter you learned how to sign, build, and distribute your applications for both Ad Hoc and App Store distribution.

After having studied all the chapters in this book you should be able to develop professional iOS applications. You have learned how to:

- Use advanced table views
- Implement Map Kit
- Use action sheets and alert views
- Internationalize your application
- Use multimedia such as audio, video and PDF
- Consume Web services and parse responses
- Implement Core Data
- Use internal and external notifications
- Send e-Mail, SMS and dial phone numbers
- Use the Address Book
- Integrate events and reminders in your application
- Integrate your application with social media platforms like Twitter and Facebook
- Perform analysis on your application
- Monetize your application by implementing In-App Purchase functionality and displaying advertisements
- Use iTunes Connect to create and manage provisioning profiles, certificates and devices
- Build and distribute your applications



# A

## Audio Codes

### WHAT'S IN THIS APPENDIX?

---

- ▶ An overview of the available audio codes for predefined sounds

If you want your application to attract the user's attention by playing a simple sound, you can use one of the many built-in system sounds.

To do so, add the AudioToolbox framework to your project and import the AudioToolbox header file into the `UIViewController` where you want to use it.

Now you can create a simple `playSound:` method that accepts an `integer` value called `soundID` as shown in Listing A-1.

---

#### LISTING A-1: The `playSound:` method

```
- (void)playSound: (int)soundID
{
    AudioServicesPlaySystemSound(i);
}
```

Before you can call the `playSound:` method you need to add the AudioToolbox framework to your project and import the AudioToolbox header file.

Table A-1 shows the `soundID` values you can use for each category of sounds.

**TABLE A-1: Sound Categories and their ID Values**

SOUNDID	CATEGORY	SOUNDID	CATEGORY
1070	AudioToneBusy	1032	SMSReceived_Alert
1074	AudioToneCallWaiting	1033	SMSReceived_Alert
1071	AudioToneCongestion	10341035	SMSReceived_Alert
1073	AudioToneError	1036	SMSReceived_Alert
1075	AudioToneKey2	1307	SMSReceived_Selection
1072	AudioTonePathAcknowledge	1308	SMSReceived_Selection
1113	BeginRecording	1309	SMSReceived_Selection
1117	BeginVideoRecording	1310	SMSReceived_Selection
1005	CalendarAlert	1312	SMSReceived_Selection
1108	CameraShutter	1313	SMSReceived_Selection
1106	ConnectedToPower	1314	SMSReceived_Selection
1114	EndRecording	1320	SMSReceived_Selection
1118	EndVideoRecording	1321	SMSReceived_Selection
1102	FailedUnlock	1322	SMSReceived_Selection
1256	Headset_AnswerCall	1323	SMSReceived_Selection
1258	Headset_CallWaitingActions	1324	SMSReceived_Selection
1257	Headset_EndCall	1325	SMSReceived_Selection
1255	Headset_Redial	1326	SMSReceived_Selection
1254	Headset_StartCall	1327	SMSReceived_Selection
1259	Headset_TransitionEnd	1328	SMSReceived_Selection
1306	KeyPressClickPreview	1329	SMSReceived_Selection
1103	KeyPressed	1330	SMSReceived_Selection
1104	KeyPressed	1331	SMSReceived_Selection
1105	KeyPressed	1332	SMSReceived_Selection
1006	LowPower	1333	SMSReceived_Selection
1000	MailReceived	1334	SMSReceived_Selection

SOUNDID	CATEGORY	SOUNDID	CATEGORY
1001	MailSent	1335	SMSReceived_Selection
1057	PINKeyPressed	1336	SMSReceived_Selection
1107	RingerSwitchIndication	1011	SMSReceived_Vibrate
1350	RingerVibeChanged	1311	SMSReceived_Vibrate
1100	ScreenLocked	1004	SMSSent
1101	ScreenUnlocked	1016	SMSSent
1109	ShakeToShuffle	1300	SystemSoundPreview
1351	SilentVibeChanged	1301	SystemSoundPreview
1051	SIMToolkitTone	1302	SystemSoundPreview
1052	SIMToolkitTone	1303	SystemSoundPreview
1053	SIMToolkitTone	1304	SystemSoundPreview
1054	SIMToolkitTone	1305	SystemSoundPreview
1055	SIMToolkitTone	1315	SystemSoundPreview
1003	SMSReceived	1200	TouchTone
1007	SMSReceived_Alert	1201	TouchTone
1008	SMSReceived_Alert	1202	TouchTone
1009	SMSReceived_Alert	1203	TouchTone
1010	SMSReceived_Alert	1204	TouchTone
1011	SMSReceived_Alert	1205	TouchTone
1012	SMSReceived_Alert	1206	TouchTone
1013	SMSReceived_Alert	1207	TouchTone
1014	SMSReceived_Alert	1208	TouchTone
1020	SMSReceived_Alert	1209	TouchTone
1021	SMSReceived_Alert	1210	TouchTone
1022	SMSReceived_Alert	1211	TouchTone
1023	SMSReceived_Alert	1050	USSDAlert

*continues*

**TABLE A-1** (*continued*)

SOUNDID	CATEGORY	SOUNDID	CATEGORY
1024	SMSReceived_Alert	1154	VCCallUpgrade
1025	SMSReceived_Alert	1153	VCCallWaiting
1026	SMSReceived_Alert	1152	VCEnded
1027	SMSReceived_Alert	1150	VCIInvitationAccepted
1028	SMSReceived_Alert	1151	VCRinging
1029	SMSReceived_Alert	4095	Vibrate
1030	SMSReceived_Alert	1002	VoicemailReceived
1031	SMSReceived_Alert		

# B

## Artwork Dimensions

### WHAT'S IN THIS APPENDIX?

---

- An overview of the different screen dimensions of the iOS devices
- An overview of the dimensions of the artwork required to publish your application in the App Store

## DEVICE DIMENSIONS

To make your application look astonishing, you need to use the correct dimensions for the artwork.

### Application Icons

TABLE B-1 Application Icons

TYPE OF DEVICE	DIMENSIONS PIXELS
iPhone Non-Retina	57 x 57
iPhone Retina (< iOS 7.0)	114 x 114
iOS 7 iPhone Retina	120 x 120
iPad Non-Retina	72 x 72
iPad Retina	144 x 144

**TABLE B-2** Spotlight Icons

TYPE OF DEVICE	DIMENSIONS
iPhone Non-Retina	29 x 29
iPhone Retina	58 x 58
iPad Non-Retina	50 x 50
iPad Retina	100 x 100

**TABLE B-3** Settings Icons

TYPE OF DEVICE	DIMENSIONS
iPhone Non-Retina	29 x 29
iPhone Retina	58 x 58
iPad Non-Retina	29 x 29
iPad Retina	58 x 58

## Launch Images

**TABLE B-4** Launch Images

TYPE OF DEVICE	DIMENSIONS
iPhone Non-Retina	320 x 480
iPhone Retina	640 x 960
iPhone Retina 4-inch (iPhone 5)	640 x 1136
iPad Portrait Non-Retina	768 x 1004
iPad Portrait Retina	1536 x 2008
iPad Landscape Non-Retina	1024 x 748
iPad Landscape Retina	2048 x 1496

## iTunes Connect Artwork Dimensions

When you create your application in iTunes Connect before publishing it to the App Store (as you learned in Chapter 18), you need to provide artwork with the correct dimensions. Table B-5 lists the dimensions you can use for various devices.

TABLE B-5 iTunes Screenshots

TYPE OF DEVICE	ALLOWED DIMENSIONS IN PIXELS
iPhone Non-Retina	320 x 480
iPhone Retina	960 x 640 960 x 600 640 x 960 640 x 920
iPhone Retina 4-inch (iPhone 5)	1136 x 640 1136 x 600 640 x 1136 640 x 1096
iPad Non-Retina	1024 x 768 1024 x 748 768 x 1024 768 x 1004
iPad Retina	2048 x 1536 2048 x 1496 1536 x 2048 1536 x 2008

TABLE B-6 iTunes Large App Icon

TYPE OF DEVICE	ALLOWED DIMENSIONS
All	1024 x 1024



# INDEX

## A

ABAddressBookGetAuthorizedStatus function, 377–378  
ABAddressBookHasUnsavedChanges function, 378  
ABAddressBookRef object, 376, 381–382  
ABAddressBookRegisterExternal ChangeCallback function, 379  
ABAddressBookRemoveRecord object, 384  
ABAddressBookRequestAccess WithCompletion function, 377  
ABAddressBookUnregisterExternal ChangeCallback function, 379  
ABNewPersonViewController class, 373–375  
ABPeopleNavigationController class, 365–366  
ABPerson.h file, 365, 380  
ABPersonViewController object, 370–373  
ABRecord objects, 380  
ABRecordRef object, 379–383  
ACAccount class, 404  
ACAccountCredential class, 404  
ACAccountStore class, 404, 406–408, 419, 422–426  
ACAccountType class, 404  
Accounts framework, 404–408  
ACFacebookAppIdKey key, 407, 419  
ACFacebookAudienceKey key, 407, 419  
ACFacebookPermissionsKey key, 407, 419, 422  
action views  
    adding buttons, 122–125  
    creating UIActionSheet objects, 120–122  
    presenting UIActionSheet objects, 125–133  
ActionSheetResponding project, 133–136  
Ad Hoc distributions  
    building applications, 510–511  
creating provisioning profile, 499–500  
testing, 512–514  
Address Book  
    access permission, requesting, 367–370, 377–378  
    accessing programmatically, 375–384  
    components, 364  
    contacts  
        creating, 373–375, 381–384  
        deleting, 375, 384  
        displaying, 370–372  
        editing, 370–372  
        selecting, 364–367  
    programming guide, 363  
    records, 379–381  
    saving/abandoning changes, 378  
    synchronizing, 379  
AdMob platform, 476–479  
advertising platforms, 448, 473  
    AdMob, 476–480  
    iAd, 473–476  
affiliate sales, 449  
AlarmClock project, 336–341  
    creating notifications, 337–340  
    receiving notifications, 340–341  
alarms, adding to reminders, 399–401  
alert views  
    adding objects, 136–140  
    implementing a crash handler, 21–28  
AlertViews project, 136–140  
alphabet index, implementing, 72–76  
analyzing applications  
    commercial analyses, 444–445  
    Flurry Analytics, 445  
    technical analyses, 435–444  
annotations  
    clustering, 100–117

- creating custom annotations, 90–94
  - responding to call-outs, 95–100
  - App Store distributions**
    - App Store Review Guidelines*, 506
    - building applications, 514–520
    - creating provisioning profile, 500–502
    - rejections, 509
    - review process, 506–508
    - submitting applications, 492–495
  - App Store Review Guidelines*, 506
  - Application layer, OSI model, 214
  - applications
    - analyzing
      - commercial analyses, 444–445
      - Flurry Analytics, 445
      - technical analyses, 435–444
    - crash management, 20–28, 436
    - defaults, initializing, 15
    - distributing
      - Ad Hoc distributions, 499–500, 510–514
      - App Store distributions, 500–502, 505–509, 514–520
    - header file, importing, 10
    - icon dimensions, 527–528
    - identifiers, creating, 489–492
    - localizing, 141–151
      - application names, 150–151
      - images, 148–150
    - Interface Builder files, 144–145
    - strings, 145–147
  - monetizing
    - advertising platforms, 448, 473–479
    - affiliate sales, 449
    - commercial analysis, performing, 444–445
    - In-App Purchases. *See* In-App Purchases
    - lead generation, 449
    - paid applications, 448
    - sales statistics, 444–445
    - subscriptions, 448
  - performance-tuning, 324–330
  - single sign-in applications, 426–431
  - submitting, 492–495
- ARC (Automatic Reference Counting), 4–5
- memory leaks and, 437–438
- Urban Airship support, 342
- attachments (e-mail), 358
- audio
  - built-in sounds, 523–526
  - formats supported, 198
  - frameworks, 182
  - playing
    - from application bundle, 182–187
    - from iTunes library, 187–191
    - streaming audio, 191–193
  - recording, 193–198
- Audio Sessions API, 182
- Audio Toolbox framework, 182
- AudioRecorder project, 193–198
- auto-renewable subscriptions, 450
- Automatic Reference Counting. *See* ARC
- AVAudioPlayer class, 182–187
- AVAudioRecorder class, 182, 195–198
- AVAudioSession class, 182
- AVFoundation framework, 182, 184–187

## B

- bandwidth usage, 438–442
- BarButton Item, 128
- BatteryDrainer project, 439–444
- blocking the main thread, 436–437
- blocks, 228–232
- building applications
  - for Ad Hoc distribution, 510–514
  - for App Store distribution, 514–520

## C

- Calendar database, 385
  - access permission, 386–387
  - default calendar, setting, 388–390
  - programmatic access, 391–397
  - synchronizing, 397
- calendars
  - creating events, 390–391, 392–396
  - default, 155–158
  - deleting events, 397
  - editing events, 390–391, 396–397
  - internationalization, 155–159
  - synchronizing with Calendar database, 397

- CalloutAction project, 95–100  
 CD command (FTP), 272  
 Certificate Signing Request (CSR), 347, 483–486  
 certificates, development, creating, 482–486  
 chat view controller, 38–55  
     building  
         chat data object, 39–41  
         chat user object, 49–50  
         custom UITableView, 42–45  
         datasource, 39  
     complete implementation, 50–55  
     custom cells, 46–49  
     flexible cell height, 45–46  
 classes. *See also* specific classes  
     singleton, 81, 454  
     YD prefix, 6  
 CLLocationManager class, 80, 84, 87–88, 442–444  
 CLOSE command (FTP), 272  
 clustering annotations, 100–117  
 ClusterMap project, 101–117  
     YDAnnotationsArray.h file, 109  
     YDAnnotationsArray.m file, 110  
     YDCluster.h file, 111  
     YDCluster.m file, 111–112  
     YDClusterAnnotationView.h file, 108  
     YDClusterAnnotationView.m file, 108–109  
     YDClusterManager.h file, 112  
     YDClusterManager.m file, 113–114  
     YDClusterMapView.h file, 102  
     YDClusterMapView.m file, 102–107  
     YDClusterPin.h file, 107  
     YDClusterPin.m file, 107–108  
     YDViewController.h file, 114–115  
     YDViewController.m file, 115–117  
 comma-separated value (CSV) files, parsing, 260–268  
 commercial analyses, 444–445  
 CommonDigest.h file, 20  
 CompleteAudioPlayer project, 183–187  
 ComplexFTPClient project, 289–296  
 concurrency with Core Data, 331–332  
 ConcurrentThreads project, 331–332  
 constants, defining, 8  
 consumable In-App Purchases, 450  
 contacts  
     creating, 373–375, 381–384  
     deleting, 375, 384  
     displaying, 370–372  
     editing, 370–372  
     selecting, 364–367  
 Continent.h file, 319  
 copyrights, 493–494  
 Core Data framework, 299–300  
     AppDelegate setup, 307–309  
     concurrency with, 331–332  
     fetch requests, 301  
     lightweight migration, 320–322  
     managed object context, 300  
         configuring, 330  
     managed object model, 300–301  
         changes to, 320–324  
         creating, 302–304  
     managed objects, 301  
         creating, 305–306, 310–11  
         deleting, 310  
         fetching, 311–315  
     performance-tuning applications, 324–330  
     persistent stores, 301, 306  
     reasons to use, 300  
     relationships, 316–320  
 CoreDemo project, 302–315.  
     *See also* Core Data framework  
     YDAppDelegate.h file, 307  
     YDAppDelegate.m file, 307–309  
     YDViewController.h file, 311–312  
     YDViewController.m file, 312–314  
 CoreDemo-Prefix.pch file, 302  
 CoreGraphics functions, 174, 178, 180  
 CorePerformance project, 325–330  
 CorePerformance-Prefix.pch file, 325  
 CoreRelationship project, 316–320  
 CoreRelationship-Prefix.pch file, 316  
 crash management, 20–28, 436  
     crash handler, implementing, 21–28  
     reasons for crashes, 21  
     signals, 21  
 CreatePDFDocument project, 177–181  
 CRUD services. *See* REST services

CSR (Certificate Signing Request), 347, 483–486  
CSV (comma-separated value) files, parsing, 260–266  
CustomAnnotation project, 90–94

## D

Data link layer, OSI model, 214  
data storage. *See* Core Data framework  
datasource, UITableView, 30–33  
date formats, 151–159  
  locales, 151–155  
  storing dates, 159  
DateHelper class, 159  
DateHelper.h file, 159  
DateHelper.m file, 159  
dealloc method, 4, 229  
delegate, UITableView, 30–33  
DELETE command (FTP), 272  
Developer Member Center, 482  
  creating  
    development certificate, 482–486  
    development provisioning profile, 496–499  
    distribution provisioning profile, 499–502  
  managing  
    applications, 489–495  
    devices, 486–489  
Developer Rejected status, 508  
Developer Removed from Sale status, 508  
development certificate, creating, 482–486  
development provisioning profiles, 496–499  
device tokens, 341  
devices  
  development provisioning profiles  
    496–499  
  dimensions, 527–529  
  distribution provisioning profiles, 499–502  
  registering, 488–489  
  unique identifier, 486–487  
digital magazines, 448  
distributing applications  
  Ad Hoc distributions  
    building applications, 510–511  
    creating provisioning profile, 499–500  
    testing, 512–514

App Store distributions  
  App Store Review Guidelines JS *ital*, 506  
  building applications, 514–520  
  creating provisioning profile, 500–502  
  rejections, 509  
  review process, 506–508  
distribution provisioning profiles, 499–502  
downloading  
  development certificates, 486  
  images from HTTP URL, 219–225  
  images from HTTPS URL, 225–228  
  remote files, 277–278  
drill-down table view, implementing, 56–65

## E

e-mail  
  attaching files, 358  
  composing, 356–358  
  sending, 355–356  
EKCalendar class, 388–389  
EKCalendarChooser class, 386, 388–390  
EKCalendarChooserDelegate class, 389–390  
EKEventEditViewAction class, 391  
EKEventEditViewActionCancelled constant, 391  
EKEventEditViewActionDeleted constant, 391  
EKEventEditViewActionSaved constant, 391  
EKEventEditViewController class, 390–391  
EKEventEditViewDelegate class, 390  
EKEventStore class, 387–390  
EKEVENTViewController class, 386, 396  
emailer project, 355–358  
en locale identifier, 151  
encryption, MD5, 18–20  
Event Kit framework  
  reference guide, 386  
  reminders, 397–398  
    adding alarms, 399–400  
    creating, 398–399  
    deleting, 399  
    editing, 399  
event store, 385  
  observers, adding, 397

EventKitUI framework, 385–386  
 access permission, requesting, 386–388  
 calendars. *See also* calendars  
   accessing, 388–390  
   creating/editing events, 390–391  
 events, 385  
   calendars  
     accessing, 388–390  
     creating calendar events, 390–391,  
       392–396  
     default, 155–159  
     deleting calendar events, 397  
     editing calendar events, 390–391, 396–397  
     synchronizing with Calendar database, 397  
 reminders, 397–398  
   adding alarms, 399–400  
   creating, 398–399  
   deleting, 399  
   editing, 399  
 external notifications, 352–354

**F**

Facebook integration, 419–426  
   Facebook SDK documentation, 404  
   options configuration, 407, 419  
   posting HTTP-based requests, 412  
   single-sign in applications, 426–431  
 fetch requests, 301  
 fetching managed objects, 311–315  
 FileZilla project, 272  
 Flurry Analytics, 445  
 Flurry.h file, 445  
 FotoDownloader project, 220–225  
 Foundation functions, 145  
 free subscriptions, 451  
 FTP

  commands. *See also* specific commands  
   commonly used, 272  
   relation between methods, 273  
   sequence of sending, 297  
 creating remote directories, 279  
 downloading remote files, 277–278  
 listing remote directories, 280–283  
 ports, 272

reading from an NSStream, 284  
 uploading files, 283–284  
 writing  
   a complex client, 288–296  
   to an NSStream, 285–288  
   a simple client, 272–277  
 FTPManager class, 273–277, 287–288  
 FTPManager.h file, 273–274  
 FTPManager.m file, 274–277  
 FX currency converter, 449

**G**

geofencing, 398, 399–401  
 GET command (FTP), 272, 273, 274, 277  
 GET operation, 215  
 git local repositories, 6  
 Google AdMob platform, 476–479  
 Google Maps, 79

  creating GPS route files with, 84–87

GPS eXchange Format (GPX) files, 80

GPSSimulator project, 80–89.

*See also* Map Kit framework

  coordinates.txt file, 85–87  
   YDLocationSimulator.m file, 82–84  
   YDSimulator.h file, 81–82  
   YDViewController.h file, 87  
   YDViewController.m file, 87–89

GPX (GPS eXchange Format) files, 80

group records, 380

groups, creating for projects, 6–7

Guestbook project, 242–248

  YDFirstViewController.h file, 243  
   YDFirstViewController.m file, 243–245  
   YDSecondViewController.h file, 246  
   YDSecondViewController.m file, 246–248

**H**

Hardware Services API, 182  
 HEAD operation, 215  
 header files, importing, 10  
 HTTP services, 216–232  
   downloading images, 219–225  
   requesting websites, 216–219

- secure websites, 225–228
  - synchronized requests, 438
  - using blocks, 228–232
  - HTTPS protocol, 215, 225–228
  - HTTPSPictureloader project, 225–228
- I**
- iAd framework, 473–476
  - images
    - downloading
      - from HTTP URL, 219–225
      - from HTTPS URL, 225–228
    - localizing, 148–150
  - In Review status, 507
  - In-App Purchases, 448
    - choosing product type, 450–451
    - implementing, 452–473
      - buying products, 467
      - creating products in iTunes Connect, 453–459
      - delivering products, 471–473
      - presenting the store, 463–467
      - processing transaction receipts, 469
      - receiving product information, 462–463
      - requesting product information, 462
      - retrieving product list, 459–461
      - sending payment requests, 468–469
      - verifying transaction receipts, 469–471
    - process steps, 451–452
    - registering products, 450
  - index path, 34
  - IndexedTable project, 72–76
  - instant messaging, chat view controller, 38–55
    - building
      - chat data object, 39–41
      - chat user object, 49–50
      - custom UITableView, 42–45
      - datasource, 39
      - complete implementation, 50–55
      - custom cells, 46–49
      - flexible cell height, 45–46
  - Instruments, 21
    - Instruments User Guide, 168
    - profiling projects, 168–170
  - instruments, 21
  - InteractiveAB project, 370–372
  - Interface Builder, 5
    - localizing files, 144–145
  - InternationalApp project, 142–151
    - localizing
      - application names, 150–151
      - images, 148–150
      - Interface Builder files, 144–145
      - strings, 145–147
      - setting up localization, 143–144
  - InternationalDate project
    - DateHelper.h file, 159
    - DateHelper.m file, 159
    - YDViewController.m file, 152–155, 156–158
  - internationalization
    - calendars, 155–159
    - date formats, 151–159
    - locales, 151–155
    - storing dates, 159
  - localizing applications, 141–151
    - application names, 150–151
    - images, 150
    - Interface Builder files, 144–145
    - strings, 145–147
    - number formats, 160–164
  - Invalid Binary status, 508
  - invalid product identifiers, 462
  - iOS Human Interface Guidelines, 165, 444, 505
  - iOS Keychain Services, 11–12
  - iPod library. *See* iTunes library
  - iTunes Connect
    - application statuses, 506–508
    - artwork dimensions, 529
    - auto-renewable subscriptions, 450
    - creating products in, 453–459
    - invalid product identifiers, 462
    - non-renewing subscriptions, 451
    - registering products, 450
    - sales statistics, 444–445
    - submitting applications, 492–495
  - iTunes library
    - audio from, playing, 187–191
    - video from, playing, 202–205
  - iTunesVideo project, 202–205

**J**

JSON (JavaScript Object Notation), 232  
 parsing responses, 233–236  
 JSONCars project, 232–235

**K**

KeychainItemWrapper class, 11–12, 20  
 KeychainItemWrapper.h file, 11  
 KeychainItemWrapper.m file, 11–12, 20  
 keychains  
   securing passwords, 18–20  
   storing passwords, 20  
 KissXML, 260

**L**

languages  
   adding support for, 143–144  
   editing localized .xib files, 144–145  
   localizing strings, 145–148  
 layers, OSI model, 214  
 LCD command (FTP), 272  
 lead generation, 449  
 LIST command (FTP), 273, 274, 277, 280  
 local git repositories, 6  
 local notifications, 336–341  
   characteristics, 336  
   creating, 337–340  
   receiving, 340–341  
 locales, 151–155  
 localizing applications, 141–151  
   application names, 150–151  
   images, 148–150  
   Interface Builder files, 144–145  
   strings, 145–147  
 login  
   logic, creating, 16–18  
   passwords  
     securing, 18–20  
     storing, 20

**M**

main thread, blocking, 436–437  
 managed object context, 300  
   configuring, 330  
 managed object model, 300–301  
   changes to, 320–324  
   creating, 302–304  
 managed objects, 301  
   creating, 305–306, 310  
   deleting, 310  
   fetching, 311–315  
 Map Kit framework, 79–80  
   annotations, 90  
     clustering, 100–117  
     creating custom annotations, 90–94  
     responding to call-outs, 95–100  
   documentation, 80  
   GPS simulator  
     (*See also GPSSimulator project*)  
     creating, 80–84  
     GPS route files, creating with Google Maps, 84–87  
     implementing, 87–89  
     need for, 80  
   mapping models, 323–324  
   MD5 encryption, 18–20  
   Media Player framework, 182  
     audio, playing, 187–191  
     video, playing, 199–201  
   memory leaks, 21, 437–438  
   MessageUI framework, 356, 359  
   Metadata Rejected status, 508  
   Missing Screenshot status, 508  
   MKAnnotation class, 91–94, 107  
   MKAnnotationView class, 93–94, 101–107  
   MKD command (FTP), 272, 273, 274  
   MKDIR command (FTP), 272  
   MKMapPoint class, 109–114  
   MKMapView class, 79–80  
     annotations, 90  
       clustering, 100–117  
       creating custom annotations, 90–94  
       responding to call-outs, 95–100

GPS simulator  
    creating, 80–84  
    implementing, 87–89

MKMapViewDelegate class, 92, 102

MKUserLocation class, 93

monetizing applications  
    advertising platforms, 448, 473  
        AdMob, 476–480  
        iAd, 473–476  
    affiliate sales, 449  
    commercial analysis, 444–445  
    In-App Purchases, 448  
        choosing product type, 450–451  
        implementing, 452–473  
        process steps, 451–452  
        registering products, 450  
    lead generation, 449  
    paid applications, 448  
    sales statistics, 444–445  
    subscriptions, 448  
        auto-renewable, 450  
        free, 451  
        non-renewing, 451

MPUT command (FTP), 272

multimedia  
    audio  
        playing from application bundle, 182–187  
        playing from iTunes library, 187–191  
        playing streaming audio, 191–193  
        recording, 193–198  
    PDF documents, 165  
        creating, 177–181  
        creating thumbnails from, 173–177  
        displaying with QuickLook, 170–173  
        displaying with UIWebView, 166–170  
    video  
        playing from application bundle, 199–201  
        playing from iTunes library, 202–205  
        playing YouTube videos, 205–207  
        recording, 207–209

MultipleChoice project, 122–125

MyAccounts project, 404–408

MyAddressBook project, 364–370  
    requesting access permissions, 367–370  
    selecting contacts, 364–367

MyCalDB project, 392–397

MyEvents project, 386–391  
    accessing calendars, 388–390  
    creating calendar events, 390–391  
    editing calendar events, 390–391  
    requesting access permission, 386–388

MyFacebook project, 421–426

MyiAdDemo project, 474–476

MyMappingModel mapping model, 324

MyPersonalLibrary project.  
    *See also* Personal Library  
        MyPersonalLibrary-Prefix.pch file, 10  
        NSString+MD5.h file, 19  
        NSString+MD5.m file, 20  
        YDAppDelegate.h file, 25  
        YDAppDelegate.m file, 26–28  
        YDConfigurationHelper.h file, 8–9  
        YDConfigurationHelper.m file, 9–10  
        YDConstants.h file, 8  
        YDCrashHandler.h file, 22  
        YDCrashHandler.m file, 22–25  
        YDLoginViewController.h file, 16  
        YDLoginViewController.m file, 16–18  
        YDRegistrationViewController.h file, 13  
        YDRegistrationViewController.m file, 14–15

MyPersonalLibrary-Prefix.pch file, 10

MyReminders project, 397–401

MyStore project, 452–473  
    initializing the YDInAppPurchaseManager, 463–464  
    presenting the store, 463  
    processing transaction receipts, 469

products  
    buying, 467  
    creating in iTunes Connect, 453–459  
    delivering, 471–473  
    displaying, 464–467  
    receiving detail information, 462–463  
    requesting detail information, 462  
    retrieving list, 459–461  
    sending payment requests, 468–469  
    verifying transaction receipts, 469–471

MyTunesPlayer project, 188–191

MyTwitter project, 408–409

**N**

Network layer, OSI model, 214  
 networking  
     operations, 215  
     OSI model, 214  
     protocols, 214–215  
     response codes, 215  
 NewContact project, 373–375  
 Newsstand, 448, 451  
 non-consumable In-App Purchases, 450  
 non-renewing subscriptions, 451  
 notifications  
     external notifications, 352–354  
     local notifications, 336–341  
     push notifications, 341–352  
 NSArray+CSV.h file, 262  
 NSArray+CSVr.m file, 263–264  
 NSAttributeDescription object, 300  
 NSCalendar object, 155–159  
 NSColor object, 301  
 NSDate object, 151–155  
 NSDateFormatter class, 156–159  
 NSDictionary object, 266–270  
 NSEntityDescription object, 300  
 NSFetchedResultsController object, 311–313  
 NSFetchedRequest object, 312, 313, 314–315  
 NSIndexPath object, 34–37  
 NSLocale object, 151–155  
 NSManagedObject object, 301, 305–306  
 NSManagedObjectModel object, 300  
 NSMutableArray object, 312  
 NSNumberFormatter object, 160–164  
 NSPredicate object, 301  
 NSRelationshipDescription object, 300  
 NSRunLoop object, 275, 283, 291–292  
 NSSetUncaughtExceptionHandler object, 22  
 NSSortDescriptor object, 313  
 NSString+base64.h file, 456–457  
 NSString+base64.m file, 457–458  
 NSString+MD5.h file, 19  
 NSString+MD5.m file, 20  
 NSURLConnection object, 216  
     and blocking, 228–232  
     downloading images, 219–225

requesting a website, 216–219  
 synchronized HTTP requests, 438  
 NSURLRequest object, 216  
     requesting websites, 216–219  
     synchronized HTTP requests, 438  
 NSUserDefaults class, 8–10, 18  
 number formats, 160–164  
 NumberFormats project, 160–164  
     YDViewController.h file, 160  
     YDViewController.m file, 161–164

**O**

objects  
     adding to alert views, 136–140  
     managed objects, 301  
         creating, 305–306, 310  
         deleting, 310  
         fetching, 311–315  
         releasing, 21  
 OPEN command (FTP), 272  
 operations, 215  
 OSI (Open Systems Interconnection) model, 214  
 over-releasing objects, 21

**P**

paid applications, 448  
 parsing  
     CSV files, 260–268  
     XML responses, 237–241  
 passwords  
     securing, 18–20  
     storing, 20  
 PayPal, 448  
 PDF documents  
     creating, 177–181  
     displaying  
         with QuickLook, 170–173  
         with UIWebView, 166–170  
     thumbnails from, creating, 173–177  
 PDFCreateThumbnail project, 173–177  
 Pending Apple Release status, 507  
 Pending Contract status, 507

- Pending Developer Release status, 507  
performance-tuning applications, 324–330  
persistent stores, 301, 306  
person records, 379–380  
`Person.h` file, 305  
`Person.m` file, 305–306  
Personal Library, 4.  
    *See also* MyPersonalLibrary project  
    crash management, 20–28  
    creating, 4–11  
        Automatic Reference Counting *vs.*  
            Interface Builder, 4–5  
        configuration settings, 5–6  
        constants, defining, 8  
        groups, creating, 6–7  
        header file, importing, 10  
        using the configuration, 8–10  
    functionalities, 4  
    registration/login, 11–20  
        application defaults, initializing, 15–16  
        iOS Keychain Services, 11–12  
        login logic, creating, 16–18  
        passwords, securing, 18–20  
        passwords, storing, 20  
        registration logic, creating, 12–15  
phone numbers, dialing, 360–361  
Physical layer, OSI model, 214  
PlainTable project, 30–33  
ports, FTP, 272  
POST operation, 215  
Prepare for Upload status, 506  
PresentActionSheet project, 120–122  
Presentation layer, OSI model, 214  
PresidentSearch project, 66–71  
Processing for App Store status, 507  
product identifiers, 450, 462  
profiling projects, 168–170  
ProgAB project, 376–384  
    address books, 376–379  
    creating contacts, 381–384  
    deleting contacts, 384  
    properties, 380–381  
    records, 379–389  
projects. *See also* specific projects  
    ARC (Automatic Reference Counting), 4–5  
    groups, configuring, 6–7  
    Interface Builder, 5  
    profiling, 168–170  
    starting, 5–6  
    templates, 3  
properties, Address Book records, 380–381  
protocols, 214–215  
provisioning profiles  
    development, 496–499  
    distribution, 499–502  
push notifications, 341–352  
    configuring developer portal, 343–346  
    elements of, 341  
    flow of, 341  
    implementing with Urban Airship, 349–352  
    obtaining certificates, 346–349  
PushDemo project, 342–352  
PUT command (FTP), 272, 273, 274
- Q**
- QLPDFViewer project, 170–173  
QLPreviewItem object, 171–172  
Quartz2D programming guide, 174  
QuickLook framework, 170–179
- R**
- Reachability application, 439–444  
`Reachability.h` file, 439  
`Reachability.m` file, 439  
Ready for Sale status, 508  
recording  
    audio, 193–198  
    video, 207–209  
records (Address Book), 379–381  
redmine, 436  
registering  
    devices, 488–489  
    products with In-App Purchases, 450  
users  
    application defaults, initializing, 15  
    iOS Keychain Services, 11–12  
    registration logic, creating, 12–15  
rejected applications, 509

- Rejected status, 508  
 relationships, Core Data framework, 316–320  
 releasing objects, 21  
 reminders, 397–398  
     adding alarms, 399–400  
     creating, 398–399  
     deleting, 399  
     editing, 399  
 Removed from Sale status, 508  
 Representational State Transfer. *See* REST services  
 response codes, 215  
 REST services, 232–248  
     constructing requests, 232–235  
     posting to, 242–248  
     processing responses, 236–241  
 RSSReader project, 237–241
- S**
- sales statistics, 444–445  
 scrolling, in UITableView, 34–37  
*ScrollingTable* project, 34–37  
 search, enabling on UITableView, 66–71  
*SecureSOAP* project, 258–260  
 security  
     securing passwords, 18–20  
     storing passwords, 20  
 Session layer, OSI model, 214  
*showFromBarButtonItem* method,  
     UIActionSheet class, 128  
*showFromRect* method, UIActionSheet  
     class, 128–130  
*ShowFromRect* project, 128–130  
*showFromTabBar* method, UIActionSheet  
     class, 125–127  
*showFromToolbar* method, UIActionSheet  
     class, 131–133  
*ShowFromToolbar* project, 131–133  
*showInView* method, UIActionSheet  
     class, 125  
 SIGABRT signal, 21  
 SIGFPE signal, 21  
 SIGILL signal, 21  
 SIGINT signal, 21
- signing applications  
     for Ad Hoc distribution, 510–513  
     for App Store distribution, 514–520  
 SIGSEGV signal, 21  
 SIGTERM signal, 21  
 Simple Object Access Protocol. *See* SOAP services  
*SimpleFTPCClient* project, 272–288  
     creating remote directories, 279  
     downloading remote files, 277–278  
     FTPManager.h file, 274  
     FTPManager.m file, 274–277  
     listing remote directories, 280–283  
     reading from NSStream, 284  
     uploading files, 283–284  
     writing FTP clients, 272–277  
     writing to NSStream, 285–288  
*SimpleHTTPCall* project, 216–219  
 Sina Weibo, 403, 407, 412, 417  
 single sign-in applications, 426–431  
 singleton classes, 81, 454  
 SKPayment class, 468  
 SKPaymentQueue class, 468–469  
 SKProduct class, 465, 467  
 SKProductRequest class, 459, 462  
 SKProductResponse class, 462  
 SLComposeViewController class, 408–412  
 SLRequest class, 409, 412–417  
 SMS, sending, 359–360  
 SMTP accounts, 358  
 SOAP services, 248–260  
     passing values to operations, 252–257  
     preparing requests, 250–252  
     secure requests, 258–260  
*SoapCalculator* project, 252–257  
*SoapRequest* project, 250–252  
 soapservice.asmx file, 248  
 Social framework, 408–426  
 social media integration  
     Accounts framework, 404–408  
     Facebook, 419–426  
         Facebook SDK documentation, 404  
         options configuration, 407, 419  
         posting HTTP-based requests, 412  
     single sign-on applications, 426–431

- Twitter  
posting tweets, 409–417  
retrieving tweets, 418–419
- sso project, 427–431
- Store Kit framework, 449. *See also* In-App Purchases  
non-consumable product restoration, 450  
non-renewing subscription synchronization, 451  
transaction receipt verification, 469
- storing  
dates, 159  
passwords, 20
- streaming audio, 191–193
- StreamingAudioAndVideo project, 191–193
- strings, localizing, 145–147
- submitting applications, 492–495
- subscriptions, 448  
auto-renewable, 450  
free, 451  
non-renewing subscriptions, 451
- SVN (subversion) systems, 6, 436
- synchronized HTTP requests, 438
- synchronizing  
Address Book, 379  
Calendar database, 397
- System Services API, 182
- T**
- table views. *See* UITableView
- TBXML, 260
- technical analyses, 435–444  
application crashes, 436  
bandwidth usage, 438–442  
battery drainage, 442–444  
blocking main thread, 436–437  
memory leaks, 437–438  
synchronized HTTP requests, 438  
user interface, 444
- tel:// URL identifier, 360
- telprompt:// URL identifier, 360
- testing Ad Hoc distributions, 512–514
- text messages  
composing, 359–360  
verifying SMS availability, 359
- thread confinement, 331–332
- thumbnails, creating from PDF documents, 173–177
- Transport layer, OSI model, 214
- Twitter  
posting tweets  
SLComposeViewController class, 409–412  
SLRequest class, 412–417  
retrieving tweets, 418–419  
single sign-in applications, 426–431
- TwitterPostRequest project, 412–417
- U**
- UIActionSheet object  
adding buttons, 122–125  
creating, 120–122  
presenting to users, 125–133  
showFromBarButtonItem method, 128  
showFromRect method, 128–130  
showFromTabBar method, 125–127  
showFromToolbar method, 131–133  
showInView method, 125  
responding to user input, 133–136
- UIAlertView object, 119  
adding a UITextField, 136–140  
implementing a crash handler, 21–28
- UIImage object  
localizing images, 148–150  
memory leaks and, 21
- UIImageView object  
localizing images, 148–150  
memory leaks and, 21
- UILocalNotification object, 337
- UINavigationController object, 131–133
- UISearchBar object, 66–71
- UITabBarController object, 125–127
- UITableView, 29  
alphabet index, implementing, 72–76  
custom instances  
chat view controller, 38–55  
drill-down logic, 56–65  
datasource, 30–33  
delegate, 30–33  
scrolling in, 34–37  
search function, enabling, 66–71

UITextField object, 136–140  
 UIToolbar object, 131–133  
 UIViewController class  
   user login process, 16–18  
   user registration process, 12–15  
 UIWebView object  
   memory leaks and, 21  
   PDF documents, displaying, 166–170  
 unixTimeStamp, 159  
 Upload Received status, 507  
 Urban Airship, 341  
   adding frameworks/libraries, 342–343  
   ARC support, 342  
   creating accounts, 349  
   downloading the SDK, 350  
   enhancing AppDelegate, 351–352  
   implementing push notifications, 349–352  
   implementing the SDK, 350  
 URL schemes, defining, 352–353  
 URLRequest.m file, 229–230  
 UrlSchemeDemo project, 352–354  
 user input  
   asking for, 119–133  
   creating multiple-option UIActionSheet, 120–125  
   presenting the UIActionSheet, 125–133  
   responding to, 133–136  
 user login  
   login logic, creating, 16–18  
   passwords  
     securing, 18–20  
     storing, 20  
 user registration  
   application defaults, initializing, 15  
   iOS Keychain Services, 11–12  
   registration logic, creating, 12–15

**V**

versioning data models, 322–323  
 video  
   playing  
     from application bundle, 199–201

from iTunes library, 202–205  
 YouTube videos, 205–207  
 recording, 207–209  
 VideoPlayer project, 199–201  
 VideoRecorder project, 207–209

**W**

Waiting for Export Compliance status, 507  
 Waiting for Review status, 507  
 Waiting for Upload status, 506  
 web services, 216  
   HTTP services, 216–232  
     downloading images, 219–225  
     requesting secure websites, 225–228  
     requesting websites, 216–219  
     using blocks, 228–232  
   REST services, 232–248  
     constructing requests, 232–235  
     posting to, 242–248  
     processing responses, 236–241  
   SOAP services, 248–260  
     passing values to operations, 252–257  
     preparing requests, 250–252  
     secure requests, 258–260  
 WebPDFViewer project, 166–170  
 WebsiteUsingBlocks project, 228–232

**X**

Xcode  
   Interface Builder, 5  
   project templates, 3  
 XML responses, parsing, 237–241, 260, 266–269  
 XMLToDict project, 266–269

**Y**

YD class prefix, 6  
 YDAnnotationsArray class, 109–110  
 YDAnnotationsArray.h file, 109  
 YDAnnotationsArray.m file, 110  
 YAppDelegate.h file implementations  
   BatteryDrainer, 439–440, 443

CoreDemo, 307  
CorePerformance, 325–326  
InteractiveAB, 370  
MyCalDB, 392  
MyFacebook, 421  
MyPersonalLibrary, 25  
MyTwitter, 409  
PushDemo, 351–352  
TwitterPostRequest, 413  
**YDAppDelegate.m** file implementations  
    BatteryDrainer, 440–441, 443–444  
    CoreDemo, 307–309  
    CorePerformance, 326–328  
    MyCalDB, 392  
    MyPersonalLibrary, 25–28  
    UrlSchemeDemo, 353–354  
    YouTubePlayer, 205  
**YDCar.h** file, 57  
**YDCar.m** file, 58  
**YDChatApp** project, 38–55  
    chat data object, building, 39–41  
    chat user object, creating, 49–50  
    complete implementation, 50–55  
    custom cells, developing, 46–49  
    custom datasource, building, 39  
    custom UITableView, building, 42–45  
        flexible cell height, returning, 45–46  
**YDChartData** class, 39–41, 46  
**YDChartData.h** file, 39  
**YDChartData.m** file, 40–41  
**YDChatTableView** class, 42–45  
**YDChatTableView.h** file, 42  
**YDChatTableView.m** file, 42–45  
**YDChatTableViewCell** class, 47–49  
**YDChatTableViewCell.h** file, 47  
**YDChatTableViewCell.m** file, 48–49  
**YDChatTableViewDataSource** class, 39  
**YDChatTableViewDataSource.h** file, 39  
**YDChatTableViewHeaderCell** class, 46–49  
**YDChatTableViewHeaderCell.h** file, 46  
**YDChatTableViewHeaderCell.m** file, 46–47  
**YDChatUser** class, 49–50  
**YDChatUser.h** file, 50  
**YDChatUser.m** file, 50  
**YDCluster** class, 110–112  
**YDCluster.h** file, 111  
**YDCluster.m** file, 111–112  
**YDClusterAnnotationView** class, 108–109, 115–118  
**YDClusterAnnotationView.h** file, 108  
**YDClusterAnnotationView.m** file, 108–109  
**YDClusterManager** class, 102, 107, 112–114  
**YDClusterManager.h** file, 112  
**YDClusterManager.m** file, 113–114  
**YDClusterMapView.h** file, 102  
**YDClusterMapView.m** file, 102–107  
**YDClusterPin** class, 107–108, 112–115  
**YDClusterPin.h** file, 107  
**YDClusterPin.m** file, 107–108  
**YDConfigurationHelper** class, 8–10, 15  
**YDConfigurationHelper.h** file, 8–9  
**YDConfigurationHelper.m** file, 9–10  
**YDConstants.h** file, 7–10  
**YDCrashHandler** class, 21–28  
**YDCrashHandler.h** file, 22  
**YDCrashHandler.m** file, 22–25  
**YDCSVParsing** project, 261–266  
**YDDetailViewController** class, 95–100  
**YDDetailViewController.h** file, 96  
**YDDetailViewController.m** file, 96–97  
**YDDrillDown** project, 57–65  
    **YDCar.h** file, 57  
    **YDCar.m** file, 58  
    **YDSectionHeaderView.h** file, 58–59  
    **YDSectionHeaderView.m** file, 59–61  
    **YDViewController.h** file, 61  
    **YDViewController.m** file, 61–65  
**YDFirstViewController.m** file  
    Guestbook implementation, 243–245  
    InternationalApp implementation, 145  
    TabbedActionSheet implementation, 126–127  
**YDFTPClient** class, 289–297  
**YDFTPClient.h** file, 289  
**YDFTPClient.m** file, 290–296  
**YDInAppPurchaseManager** class, 453–473.  
    *See also* In-App Purchases  
**YDInAppPurchaseManager.h** file, 454–455

- YDInAppPurchaseManager.m file, 454–455  
YDLocationSimulator.h file, 81–82  
YDLocationSimulator.m file, 82–84  
YDLocationsSimulator class, 80–89  
YDLoginViewController class, 16–18  
YDLoginViewController.h file, 16  
YDLoginViewController.m file, 16–18  
YDLoginViewController.xib file, 18  
YDMainViewController class, 26  
YDPresident class, 66–71  
YDPresident.h file, 66  
YDPresident.m file, 67  
YDRegistrationViewController class, 12–15  
YDRegistrationViewController.h file, 13  
YDRegistrationViewController.m file, 13–15  
YDRegistrationViewController.xib file, 12–13  
YDRestaurantAnnotation class, 91–94  
YDRestaurantAnnotation.h file, 91  
YDRestaurantAnnotation.m file, 91–92  
YDSecondViewController.h file  
    Guestbook implementation, 246  
    InternationalApp implementation, 148  
YDSecondViewController.m file  
    Guestbook implementation, 246–248  
    InternationalApp implementation, 149–150  
YDSecondViewController.xib file, 148, 245  
YDSectionHeaderView class, 58–65  
YDSectionHeaderView.h file, 58–59  
YDSectionHeaderView.m file, 59–61  
YDViewController.h file implementations  
    AlarmClock, 337  
    AudioRecorder, 195  
    BatteryDrainer, 441–442  
    ClusterMap, 114–115  
    CompleteAudioPlayer, 184–185  
    CoreDemo, 311  
    CreatePDFDocument, 178  
    CustomAnnotation, 92–93  
    FotoDownloader, 221  
    GPSimulator, 87–88  
    IndexedTable, 73  
    iTunesVideo, 203  
    JSONCars, 233  
    MyAccounts, 405  
    MyCalDB, 393  
    MyEvents, 389  
    MyFacebook, 422  
    MyStore, 463  
    MyTunesPlayer, 188–189  
    NSNumberFormat, 160–161  
    PDFCreateThumbnail, 174  
    PlainTable, 31  
    PresidentSearch, 67  
    QLPDFViewer, 170–171  
    RSSReader, 238  
    ScrollingTable, 35  
    SecureSOAP, 259  
    ShowFromToolbar, 131  
    SimpleFTPCClient, 287  
    SimpleHTTPCall, 218  
    SoapCalculator, 254  
    SoapRequest, 251  
    SSO, 428  
    StreamingAudioAndVideo, 192  
    VideoPlayer, 200  
    VideoRecorder, 208–209  
    WebsiteUsingBlocks, 231  
    YDCSVParsing, 264  
    YDDrillDown, 61
- YDViewController.m file implementations
- ActionSheetResponding, 134–136
  - AlarmClock, 338–340
  - AlertViews, 136–140
  - AudioRecorder, 195–198
  - BatteryDrainer, 441–442
  - CalloutAction, 98–100
  - ClusterMap, 115–118
  - CompleteAudioPlayer, 185–187
  - CreatePDFDocument, 178–181
  - CustomAnnotation, 92–94
  - GPSSimulator, 87–89
  - HTTPSPictureDownloader, 225–228
  - IndexedTable, 73–76
  - InteractiveAB, 371–373
  - InternationalDate, 152–155
  - iTunesVideo, 203–204
  - JSONCars, 233–235
  - MultipleChoice, 123–124

MyAccounts, 406–408  
MyAddressBook, 365–370  
MyCalDB, 393–396  
MyEvent, 389–390  
MyFacebook, 422–425  
MyiAdDemo, 475–476  
MyStore, 463  
MyTunesPlayer, 189–191  
MyTwitter, 409–412  
NewContact, 374  
NumberFormats, 161–164  
PDFCreateThumbnail, 174–177  
PlainTable, 31–33  
PresentActionSheet, 121  
PresidentSearch, 67–71  
QLPDFViewer, 171–172  
RSSReader, 238–241  
ScrollingTable, 35–37  
SecureSOAP, 260  
ShowFromRect, 129–130  
ShowFromToolbar, 131–133  
SimpleFTPCClient, 287–288  
SoapCalculator, 254–257  
SoapRequest, 251–252  
SSO, 428–431  
StreamingAudioAndVideo, 192–193  
TwitterPostRequest, 413–417  
VideoPlayer, 200–201  
VideoRecorder, 208–209  
WebsiteUsingBlocks, 231–232  
YDChatApp, 50–55  
YDCSVParsing, 265–266  
YouTubePlayer, 206–207  
**YDViewController.xib file**, 6  
    AlarmClock, 336–337  
    AudioRecorder, 194  
    BatteryDrainer, 440–441  
ClusterMap, 117  
CompleteAudioPlayer, 184  
CreatePDFDocument, 177  
FotoDownloader, 220  
GPSSimulator, 87  
iTunesVideo, 202  
JSONCars, 233  
MultipleChoice, 123  
MyAccounts, 405  
MyAddressBook, 365  
MyCalDB, 392–393  
MyEvents, 388  
MyFacebook, 421  
MyStore, 452–453  
MyTunesPlayer, 188  
NewContact, 373–374  
PlainTable, 30  
PresentActionSheet, 120  
RSSReader, 237  
ScrollingTable, 34  
SecureSOAP, 258  
SoapCalculator, 253  
SoapRequest, 250  
SSO, 427  
StreamingAudioAndVideo, 192  
VideoPlayer, 199  
WebsiteUsingBlocks, 230–231  
YouTubePlayer, 205  
**YDViewControllerViewController.m file**,  
    61–65  
**YDXMLReader.h file**, 266  
**YDXMLReader.m file**, 267–269  
YouTubePlayer project, 205–207

## Z

Zombie instrument, 21



# Try Safari Books Online FREE for 15 days and take 15% off for up to 6 Months\*

Gain unlimited subscription access to thousands of books and videos.



With Safari Books Online, learn without limits from thousands of technology, digital media and professional development books and videos from hundreds of leading publishers. With a monthly or annual unlimited access subscription, you get:

- Anytime, anywhere mobile access with Safari To Go apps for iPad, iPhone and Android
- Hundreds of expert-led instructional videos on today's hottest topics
- Sample code to help accelerate a wide variety of software projects
- Robust organizing features including favorites, highlights, tags, notes, mash-ups and more
- Rough Cuts pre-published manuscripts

**START YOUR FREE TRIAL TODAY!**

Visit: [www.safaribooksonline.com/wrox](http://www.safaribooksonline.com/wrox)

\*Discount applies to new Safari Library subscribers only and is valid for the first 6 consecutive monthly billing cycles. Safari Library is not available in all countries.



An Imprint of  **WILEY**  
Now you know.