



Learn by doing: less theory, more results

iPhone Location Aware Apps by Example

Build five complete location-enabled apps from
scratch—from idea to implementation!

Beginner's Guide

Zeeshan Chawdhary

[PACKT]
PUBLISHING

www.allitebooks.com

iPhone Location Aware Apps by Example

Beginner's Guide

Build five complete location-enabled apps from scratch—from idea to implementation!

Zeeshan Chawdhary

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

iPhone Location Aware Apps by Example

Beginner's Guide

Copyright © 2012 Packt Publishing

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

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

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

First published: March 2012

Production Reference: 1160312

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

ISBN 978-1-84969-224-3

www.packtpub.com

Cover Image by Asher Wishkerman (a.wishkerman@mpic.de)

Credits

Author

Zeeshan Chawdhary

Project Coordinator

Leena Purkait

Reviewers

Sebastian Borggrewe

Taylor Jasko

Robb Lewis

Jose Luis Manners

Shuxuan Nie

Martin Selva

Alex Zaltsman

Proofreader

Mario Cecere

Copy Editor

Leonard D'silva

Indexer

Monica Ajmera

Acquisition Editor

Alina Lewis

Graphics

Manu Joseph

Lead Technical Editor

Susmita Panda

Production Coordinator

Shantanu Zagade

Technical Editors

Lubna Shaikh

LLewellyn Rosario

Cover Work

Shantanu Zagade

About the Author

Zeeshan Chawdhary has been a keen developer for the last six years, and has worked in the location-based space for the past five years. He is currently the Chief Technology Officer of Wcities Inc, a San Francisco-based Location Content Provider. He is currently experimenting with PostGIS, PhoneGap, and iOS 5, and is currently trying his hand at blogging again at <http://justgeeks.in>.

I would like to thank all the lovely people at PacktPub, especially Mary Nadar for having introduced me to the PacktPub family.

A special thanks to Leena, Susmita, Lubna, and Llewellyn for working with me tirelessly on the book.

I would also thank Christopher D. Sloop from WeatherBug, Lauren Sperber and Janine Iamunno from AOL Patch.com, Tim Breidigan, and Robert Martindale from Eventful.com – for having allowed me to use their respective APIs in the book, you guys rock man!

About the Reviewers

Sebastian Borggrewe, born and raised in Germany, is a computer science Master's student at the University of Edinburgh/RWTH Aachen. Since he was 16, he has been freelancing for several web and mobile agencies, and has founded his own agency.

Currently, he is the co-founder and CTO of Loyalli Ltd., a London-based startup, developing mobile loyalty card solutions (<http://www.loyalli.com>).

When he is not coding, he is searching for new technology or ways to use technology to make life even more fun. You will probably find him hanging out with friends, hitting the gym, cooking, or playing the guitar in his 60's band.

One of his goals in life, apart from working in a kick-ass office in central London with an "office slide", is obtaining a pilot license.

More information about Sebastian, and his projects can be found at <http://www.sebastianborggrewe.de>.

Taylor Jasko has been fascinated with technology ever since the day he laid his hands on a Windows 95-based computer. Since then, now being eighteen years old, he has dived into web design and development, computer programming, and even system administration with his favorite server-oriented operating system, Debian Linux.

He founded the technology blog Tech Cores (<http://techcores.com>), and has been working on it ever since it was created back in late 2008. Tech Cores is a great example of his work; he designed and created it using the powerful WordPress content management system, and with the help of his Wacom Intuos4 graphic tablet plus Adobe Photoshop.

While in school, he can be found freelancing graphic design and programming work. His technical strengths include PHP, JavaScript (including libraries such as jQuery), AJAX, HTML, CSS, Perl, Objective-C, Linux/UNIX, MySQL, Apache, Nginx, and to finish it all off, a dab of Python. Essentially, he is a programmer, system admin, and a designer!

He can be reached at taylor@taylorjasko.com.

Jose Luis Manners is a seasoned IT professional with over 20 years of software development experience, including project management, technical architecture, and full life-cycle systems development. His software development experience includes enterprise systems for Fortune 500 clients as well as federal, state, and local governments. Mr. Manners has been recognized on several occasions by Microsoft with their Most Valuable Professional award for his outstanding work with .NET, and his contributions to Microsoft's product teams. He specializes in .NET, iOS, and Android mobile development for clients in the Washington, DC metropolitan area.

He can be reached at jose@josemanners.com.

Shuxuan Nie is a SOA Consultant, specializing in SOA and Java technologies. She has more than 10 years of experience in the IT industry that includes SOA technologies, such as BPEL, ESB, SOAP, XML, and Enterprise Java technologies, Eclipse plugins, and other areas, such as C++ cross-platform development.

Since 2010, Shuxuan has been working in Rubiconred, and has been helping customers resolve integration issues.

From 2007 to 2010, Shuxuan had been working in Oracle Global Customer Support team, and focussed on helping customers solve their middleware/SOA integration problems.

Before joining Oracle, Shuxuan had been working in IBM China Software Development Lab for four years as staff software engineer, where she participated in several complex products on IBM Lotus Workplace, Webshpere and Eclipse platform, and then joined the Australia Bureau of Meteorology Research Center, which is responsible for implementation of Automated Thunderstorm Interactive Forecast System for Aviation and Defence.

Shuxuan holds an MS in the Computer Science degree from Beijing University of Aeronautics and Astronautics.

Martin Selva heads the Gaming team at Hungama Digital Media Entertainment Pvt Ltd, with over nine years of experience. His passion, experience, and expertise have also helped him develop a keen interest in Product Development for Online & Devices.

At Hungama, Martin is responsible for building a gaming portal called www.thegamebox.com, and heads a Gaming Studio that comprises of iOS Developers, PHP Developers, Game Designers, and Content Writers.

He can be reached at martin@hungama.com and martinselva@gmail.com.

Alex Zaltsman is the CEO and founder of Xcela Mobile, a software and mobile cloud infrastructure hosting company that develops applications for mobile devices, such as the iPhone, iPad, and Android. Prior to finding TourSpot, Alex was a co-founder and managing partner of a technology services company. Alex has been in the technology field for over 15 years. Prior to founding the technology services company, Mr. Zaltsman worked at Lucent Technologies, AT&T Labs, and Johnson & Johnson, in technical and management capacities. Alex is also on the Board of Advisors of BizWorld, a non-profit organization that has created curriculum for teaching entrepreneurship and money management to kids. Alex is on the Board of Directors for the New Jersey chapter for Entrepreneurs Organization (<http://www.eonetwork.org>).

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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



<http://PacktLib.PacktPub.com>

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: The Location-based World	7
Understanding Location-based Services	8
Time for action – consuming Location-based Services with Google	9
Buzzwords in the Location-based Industry	11
Application of LBS and common use cases	13
Military	13
Government	13
Commercial	13
How Apple uses LBS in the iPhone, iPad, and iPod devices	14
iOS location API	15
Time for action – turning off Location Tracking in your iPhone	15
Behind LBS – GPS	17
User segment	17
Space segment	18
Control segment	18
Push and Pull methods of Location Services	18
Push Service	18
Pull Service	19
Life without GPS: Wi-Fi-based location detection	19
Life without GPS: cell ID positioning and cell tower triangulation	21
Time for action – using the SkyHook Wireless Loki framework to determine your location	22
Life without GPS: Google Maps API	23
Understanding Indoor and Outdoor Navigation	26
Summary	27

Chapter 2: The Xcoder's World	29
Introducing Xcode 4	29
Xcode 4: Prerequisites and features	30
Prerequisites	30
Features	30
iOS 5 and Xcode 4.2: new and notable features	31
iOS 5 new features	32
Xcode 4.2's new features	33
Transitioning from Xcode3: What you need to know	34
Time for action – installation	36
Time for action – Hello Location	38
Tools for the overnight coders: HTML5	46
PhoneGap	48
Time for action – Using PhoneGap to build a Hello Location App	49
Time for action – using Titanium Appcelerator for building the Hello Location app	54
Time for action – Hello Location with Sencha Touch	59
Exploring location-based SDKs/APIs	63
Foursquare	63
Gowalla	64
Eventful and Last.fm API – some music is always good	65
Still more tools: SimpleGeo and Factual	66
Other Notable APIs – YQL and Location Labs	66
Summary	67
Chapter 3: Using Location in your iOS Apps—Core Location	69
Core Location framework – an overview	70
Time for action – location debugging	70
Core Location Services	72
Standard location	72
Significant change	73
Region monitoring	73
Geocoding and reverse Geocoding – CLGeocoder	74
Direction using heading	75
Core Location Manager – CLLocationManager	76
Time for action – checking for location service availability	77
User authorization	80
Time for action – using Core Location with user authorization	81
The CLLocation object	84
Time for action – receiving location updates in your application	85
Time for action – boundary monitoring with Location Manager	87
Extending Hello Location for nearby events	90
Important things to know before we begin	91

Time for action – extending Hello Location for nearby events	91
Time for action – Last.fm API in your app	97
What just happened?	98
Extending Hello Location for local search	99
Important things to know before we begin	100
Time for action – building a local search app with foursquare	100
Summary	103
Chapter 4: Using Maps in your iOS apps—MapKit	105
Overview of the MapKit framework	105
Understanding map geometry	106
Time for action – using MapKit in your app	108
Time for action – using map gestures – panning and zooming	115
Annotating Maps – an overview	117
Time for action – adding annotations to your maps	117
Time for action – draggable annotations	119
Time for action – custom map annotations	123
Map overlays – an overview	126
Time for action – customizing map annotations	126
User tracking modes	129
Bonus – offline maps in your app	130
Time for action – using OpenStreetMaps with CloudMade API	131
Summary	134
Chapter 5: Weather App—WeatherPackt	135
Storing and retrieving the user's location with SQLite	136
Time for action – storing and retrieving the user's location with SQLite	136
Converting location data into city name – using Geonames API	142
A bit on GeoNames	142
Time for action – converting location data into city name	143
Consuming the WeatherBug API	148
Important things to know before we begin	148
Time for action – using WeatherBug API	150
Building your Weather App: WeatherPackt	162
Start a new Xcode project	162
Define the Home screen	165
Time for action – defining the Home screen	165
Set up a default location	168
Formatting the Weather API for display	169
The settings page	169
Bonus: building WeatherPackt with PhoneGap	174
Bonus: text-to-speech	174
Summary	177

Chapter 6: Events App—PacktEvents	179
PacktEvents: Overview and architecture	180
Architecture of PacktEvents	180
Storing and Retrieving Events with SQLite	181
Time for action – storing and retrieving events with SQLite	182
Plotting events on a map	191
Time for action – plotting events on a map	192
Filtering Events display by Event Categories	197
Time for action – filtering Events by categories	198
Using the Event Kit framework to add events to your iPhone calendar	205
Time for action – adding events to your iPhone calendar	207
Using the Twitter framework	210
Time for action – adding Twitter capabilities to your iPhone app	210
Bonus: using the Layar Player API in your app: Augmented Reality	213
Time for action – adding Augmented Reality to your iPhone app	213
PacktEvents: building the app	219
Summary	220
Chapter 7: Advanced Topics	221
Using directions with location	222
Direction using heading	222
Getting your app ready for direction	222
Understanding heading using magnetometer	222
Time for action – using heading for direction in your app	223
Direction using course	226
Time for action – using course for direction in your app	226
Core Motion: Motion Manager	229
How to use Core Motion	230
Time for action – using MotionManager: accelerometer	231
Core Motion conclusion	235
Background app execution	235
What apps can run in the background?	236
Background location	236
Push notifications - overview	240
Local notifications	241
Time for action – using local notifications	241
Summary	246
Chapter 8: Local Search—PacktLocal	247
Consuming the foursquare venue API	248
Venue categories	248
Time for action – consuming the foursquare venue API - categories	248
Recommended and popular venues	255

Time for action – recommended and popular venues	255
Search for venues	262
Time for action – exploring the foursquare Search API	263
Building an UI for our local search app - PacktLocal	267
Saving venue information on the device	268
Building the app: PacktLocal	268
Time for action – building the app - PacktLocal	269
Summary	282
Chapter 9: Location Aware News—PacktNews	283
Understanding the Patch News API – HyperLocal News	283
Authentication	284
Taxonomy	284
Vertical	285
Format	285
Author	285
Finding stories by location	285
Find location by names	286
Time for action – consuming the Patch News API	286
Adding the Geo Fencing support	295
Time for action – adding the Geo Fencing support	296
Building our app - PacktNews	299
A bit on StoryBoard	299
Time for action – building PacktNews	300
Summary	313
Chapter 10: Social Governance—TweetGovern	315
Social governance – an overview	316
TweetGovern – behind the scenes	316
Stackmob	318
Our approach: Twitter	319
Icons and images	322
SDKs and frameworks	322
Time for action – creating the UI for TweetGovern	323
Time for action – detecting the user location and showing nearby issues	326
Time for action – creating and voting for an issue	335
Summary	347
Appendix: Pop-Quiz Answers	349
Index	353

Preface

iPhone Location Aware Apps Beginner's Guide is probably the first book from any technical publisher that teaches you to build real world applications (five of them). That's a bold step from PacktPub - by undertaking more lively practical examples, rather than 400 pages of text! The book lays emphasis on location services, due to the ever-increasing role of location in our day-to-day lives and increased geo-referenced content being produced/consumed on the Internet and Mobiles. Be it news, sports or gossip, consumers no longer want to read/search about content happening far off from their current location. If it is news – it has to be local, similarly neighborhood gossip and news is more relevant to consumers seeking information on their smartphones. Applications such as foursquare confirm this behavior.

This book will help you learn location based techniques using iOS 5 as well as solutions to common location and mapping problems, ranging from simple location usage to caching user's last position, from simple Google maps examples to using OpenStreetMaps. Find five full working apps as a part of the book (along with the source code and business logic).

In this book, we have covered everything to make your next killer app, from app design to using free icons and background from the Internet (of course with due attribution to the author/designers), from integrating Twitter in your iOS 5 app to using the Nuance Speech SDK. This book is a practical beginners guide for new comers to the Apple iOS world. Happy Reading.

What this book covers

Chapter 1, The Location-Based World, explains location-based services, how it works, and the role of GPS in Location Services. We also learn how Apple uses location-based Services in iOS. Buzzwords in the industry are also explored.

Chapter 2, The Xcoder's World, explains the Xcode tool, introduction to HTML5 with Phonegap, Appcelerator Titanium, and Sencha Touch. We also have a look at a couple of location-based APIs/ SDK including FourSquare, EventFul, and Last.fm.

Chapter 3, Using Location in your iOS Apps—Core Location, explains a number of techniques used to read location information from your iPhone. This includes reading location information on an event, and receiving location updates in your app automatically. We also look at Region monitoring with Core Location framework. Example apps using Foursquare, Eventful, and Last.fm are also included.

Chapter 4, Using Maps in your iOS apps—MapKit, brings us to Maps—We learn to use the MapKit Framework in our app. We go behind the scenes with a small review of Map Geometry. We also explore annotations and overlays along with their customizations.

Chapter 5, Weather App—WeatherPackt, builds a complete Weather App using WeatherBug API. It also provides a Settings page in the app to customize the Weather display. As a bonus to the readers, we also included the Nuance Speech SDK for reading out the weather!

Chapter 6, Events App—PacktEvents, builds an Events app that shows us nearby events, concerts, and gigs by Artists, by using the excellent Eventful.com API. This chapter also shows how to use the Twitter API in iOS 5, and gives us a taste of Augmented Reality with the Layar Player SDK.

Chapter 7, Advanced Topics, teaches us using directions with location background services including background location. It also explores the Motion Manager in iOS SDK, along with Push and Local notifications.

Chapter 8, Local Search—PacktLocal, works with the foursquare API to build a local search app, with geo-fencing support.

Chapter 9, Location Aware News—PacktNews, uses the AOL's Patch News API to build a hyperlocal news app. It uses the new iOS 5 Storyboarding feature in this application, with support for offline content using SQLite.

Chapter 10, Social Governance—TweetGovern. Twitter provides the backbone for this chapter and the accompanying app. We use Twitter and hashtags for building the business logic for our social governance app titled `tweetgovern`. We learn to use the twitter re-tweeting concept as well, building upon our business logic.

What you need for this book

To run the examples and apps provided in the book, you will need a Mac running on Intel Architecture with Xcode 4.2 or higher and iOS 5 installed on your iPhone or iPad.

Some examples need an API key, which is duly mentioned at the beginning of the chapter/topic.

Who this book is for

Novice to professional level iOS programmers, who want to master location awareness and augmented reality. Build five practical location-based iOS Apps from scratch, a first for any book, converting learning into actual implementation.

Conventions

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

Code words in text are shown as follows: "Writing a simple `Hello Location` app in Xcode and Objective C."

A block of code is set as follows:

```
if(range.location == NSNotFound)
{
    deviceType =@"iPad";
}
else
{
    deviceType =@"iPhone";
}
```

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

```
for(NSInteger i=0;i<count;i++)
{
    NSString *venueName  =  [[[items
        objectAtIndex:0]objectAtIndex:i]objectForKey:@"name"];

    if (![venues containsObject:venueName])
    {
        [venues addObject:venueName];
    }
}
```

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

```
2011-09-04 16:40:09.421 Hello Location GeoNames[3896:f803] Location
Inserted Cupertino
2011-09-04 16:40:33.977 Hello Location GeoNames[3896:f803] Location
Inserted Soho
2011-09-04 16:40:42.230 Hello Location GeoNames[3896:f803] Location
Inserted Wadala
2011-09-04 16:40:48.889 Hello Location GeoNames[3896:f803] Location
Inserted Cupertino
2011-09-04 16:40:55.913 Hello Location GeoNames[3896:f803] Location
Inserted Financial District
2011-09-04 16:41:04.692 Hello Location GeoNames[3896:f803] Location
Inserted Sydney CBD
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Enter **Hotels in San Francisco** as the search key and hit *Enter*".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1

The Location-based World

Location-Based Services will be worth \$10 Billion by 2016 – GigaOm

Location-Based Services (LBS) are a revolutionary, but still fresh from the oven, breed of services that has grown tremendously to carve itself as a new industry in just a few years.

*Location-Based Services is the next step in the evolution for search, on the web and mobile, adding the **Location Context** (where am I or things around me) for search. To quote from Wikipedia on the definition of LBS:*

A Location-Based Services (LBS) is an information or entertainment service, accessible with mobile devices through the mobile network and utilizing the ability to make use of the geographical position (read Geocodes or Latitude/Longitude) of the mobile device.

You may have already used LBS when on Twitter, Facebook, Foursquare, Groupon, or visit hyperlocal web pages such as Wcities.com, Yelp.com, Qype.co.uk, and Eventful.com to find the top venues in the city or events happening in your city.

Want to know how the location is determined? Continue reading the chapter to understand the different location detection methods and which one is the right choice for you.

In this chapter, we shall understand:

- ◆ Location-Based Services
- ◆ Buzz words in the LBS Industry
- ◆ Applications of LBS and common use cases
- ◆ How Apple uses LBS in iOS devices
- ◆ GPS – Global Positioning System
- ◆ Indoor and outdoor navigation with GPS

So let's get on with it...

Understanding Location-based Services

The concept of Location-Based Services (LBS used as reference henceforth in the rest of the book) refers to services that integrate a mobile device's location with other topical information to provide added value to users.

Consider a weather app that shows weather information for all of the cities in the United States of America. For a user living in San Francisco, this behemoth of information is not very helpful, unless he can see the exact weather information for his city. This is achieved by *mashing up* the weather information with the user's location (generally obtained using a GPS system).

Another example of LBS are **local search websites** such as `Wcities.com` that present a user with hyper local (read local, nearby or neighborhood-centered) information on hotels, restaurants, shopping, and entertainment venues that makes a user feel connected with the type of information shown to him/her.

The core requirement for LBS is GPS (covered in more detail shortly), a space-based satellite navigation system developed and maintained by the United States of America. Other countries have similar systems too; Russia has **Russian Global Navigation Satellite System (GLONASS)**, Europe has the **Galileo Positioning System**, India and China are working on their own positioning system as well, but GPS remains the most popular and preferred choice for device makers and application developers worldwide.

Anyone can use GPS freely by using either a Personal Navigation Device (Garmin, TomTom), or an In-Car Navigation System (Ford Sync), or by using a Smart Phone.

On the mobile front, LBS also use Google Maps and other cartographic API services extensively (even in cases where the device does not support GPS). This is done using rich map data and Geocoding services. Using Geocoding and smart algorithms, a user's position can be *guessed or approximated*. Mobile Operating Systems, such as Android, further the cause of LBS by integrating locations into the Core OS, where the location can be fetched, used, and updated by all applications.

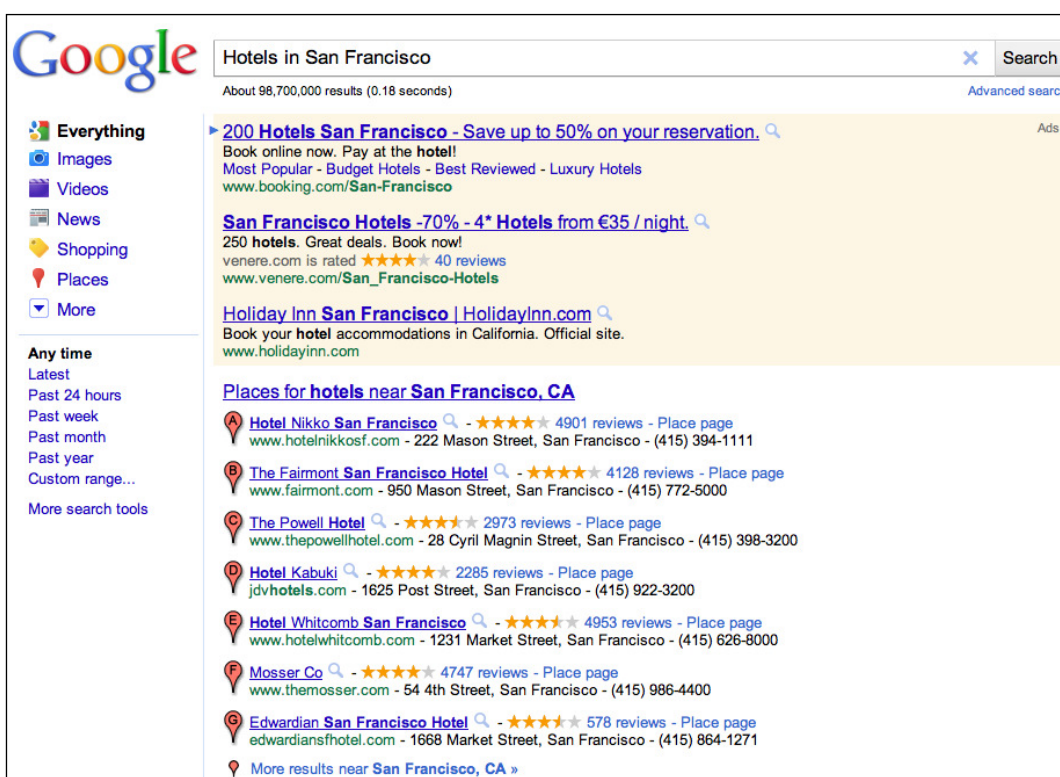
Apple iOS leads the pack with the best software API support, coupled with excellent hardware and positioning system integrated in the Apple Eco System. It also has network-based **Assisted GPS (AGPS)** that uses the network's data connection in the case of weak GPS signals as well as Apple's own Wi-Fi location database. iOS developers have a plethora of location tools and API to work with.

In short, LBS can be described as a combination of two components, **Location Providers** and **Location Consumers**, with GPS, AGPS, iOS API, and Google Maps API as the location providers and GPS receivers, mobile phones, and websites as the consumers of location data.

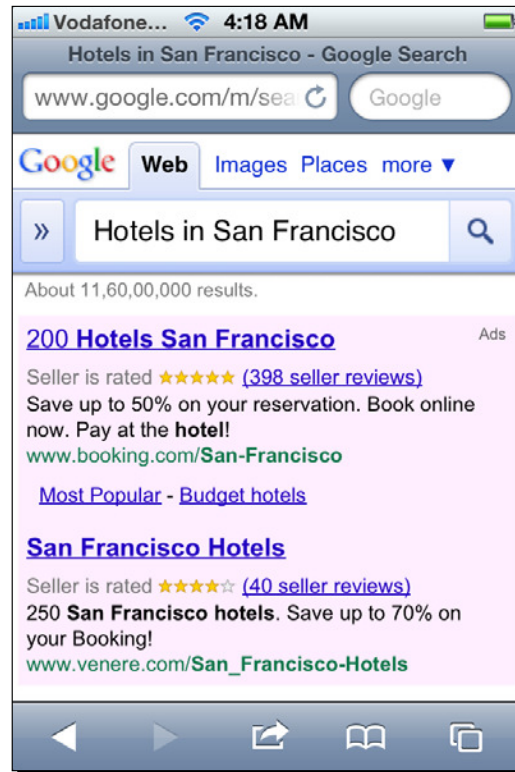
Time for action – consuming Location-based Services with Google

To understand how LBS work behind the scenes, let's take an example of the most common use of LBS, that is, how `Google.com` uses LBS for its search.

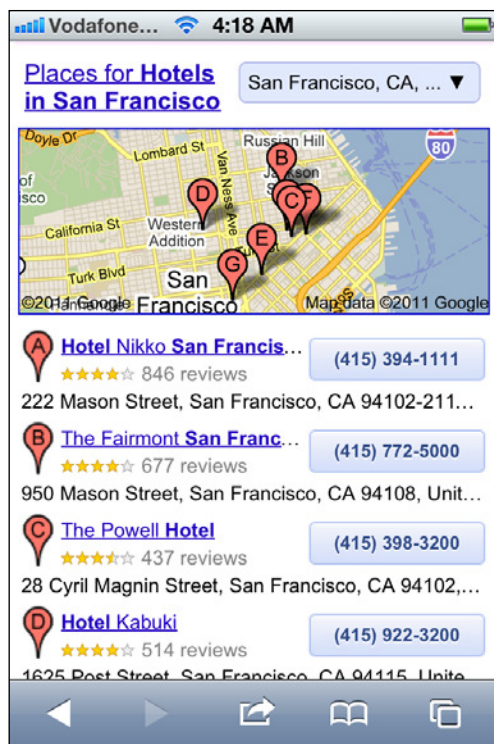
1. Fire up your Safari Browser and navigate to `http://google.com`.
2. Enter **Hotels in San Francisco** as the search key and hit *Enter*.
3. You are presented with results from the Google Places database, as shown in the following screenshot:



4. You get similar (but formatted) results from the iPhone browser search, as follows:



5. Scroll further down the page to see the actual results (following screenshot). The preceding screenshot is an ad-supported display that shows up on each search query (that's how Google makes money).



What just happened?

When you searched for **Hotels in San Francisco** on Google, Google first presented you with the hotels in its database (Google Places database) that matched the query for hotels as well as for San Francisco; this is done by Geocoding those hotels and storing it in a **Geographic Information System (GIS)**-aware database.

The mobile search on Google via the iPhone makes it more relevant to the user as it presents a nice map with the hotels plotted on the map and the hotels' details below (as seen in the preceding screenshots). Other mobile-optimized websites present similar content based on the location detected from the mobile phone.

Buzzwords in the Location-based Industry

As the book deals with iPhone location-aware apps, it is a good time now to understand the buzzwords and key terms used in the Location-Based Industry, so that term such as Check-In, GPS, AGPS, Geocoding, Reverse Geocoding, Geo fencing are made familiar to the reader.

GPS: Global Positioning System – A set of satellite systems that provides global navigation data including location and time.

GLONASS: Global Navigation Satellite System – Russian Navigation System.

AGPS: Assisted GPS – Mobile network-assisted GPS system, which uses the mobile network as a fall back in areas of poor GPS coverage.

GIS: Geographic Information System – A system for storing, processing, and retrieving geographically-aware data. It uses the user interface (usually Raster Map Images) for easier management. A GIS typically involves both hardware and software.

Spatial Database: A database management system that is used for storing, querying, and fetching geographically-aware databases and is used in conjunction with GIS for data management.

Geocodes: The latitude and longitude pair used to refer to a point on the earth's surface.

Geocoding: The process of converting a text address to Geocodes using Geocoding services such as GeoNames or Google Maps API.

Reverse Geocoding: The process of converting Geocodes to a text address.

Geo Fencing: Geo Fencing refers to the process of device-based alerts or notifications when entering a virtual geographical area. This geographic area can be a block, a lane, a neighborhood, a city, and so on, based on the application logic.

Check-ins: Made popular by start-up companies such as Foursquare and Footfeed, check-ins refers to the process of confirming that you actually entered/checked-in to a place via a mobile phone app.

GeoTagging: GeoTagging is the process of assigning Geocodes (latitude-longitude pair values) to any news article, blog post, twitter tweet, photo, or any other web resource, so that location-based searches can be performed on them.

Location-Based Advertising (LBA): Location-based advertising is a new paradigm in web and mobile ads, which are triggered by the location of the mobile device. Location-specific adverts for deals, events, movies, shopping, and restaurants offers are all possible with LBA.

Augmented Reality (AR): Augmented Reality is an exciting visual manipulation (augmentation) of the real-world environment (usually captured via a mobile phone camera) combined with computer generated (location-based) multimedia elements (pictures, audio, videos, 3D animation) usually in real time, giving the user a perception of superimposition of computer-generated elements onto the real world.

HTML5: HTML5 is the new version of the Hypertext Mark-up Language that is under heavy development at W3C and browser companies such as Mozilla, Apple, Google, and Microsoft. HTML5 is poised to bring in a new and better way of writing HTML pages using standardized tags that not only help the web developers maintain code reusability, but also makes it easy for search engines to semantically extract information from such HTML5 websites.

Application of LBS and common use cases

The primary use of LBS combined with GPS was and will remain the same, that is, navigation. There are new and exciting (and sometimes crazy!!) ideas being implemented in LBS every other day. Research and markets (<http://www.researchandmarkets.com/>) has predicted a market forecast of US \$10 billion for the LBS industry in 2015, from \$2.8 billion in 2010. **Gigaom** (<http://gigaom.com>), which is a technology blog by a Silicon Valley veteran Om Malik, has similar views on the LBS industry.

Government and military, navigation, and commercial industries such as advertising, social networks, and web portals are the primary consumers of LBS. GPS, in fact, was funded by the US **Department of Defense (DOD)** and still is maintained by DOD. It was initially designed for military use. In the late 1980s and early 1990s, it was opened up for civilian use. Let's review the common use cases:

Military

The US military uses GPS for navigation purposes including troops' movements. Target tracking weapons use GPS to track their targets. Military aircrafts and missiles use GPS in various forms.

Government

The government uses GPS for emergency services such as the US 911 service, which uses GPS to pinpoint the caller's location for faster pinpointing of the user and for providing emergency service on time.

Commercial

Navigational GPS units that provide car owners with directions to destinations are the biggest commercial users of GPS. Air traffic control, seaport control, freight management, car and transport tracking, and Yellow pages data management (local search) are other commercial uses of GPS.



Interestingly, GPS is also used for time synchronization. The precision provided by GPS improves the time data by 40 billionths of a second.

If you have the new iPhone 4S and are overwhelmed by Siri and its intelligence, then you should know that GPS and/or other location-detection methods play an important role in making the intelligent decisions.

How Apple uses LBS in the iPhone, iPad, and iPod devices

The Apple iPhone is a revolutionary Smartphone launched by Apple on June 29 2007. Since its launch, it has gone on to become the most popular Smartphone, carving a new market for itself. It has also seen revisions almost every year, with the current version being the iPhone 4S, launched in October 2011.

Besides the iPhone, Apple has other products, now branched together as iOS devices, which include iPod Touch and iPad 2, and they have all the features of the iPhone besides the fact that you cannot make calls with them. Apple has provisioned the iPhone 4 with the following location-supported hardware that helps the device establish location positioning for the core OS and apps:

- ◆ AGPS
- ◆ Digital compass
- ◆ Wi-Fi
- ◆ Cellular location

The iPod Touch uses Wi-Fi (Apple's location database) and the **Maps** application to approximate the user's location. The iPad uses Wi-Fi and the compass for location, while the iPad 3G uses AGPS, Digital Compass, Wi-Fi, and Cellular locations, just like the iPhone 4. The following table summarizes the location features in all the iOS devices:

Device	AGPS	Digital compass	Wi-Fi	Cellular
iPhone 4	Yes	Yes	Yes	Yes
iPod Touch	No	No	Yes	No
iPad	No	Yes	Yes	No
iPad 3G	Yes	Yes	Yes	Yes

Apple uses this hardware for location detection of the user on the software side; Apple's new advertising product **iAd** may also be used for location tracking of the user.

However, for the safe keeping of the user's location data, the user has to opt-in for the location tracking, so that iOS and third-party applications can use his/her location. Most apps show an alert message asking for user confirmation to use their location data. The iAd network also has an option where users can elect to share their data with the service via <http://oo.apple.com>

iOS location API

The **Core Location** API in the Apple iOS SDK is used for communicating with device hardware to get user location information. We will cover Core Location in the forthcoming chapters.

Core Location also supports direction-related API calls, the magnetometer in the iOS device reports the direction in which a device is pointing. Besides the heading, the GPS hardware can also return where the device is moving; this is known as its course. This is used by navigation apps to show continuous user movement. The Core Location framework contains various classes to handle the heading and course information. More details on this as we move along the course of the book!

Time for action – turning off Location Tracking in your iPhone

To turn on or off the Location Tracking by inbuilt and third-party apps in your iPhone (for iOS version 5), carry out the following steps on your iPhone:


1. Go to **Settings | Location Services**.
2. Select the apps that you want to allow usage of your location information.



3. To reset the **Location Warnings** made by applications such as Camera or Compass, go to **Settings | General | Reset** and select **Reset Location Warnings**.

4. In iOS 5, you can also switch location tracking for System Services, including iAds, Compass, Traffic, Time Zone Settings and Diagnostics, and Usage. These settings can be found at **Settings | Location Services | System Services**.



[ Note that this works for iOS version 5 (tested on iOS 5). For iOS 4.x, please refer to the official Apple documentation at <http://support.apple.com/kb/HT1975>]

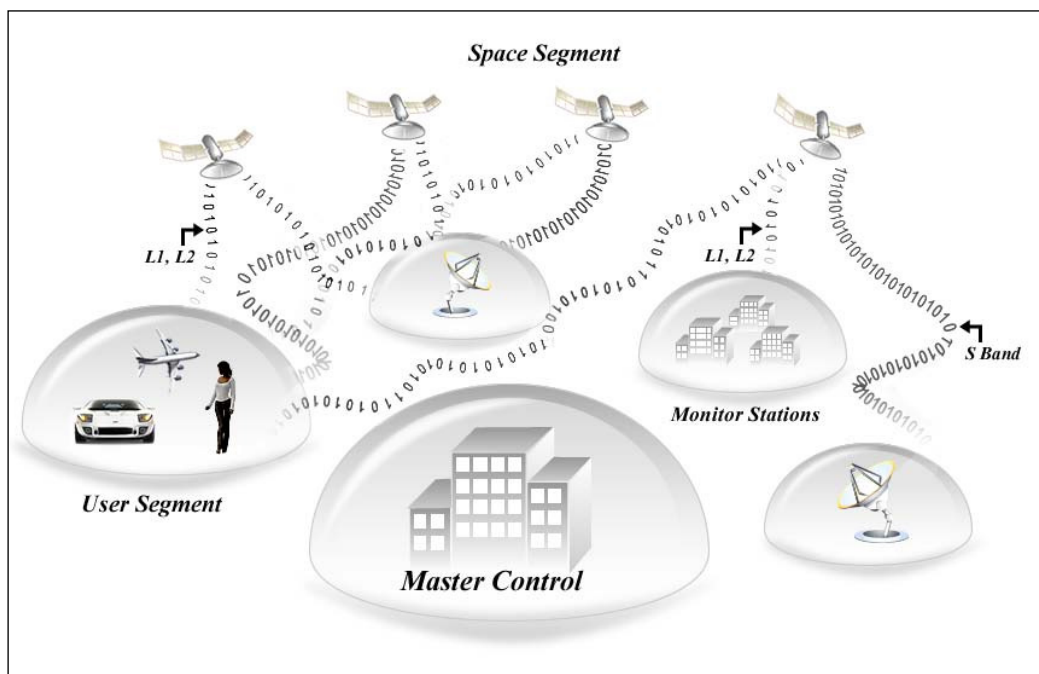
What just happened?

iOS versions prior to iOS 4.3.3 had a bug with the location settings, where the user's location information was stored on the iPhone, backed up to iTunes, and was open to hacks by third-party applications. Apple removed these bugs in version iOS 4.3.3, after a lot of hue and cry from security watchdogs. With the new release, the user's location history is deleted every time the user switches the location services off. Apple also reduced the cache size so less location information of the user will be cached on the device. In iOS 5, the location history of the user is encrypted, so third-party applications will not be able to read location information without the right authorization.

Behind LBS – GPS

Let's learn a bit more about GPS, as it powers all the current LBS implementations. If you are building the next generation navigation software or your own mapping applications, it's the right time to know more about GPS and how it works, so that it can help you make key decisions for your application.

GPS has three major components, as depicted in the following diagram, the user segment (GPS receivers, mobile phones, car navigation units), the space segment (24 satellites in orbit), and the GPS control segment having its base on Earth, with the **Master Control Station (MCS)** in Colorado Springs, Colorado (so now you know where to head to get a clear signal!).



User segment

The user segment comprises the GPS receivers embedded in millions of military equipment, almost all cell phones these days, aircraft, and car navigation systems.

Space segment

Space segment comprises the satellites orbiting the earth. The 24 satellites move on six different orbits around the earth at a distance of 20,200 km.

The satellites move in a manner that at every point of the earth's surface, at least five and at most 11 satellites are visible over the horizon for maximum accuracy.

Control segment

The control segment is the base on Earth that controls the functioning of the GPS satellites and passes on the administrative commands such as correcting the satellite orbit and internal data. Several monitoring stations receive the satellite signals based on their location; they are synced with atomic clocks to calculate the correction data. This corrected data is then sent to the Master Control Station.

Push and Pull methods of Location Services

LBS implementation is based on *Push Services* or *Pull Services*, depending on the way location information is retrieved.

Push Service

Push Services imply that the user receives location information without having to actively or continuously keep requesting for it. However, the user consent is acquired beforehand. For example, the navigation software in your car will require your consent to use your location information when you switch it on. However, as you drive your car around town, your new location will automatically be acquired via Push Services.

Some more examples of Push Service include Emergency Alert System (in case of terror attacks), location-based advertising apps on your phone that notify you with deals, messages, and alerts on entering a new city or town.

Pull Service

Pull Services work on the on-demand principle. Your apps would request location information from the network on demand, usually on application load, but is not limited to other stages in the app. For example, a Restaurant Search app on your iPhone would request location information when it loads, and you can change the location via the settings page of the app. In this way, the application *pulls* location information when needed and not continuously in the background.

In the forthcoming chapters, we will be building apps by mostly using the Pull Services, including a local search app and an events app that will pull location information on demand and mash it with information retrieved via Web Services.



Note that this type of location retrieval is also good for the battery power consumption of your phone, as GPS positioning does involve a significant amount of battery power.

Life without GPS: Wi-Fi-based location detection

There are alternative ways to detect the location through mobile phone devices using their Wi-Fi MAC addresses (access points that connect to the Internet) to determine the location. Wi-Fi-based positioning returns the approximate location, which may not be the exact latitude-longitude pair, but it would be the closest.

Companies such as SkyHook Wireless, and Google (with Google Latitude) were the first to provide this service. Apple launched a similar service in April 2010 with its own Location Database for devices having iOS version 3.2 and above.

Skyhook Wireless' location is pretty much public, with provisions for end users to add their location data to its database via a web interface, which is then available to all implementations of SkyHook wireless API users. Their database uses over 250 million Wi-Fi access points and cellular tower information for location analysis. Skyhook deploys data collection vehicles to conduct the access point survey, much similar to the manner Google Street views cars. The accuracy provided by SkyHook Wireless is ten meters. To know more about SkyHook Wireless coverage, visit <http://www.skyhookwireless.com/howitworks/coverage.php>

Google Latitude uses a mix of Wi-Fi, GPS, and cell tower-based location-positioning methods. It is tightly integrated with Google's Mobile Operating System – Android, and its Google Maps application. It works on PCs, Laptops, and mobiles alike. The Google Latitude app for iPhone is available from the Apple app Store from <http://itunes.apple.com/us/app/google-latitude/id306586497>; it supports automatic location detection coupled with automatic check-ins to nearby places, as you move around.



Life without GPS: cell ID positioning and cell tower triangulation

Low cost or price-sensitive mobile phones often come without GPS and Wi-Fi. These phones are meant to do what mobile phones are intended to be used for – Talk. However, the location of the user can still be detected on such phones using cellular towers. Cell ID Positioning and cell tower triangulation are two different ways to get location information from cell towers. Cell ID result accuracy is only 200-1000 meters, hence it is used as the last option for most location detection methodologies.

Cell ID Positioning uses your mobile network's cell tower to find your location. This involves the nearest tower your phone connects to in order to let you place calls. Cell tower triangulation, on the other hand, uses all the Cellular Towers around you to calculate your position based on the signal strength your phone receives from each of the towers. Triangulation is more accurate, but a slower process.

The iOS SDK has a region monitoring API that we will discuss in later chapters. This API is `CLLocationRegion`, which monitors the iPhone location and triggers an alert if you enter or leave a region. It works by using the Cellular Tower position as the trigger. When the iOS device detects a different Cellular Tower using the Triangulation technique to ascertain that the user has indeed crossed or entered a region, it triggers an alert to the application. This is an efficient way of location tracking without using GPS (and hence, more battery juice).



It is important to acknowledge privacy and security issues for end users while developing your applications and choosing the type of location detection and storage. In April 2011, it was discovered that Apple kept an unencrypted location database on your iPhones, even if the Location Setting was turned off. This file could tell any hacker where you have been and the timing details. Apple rectified this with a software upgrade, but it has been an eye opener for user privacy and security concerns.

Time for action – using the SkyHook Wireless Loki framework to determine your location

Loki is a SkyHook Wireless product targeted at website owners to help them locate their visitors. It is a JavaScript implementation done using the same SkyHook algorithms as on the mobile devices.

1. Go to <http://loki.com/findme> with your favorite browser.
2. You will get a permission request from a Java Applet, as shown in the following screenshot:



3. Wait for a few seconds and you should see your location detected.
4. If your Wi-Fi is not registered with SkyHook wireless, then you can do so by adding the same on http://www.skyhookwireless.com/howitworks/submit_ap.php

What just happened?

Loki.com uses a proprietary JavaScript code and uses a Wi-Fi Positioning system to determine your location. Users can also submit their Wi-Fi MAC ID to be included in the Loki database.

Loki also has a developer API that can be used by website developers to integrate a location in their websites.

Life without GPS: Google Maps API

Google Maps API is the most powerful mapping and Geocoding API, used by millions of developers to integrate locations and maps in their Web and mobile applications. It provides a rich set of APIs for Directions, Maps, and Geocoding. We will focus on the Geocoding API in Google Maps Version 3, as that is what we are interested with in this book; the rest is beyond the scope of this book.

Geocoding, as described earlier, is a process of converting addresses into geographic coordinates (latitude and longitude pair). Google Maps uses these co-ordinates to plot them on a map. Google Maps API provides options for both Geocoding and Reverse Geocoding.

The Geocoding API is a RESTful API that can be consumed with the following API call <http://maps.googleapis.com/maps/api/geocode/output?parameters> where the output can be json/xml and the parameters can use any one of the following:

address (required) or latlng (required)

bounds

region

language

sensor (required)

An example of a Geocoding request for South Park in San Francisco, CA, USA is constructed as follows:

```
http://maps.googleapis.com/maps/api/geocode/json?address=South+Park,+  
San+Francisco&sensor=false
```

This returns the following results:

```
{  
  "results" : [  
    {  
      "address_components" : [  
        {  
          "long_name" : "South Park",  
          "short_name" : "South Park",  
          "types" : [ "neighborhood", "political" ]  
        },  
        {  
          "long_name" : "San Francisco",  
          "short_name" : "SF",  
          "types" : [ "locality", "political" ]  
        },  
        {  
          "long_name" : "San Francisco",  
          "short_name" : "San Francisco",  
          "types" : [ "administrative_area_level_3",  
            "political" ]  
        },  
        {  
          "long_name" : "San Francisco",  
          "short_name" : "San Francisco",  
          "types" : [ "administrative_area_level_2",  
            "political" ]  
        },  
        {  
          "long_name" : "California",  
          "short_name" : "CA",  
          "types" : [ "administrative_area_level_1",  
            "political" ]  
        },  
        {  
          "long_name" : "United States",  
          "short_name" : "US",  
          "types" : [ "country", "political" ]  
        },  
        {  
          "long_name" : "94107",  
          "short_name" : "94107",  
          "types" : [ "postal_code" ]  
        }  
      ]  
    }  
  ]  
}
```

```

    ],
    "formatted_address" : "South Park, San Francisco, CA
                          94107, USA",
    "geometry" : {
      "location" : {
        "lat" : 37.78160380,
        "lng" : -122.39389940
      },
      "location_type" : "APPROXIMATE",
      "viewport" : {
        "northeast" : {
          "lat" : 37.7904220,
          "lng" : -122.3778920
        },
        "southwest" : {
          "lat" : 37.77278450,
          "lng" : -122.40990680
        }
      }
    },
    "types" : [ "neighborhood", "political" ]
  },
  "status" : "OK"
}

```



Downloading the example code

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

The values in bold are the ones of real importance to us, that is, the **geometry | location | lat** and the **geometry | location | lng** values. We will also be using the Geocoder provided in the iOS SDK in the coming chapters. If you are developing your apps in HTML5 for web and mobile, then you can check the Google Maps API at <http://code.google.com/apis/maps/>, as the W3C Geolocation standard has been implemented in Google Maps API.

Apple iOS Map Kit API uses Google Maps as the underlying technology to Geocode and reverse Geocode. We will cover Map Kit extensively in *Chapter 4, Using Maps – Mapkit*, but now is a good time to play with Google Maps API to get an overview of how things work.

Understanding Indoor and Outdoor Navigation

Navigation functionalities in cars, airplanes, rail, and mobile phones are mostly optimized for on-the-move functionality, it assumes that the user of such services tends to exhibit movement from one place to another with time; this is classified as **Outdoor Navigation**, implying navigation done outside homes, offices, malls, any place not confined to a building or large area.

This is where Indoor Navigation sets in, while GPS and other positioning systems have high coverage and accuracy, they fail when you are indoors, in a mall or a shopping complex, even airport lounges, stadiums, office complexes, as the radio signals from GPS transmitters cannot penetrate walls. Indoor Navigation works in such places, using techniques dissimilar to outdoor navigation; in short, no GPS for Indoor Navigation.

There are various implementations of Indoor Navigation, some using Infrared techniques, some using **Radio signals (RFID)**, and another implementation using Ultrasound. Companies such as VisioGlobe (<http://visioglobe.com>) offer an SDK for Indoor Navigational purposes. Another company – **WiFISLAM** is building a Wi-Fi-based solution. While the market for Indoor Navigation is quite big and the outlook for growth is very positive, the implementation and standardization is at a very nascent stage, partly due to the fact that a generic solution that fits all is not possible for Indoor navigation. Additionally, interactive kiosks at malls, airports, convention centers solve the problem of information management for visitors.

Google Maps on Android now include Indoor Navigation that allows you to navigate through floor plans for Airports, Shopping Malls, and Retail Stores. More information at <http://googleblog.blogspot.com/2011/11/new-frontier-for-google-maps-mapping.html>

Pop quiz – play safe with location!

1. What are the various methods of location detection?
 - a. Detect and Store
 - b. Push and Pull
 - c. Device hardware and software
2. To conserve device battery consumption, what method of location detection will you employ and why?
 - a. No Location
 - b. Pull methods
 - c. Push methods

3. In case there is no source of location detection, either via GPS, WIFI, or Cell ID triangulation, how will you model your Location-based app?
 - a. I am out of luck; need to remove the location feature completely
 - b. I will assume location information
 - c. I will ask the user and convert the user input to relevant location values, based on pre-set rules

Summary

In this introductory chapter, we have identified how LBS work—the buzzwords behind all things location and the importance of GPS.

Specifically, we discovered:

- ◆ LBS, and its applications in the real world
- ◆ How Apple uses Location in its iOS devices
- ◆ GPS – how it works
- ◆ Non-GPS-based solutions for location

We also discussed new potentials in LBS markets—that of Indoor Navigation.

Now that we've got our feet grounded in location, we can move to the next chapter that introduces Apple's Xcode IDE and HTML5-based app development tools.

2

The Xcoder's World

*Apple Xcode (the latest version is 4.2) is a complete **Integrated Development Environment (IDE)** for MacOS X and Apple iOS program development. It includes all the tools necessary to build, debug, test, and deploy apps for Mac OS, iPhone, iPad, and iTouch devices.*

In this chapter, we shall:

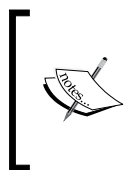
- ◆ Learn about Xcode 4's prerequisites and features
- ◆ Learn about the new iOS 5 SDK and the new features introduced
- ◆ Install Xcode 4 and understanding the new features in it
- ◆ Build a Hello Location app
- ◆ Introduce you to HTML5-based app Authoring Tools
- ◆ Explore Location-Based APIs and SDKs

So let's get on with it...

Introducing Xcode 4

Xcode is a set of developer tools packaged in a nice IDE, which brings together all of Apple's developer goodies under one umbrella. If you have used Eclipse or NetBeans before, then you will find Xcode to be quite similar in terms of IDE functionalities. Xcode not only supports making apps for iPhone, but it also supports Mac OS app development.

The Xcode toolkit includes the Xcode IDE, Interface builder, Apple LLVM Compiler, Debugger, and the Instruments analysis tool. This makes it a complete tool to design, code, test, debug, and submit your apps to the Apple Store, everything right under one tool. Not to forget the great iOS Simulator that lets you test your iPhone and iPad apps within the simulator, in case you do not have multiple devices to test your apps. The iOS 5 SDK now includes location simulation, so you don't have to run outside to test your apps for location testing! Let's explore the features and tools of Xcode in more granularity.



Note: The iOS simulator cannot simulate the Accelerometer or the Camera on your iOS device. You will need to test your apps on a real device, in case you intend to use either the accelerometer or camera in your app.

Xcode 4: Prerequisites and features

The top feature of Xcode 4, from a developer's point of view, is the built-in support for **Git**, the popular software version control system. Another notable feature is distributed building of source code via multiple computers using the Bonjour Protocol.

Prerequisites

Xcode 4 requires an Intel-based (x86-based processors) Mac running Mac OS X 10.6.7 and higher; support for development for PowerPC arch has been removed. You also need to be a member of Mac or the iOS Developer program to download Xcode 4. Alternatively, you can download Xcode 4 from the Mac App Store (Version 4.1.1 is now available for free on the Mac App Store).

Features

Besides a brand new, single window workflow interface, Xcode 4 has the following features:

1. Interface Builder – fully-integrated graphical tool for UI designing
2. Assistant Editor – finds and opens related files
3. iOS Simulator with Location Simulation (iOS 5 and Xcode 4.2 only)
4. C, C++, and Objective-C compiler optimized for multiple core processors
5. Dashcode – A **Rapid Application Development (RAD)** – tool for Dashboard widgets creation and web application development

6. Instruments tool with Data recording and visual comparison
7. Integrated Build System – for simpler app builds and on-device installation
8. Live Fix it – fixes your symbol names and code syntax as you type, with a single keyboard shortcut
9. Completed integrated documentation
10. Version Editor – Compare code revisions with SCM, use SVN or GIT, or both to manage your source code.
11. LLDB Debugger – Based on LLVM, it is a new debugger introduced to enhance performance and reduce memory consumption.
12. Miscellaneous tools for Audio, OpenGL, and Quartz for Animation

You can find a full set of features at <http://developer.apple.com/technologies/toolsfeatures.html>

To summarize the features and tools available in Xcode, with respect to the full application life cycle, you use the **Xcode IDE** to create a new project, the **workspace window** provides you with access to all the tools needed to build your application. The application user design is handled by **Interface Builder**, which allows you to position your UI elements as well as helps you connect the UI elements to actual code via **outlets** and **actions**. After which, you test the application on the **iOS simulator** to see if it works as intended; the simulator gives an almost real-world usage behavior. With iOS 5.0, you can also simulate location, and that was not possible before. The **Instruments Application** is used to analyze the performance of your application. Finally, with the build functionality in Xcode 4, you can get your app ready for distribution.

iOS 5 and Xcode 4.2: new and notable features

iOS 5 is the next increment in the iOS SDK line-up. It has been upgraded with 200 new features and some new never-before-seen features in iOS. Some of the biggest new features in iOS 5 are as follows:

- ◆ iCloud - Apple's cloud service
- ◆ iMessage - Apple's answer to blackberry messenger
- ◆ Twitter - Integration in the iOS base system

iTunes version 10.5 or higher is needed to activate iOS 5 on your iPhone.

iOS 5 new features

With iOS 5, Apple has introduced features that have been requested for a long time, both from the developer community as well as the consumers. Let us explore the notable features:

1. **Notification center:** All notifications, be it SMS, e-mail, or app alerts are now shown in one convenient location (usually on the top of your iPhone screen). This is quite similar to Android OS notification.
2. **Reminders:** Reminders are to-do lists with dates and locations enabled. The location reminder feature is a great way to show a reminder alert when you reach a particular location.
3. **Computer-free operation:** You no longer need a Mac or PC to work around with your iPhone, iPad, or iPod touch. All the tasks can now be done wirelessly, with backup and restore functionality added via the iCloud service.
4. **Wi-Fi Sync:** No cables needed to sync your iTunes library.
5. **Newsstand:** The newspaper industry is now friends with Apple and introduced newsstand. All newspaper and magazine subscriptions are now handled by this one application. There is also a newsstand store integrated, so you can search for and subscribe to new services.
6. **Camera and photo enhancements:** It has easy photo capture, autofocus, and gridlines for the camera app and crop, rotate, enhance and red eye removal for the Photo app. It also has some nifty iCloud features.
7. **Safari:** The Safari browser has been enhanced with a reading list that lets you save articles to read later. Tabbed browsing on the iPad and a clutter free reading mode are pleasant additions. For developers, Apple has brought the Nitro Engine to Home Screen Web apps.
8. **Air Play Mirroring for iPad 2:** Stream HD content from your iPad 2 to your HDTV via Apple TV. It is very useful for classroom and board meetings.
9. Game Center, mail, calendar, multitasking gestures for iPad are some other enhancements added in iOS 5.
10. **Twitter Integration:** You no longer need to sign in to multiple twitter-enabled applications every time you use the applications or switch from one application to another. Just sign-in once via your iPhone settings and use the twitter services from any application or service—no more multiple OAuth and login. Tweeting can happen via Safari, Photos, YouTube, or Maps without re-entering your credentials. Same with third-party applications.


11. **iMessage:** iMessage is a new messaging platform that works over Wi-Fi or 3G, enabling you to send unlimited text messages. Group messages, sending images and videos, reading receipts, Google chat such as keyboard updates, like *'xyz is typing...'*, are other features in iMessage
12. **iCloud:** All the new apps added in iOS 5, be it Twitter or Reminders, as well as existing apps such as photos and e-mail, are all iCloud enabled. This creates a good way of sharing your data across your iOS devices – iPhone, iPad, or iTouch.
13. **Speech Recognition (Siri):** Apple has added Speech-to-Text and Text-to-Speech functionality in iOS 5, which is only compatible with iPhone 4S. In collaboration with Nuance, the feature is known as **Siri**, which is poised as an intelligent assistant to iPhone users. It can schedule appointments, read out text messages and e-mails, as well as send out e-mails by transcribing your voice to text. It is still in the beta stage, but is a revolutionary step for mobile phone users.

Xcode 4.2's new features

The iOS 5 SDK includes a new version of Xcode 4.2 that is needed to develop apps on iOS 5. Some of the improvements done in Xcode 4.2 include the following:

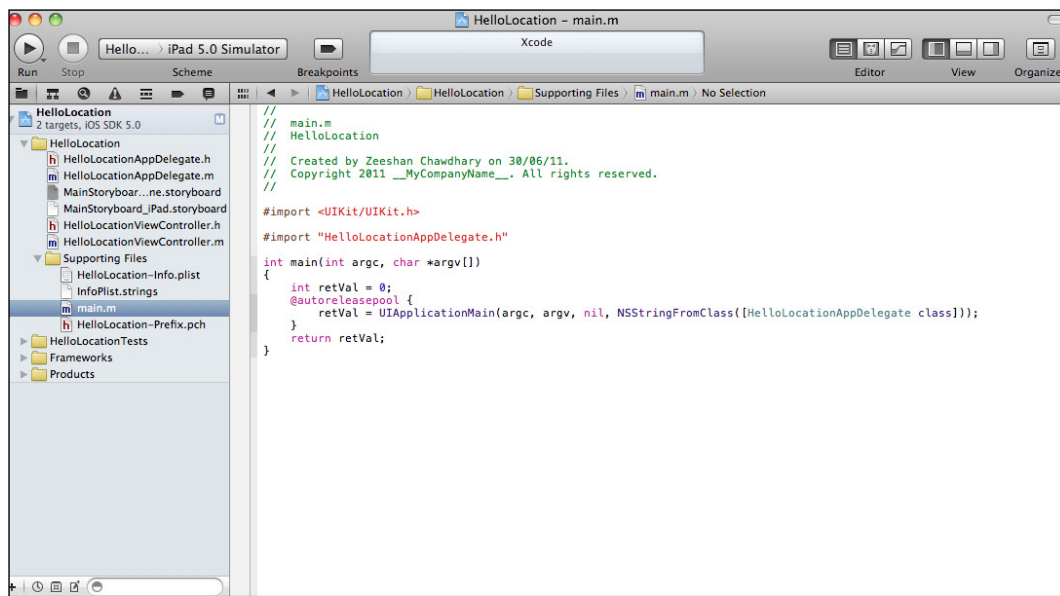
1. **Location Simulation:** You can now simulate location data for your app from within Xcode.
2. **Storyboard:** Storyboarding is a new feature added in the Interface builder to manage the transitions between different views in your app.
3. **Automatic Reference Counting (ARC):** ARC eases the developer's pain of memory management, which is often the toughest and most brain whacking exercise in app development. With ARC, the LLVM compiler takes care of the memory retain or release cycle.
4. **LLVM compiler:** The LLVM compiler is the default compiler now.
5. **OpenGL ES Debugging:** Frame capturing from OpenGL ES-based apps is now possible.
6. **New Instruments tools:** The Instruments tools have added some new instruments:
 - ❑ **Network Connections Instrument:** Understand the data flow from TCP/IP and UDP connections for your app; seek latency times and other statistics with this tool.
 - ❑ **System Trace:** Profile system calls, thread scheduling, and VM operations with Dtrace.
 - ❑ **Automation:** This is a nice and much-needed instrument to test your iOS Application's UI, done via JavaScript. Imagine running User interaction scripts on your app UI and getting logs back from the Automation Tool.

Besides these, there are more new features added, as well as enhancements done to existing functionalities. New frameworks include the Twitter, OpenGL framework, and Image and Accounts framework. Enhancements have been done in the **UIKit**, **EventKit**, **MapKit**, **Game Kit**, **Core Data** (with iCloud support), **Core Motion**, and **Core Location** frameworks.

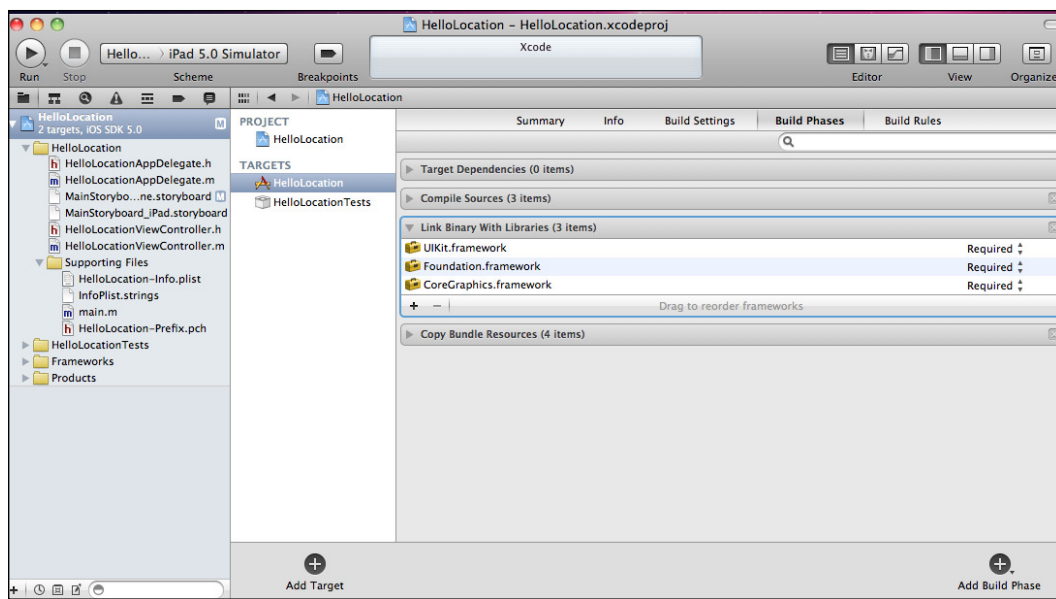
 Another very big feature for developers is the ability to download your application data from an iOS device and automatically restore that data when debugging or testing in the iOS simulator or even on a new device. This is a boon to developers, especially in circumstances when everything works fine on the simulator but crashes on the device.

Transitioning from Xcode3: What you need to know

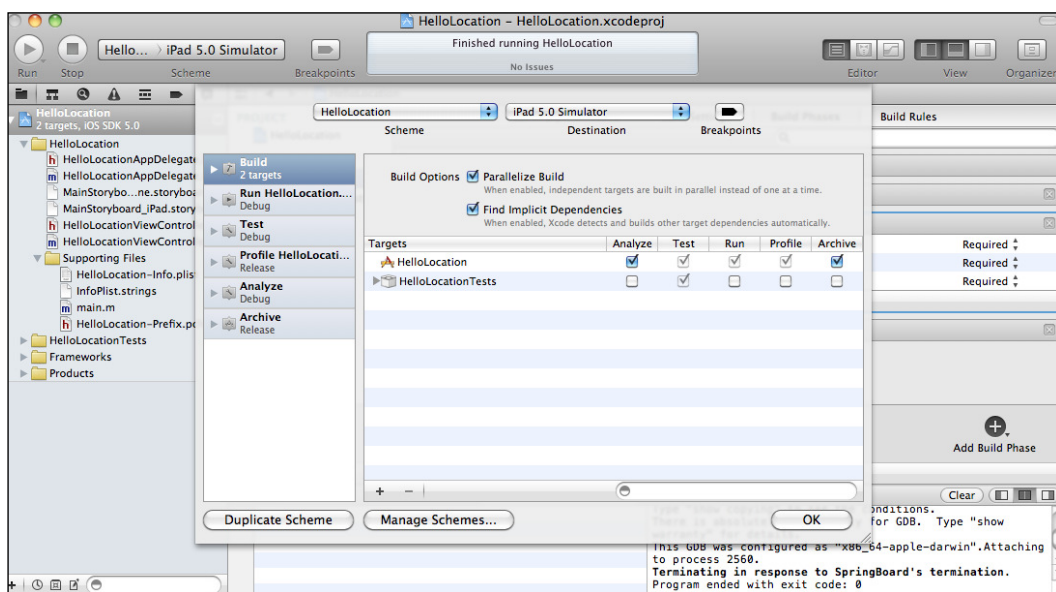
You can open Xcode 3 projects in Xcode 4. The major new feature of Xcode 4 is the workspace, which is a container for all the files in your project or multiple projects as well.



Working with external libraries is different in Xcode 4. In order to add libraries to your build target, you need to browse to **Project | Targets | Select the Target | Build Phases** and then manage the libraries from the **Link Binary with Libraries** group.



The compiler has been changed from GCC in Xcode 3 to LLVM in Xcode 4, the features of which we have discussed in text before. The `target` | `build` | `execute` configuration has been revamped in Xcode 4 by introducing a new concept called scheme. It specifies which targets to build for, what build configuration to use, and which executable to run, all specified in a scheme. You can edit a scheme via the Scheme Editor from the Product Menu in Xcode 4, as shown in the following screenshot:

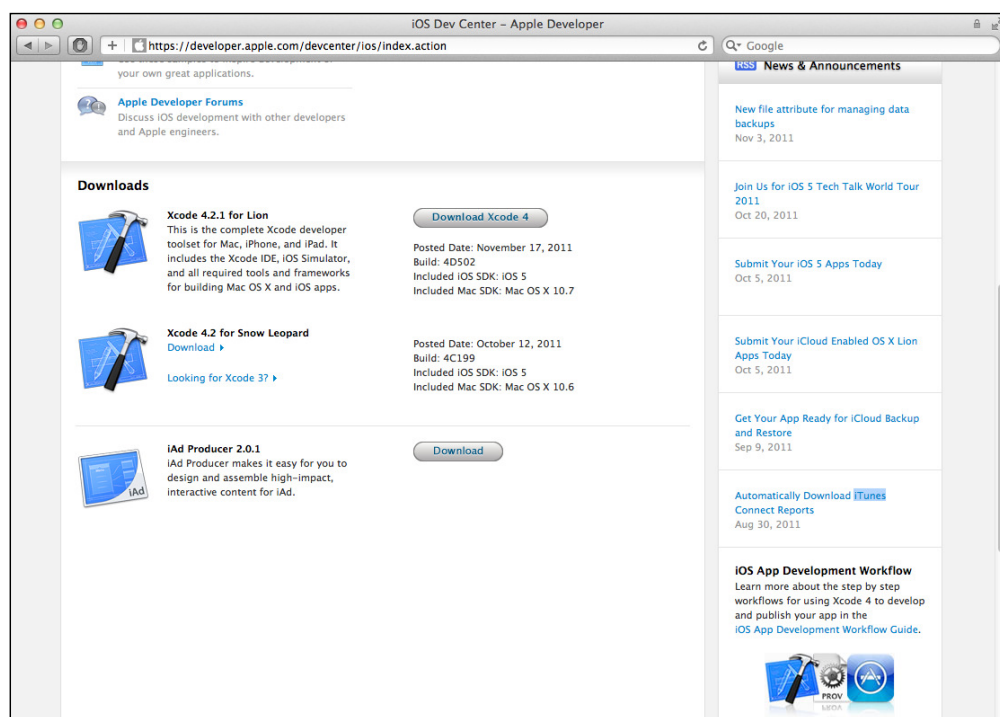


There are tons of new enhancements added in Xcode4, detailing them all is beyond the scope of this book. Apple has a good transition guide available at the Apple developer website titled *Xcode 4 Transition Guide*, which details the transition from Xcode 3. We will cover the related tools and features of Xcode 4, as we build our apps and understand location-based services throughout the course of the book. Let's get started with installing the iOS SDK 5 and testing some location features with our `Hello Location` example.

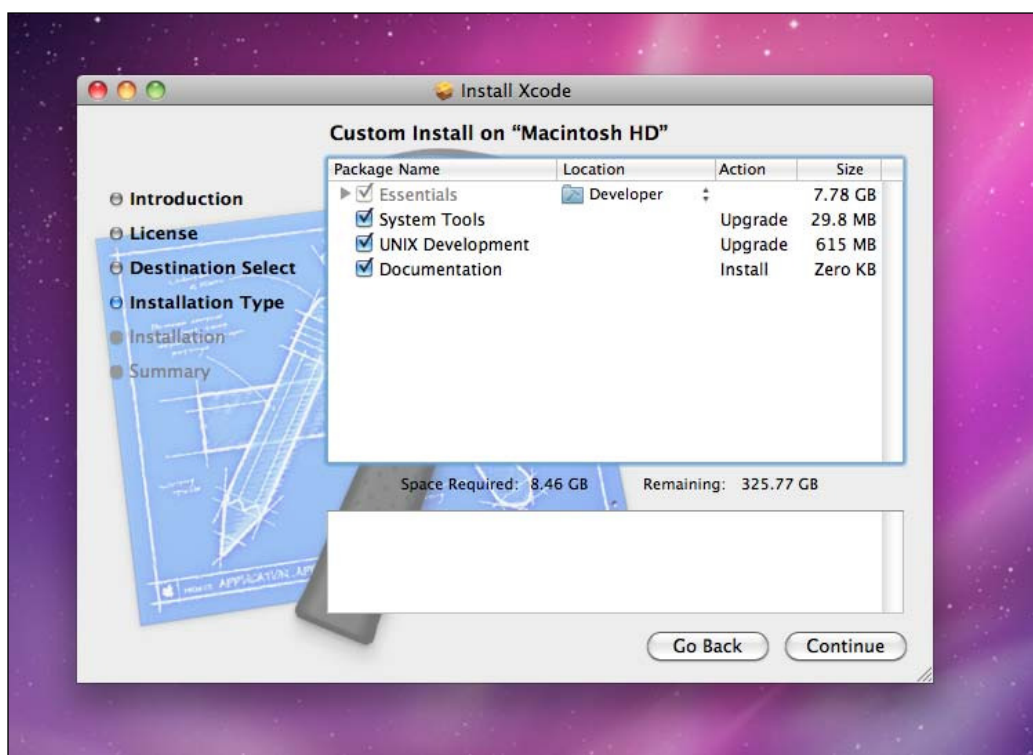
Time for action – installation

You need to sign up for the Apple Phone Developer Program to download Xcode and iOS SDK and to submit your apps to apple iTunes store. It is a 99\$ per year program (if you choose the standard iOS Developer program, then there is an Enterprise developer program as well). Alternately, you can now download Xcode 4 from the Mac App Store. Here is how you can obtain the right iOS SDK from Apple's Developer site:

1. Go to <http://developer.apple.com/ios> to sign up for an iOS Developer Account.
2. If you have an Apple ID, use that; else you need to create one.
3. If you are using Snow Leopard, then download Xcode 4.2 for Snow Leopard. If you have upgraded to Mac OS X Lion, then you need to download Xcode 4.2.1 for Lion.



Installation of the iOS SDK is pretty straightforward; double-click on the download file and follow the instructions.



Make sure you have enough free disk space; the complete installation takes around 8.5 GB of disk space. If everything went well with the installation, then you will find Xcode and other tools in the `/Developer/Applications` folder on your Mac.

What just happened?

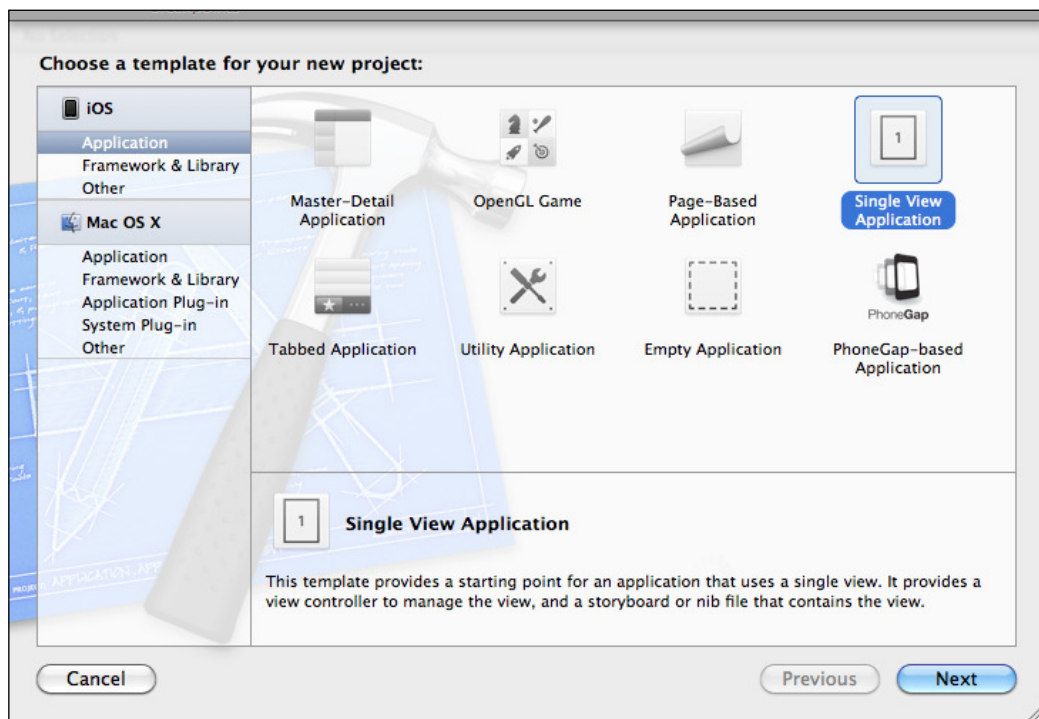
We downloaded and installed Xcode IDE from the Apple Developer site. Make sure you download the right DMG package, as there are two versions of the SDK now: one for Mac OS X Lion and another for Mac OS X Snow Leopard. So, depending on what version of the Mac OS you have, download the appropriate package.

Furthermore, make sure you download the updated iTunes version and have the latest iOS image file for your iPhone, iPad, or iPod Touch to use the device for development.

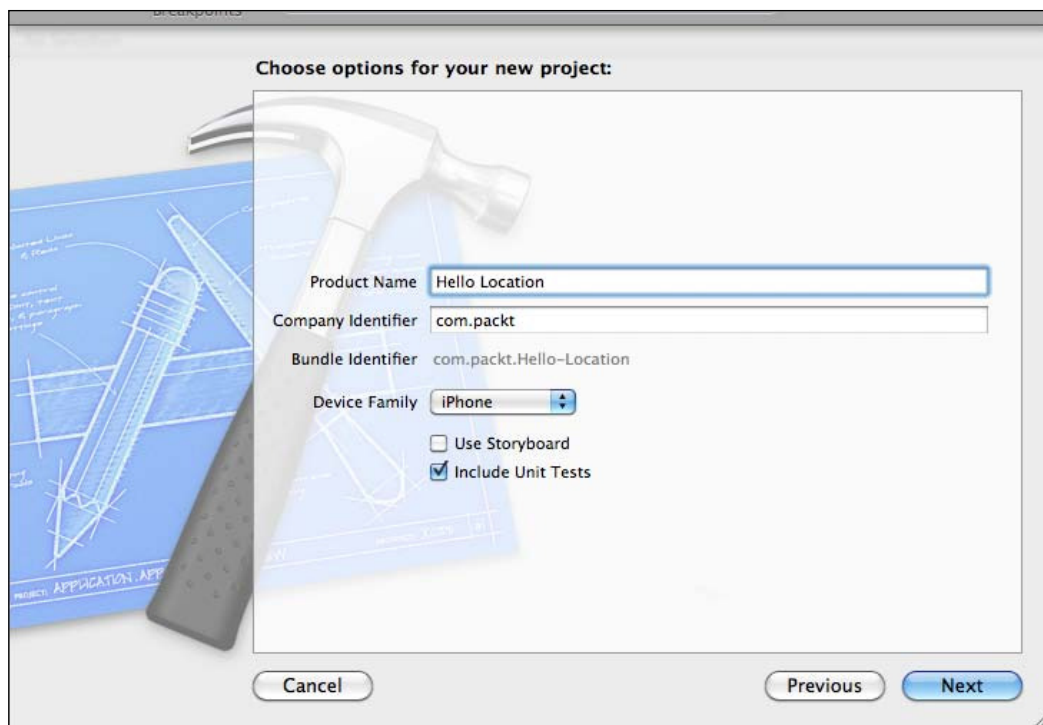
Time for action – Hello Location

Now that you have installed Xcode 4, let us quickly build a `Hello Location` app that detects your position and shows it (latitude and longitude pair) on the iPhone screen. Don't worry if you do not understand everything right away; we will explore more over the course of this book.

1. In Xcode, go to **File | New Project**, and select **Single View Application**.

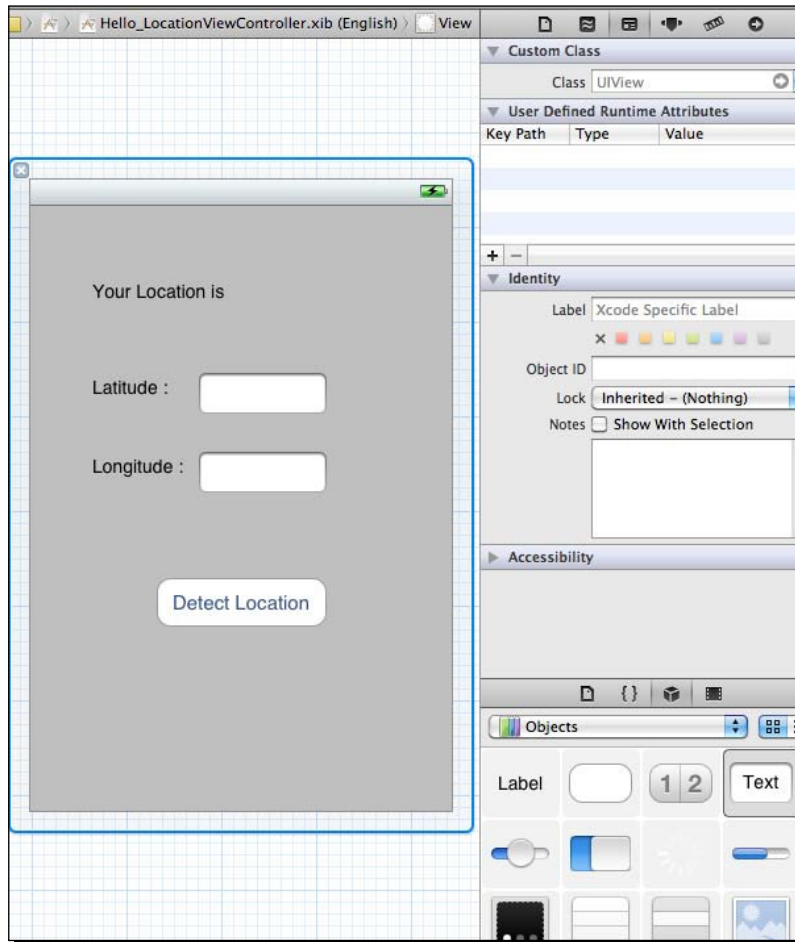


2. Name the product as `Hello Location` and the company identifier as your company names; in this case, we are using `com.packt`. Continue to save the project on your disk.



3. Now that your project is created, it is time to add a label to the `Hello_LocationViewController.xib` file, which holds your projects' main UI. Select the `Hello_LocationViewController.nib` file in the workspace and add a `Label` control from the Library Pane. Double-click on the `Label` to set its text to `Your Location is`.
4. Similarly, add two more `Labels` below the first one and change the text to `Latitude` and `Longitude`. These labels are the placeholders for our location values retrieved from the device.
5. Next, add two `Text Fields` next to the labels defined above (one for holding the latitude value and the other for the longitude value).

6. Lastly, add a **Round Rect** button and change the title to Detect Location. Your UI should like the following screenshot:



7. Now let's add the code to detect the user's location and add it to the latitude and longitude text fields added in step 5.
8. Open `Hello_LocationViewController.h` and add the `CLLocationManager` Delegate directive just after the `Hello_LocationViewController` class definition. Next, define two Outlets, namely, `UITextField *latitudeText` and `UITextField *longitudeText`. These outlets serve as the connector to your UI textfields created earlier.
9. Do not forget to import the `CoreLocation` header file by using the `#import <CoreLocation/CoreLocation.h>` directive.

10. Now define the properties of these outlets, so we can reuse these objects in our class implementation.

11. We also define an action called `locationDetect` that is fired when the **Detect Location** button is pressed. Here is the complete code for the `Hello_locationViewController.h` file:

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

@interface Hello_LocationViewController : UIViewController
<CLLocationManagerDelegate>
{
    IBOutlet UITextField *latitudeText;
    IBOutlet UITextField *longitudeText;
    CLLocationManager *locMgr;
}

@property (retain, nonatomic) IBOutlet UITextField *latitudeText;
@property (retain, nonatomic) IBOutlet UITextField *longitudeText;

- (IBAction)locationDetect:(id)sender;

@end
```

12. In the `Hello_LocationViewController.m` file, add the following line:

```
@synthesize latitudeText, longitudeText;
```

13. In the `viewDidLoad` function, add the following lines after the `[super viewDidLoad]` statement:

```
locMgr = [[CLLocationManager alloc] init];
[locMgr startUpdatingLocation];
```

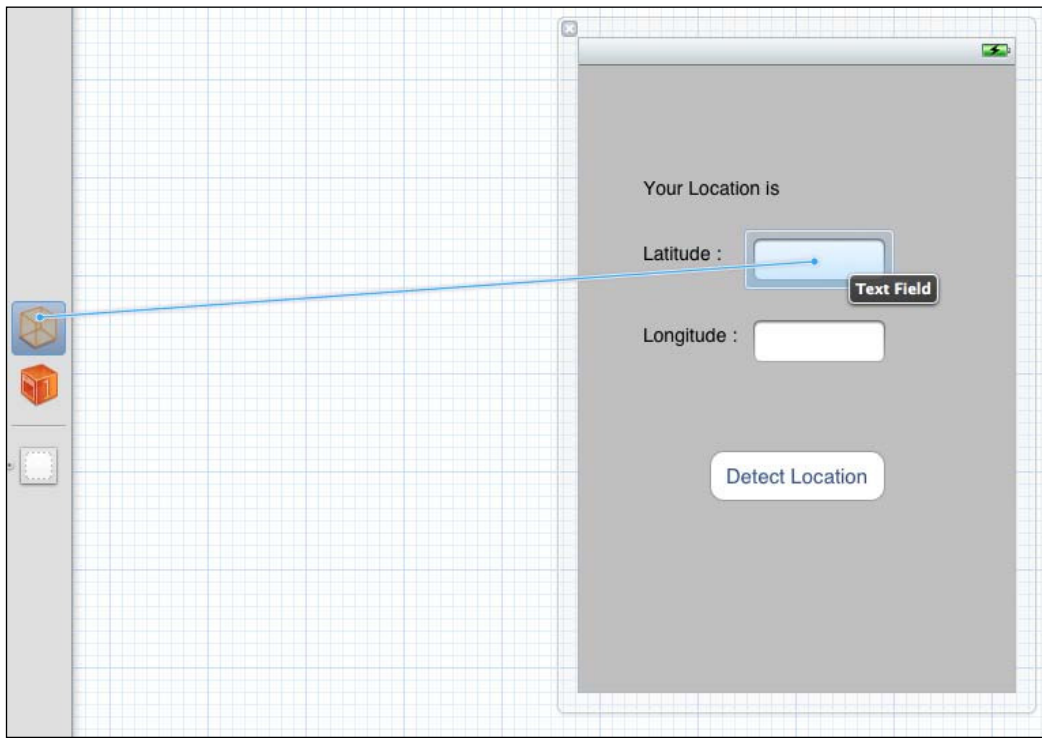
14. We created a `LocationManager` object and started the location update process by using the `startUpdatingLocation` method. Now we implement the `locationDetect` function defined in `Hello_LocationViewController.h`:

```
- (IBAction)locationDetect:(id)sender
{
    NSString *latitudeTextData = [[NSString
    alloc] initWithFormat:@"%g", locMgr.location.coordinate.latitude];

    NSString *longitudeTextData = [[NSString alloc]
    initWithFormat:@"%g", locMgr.location.coordinate.longitude];

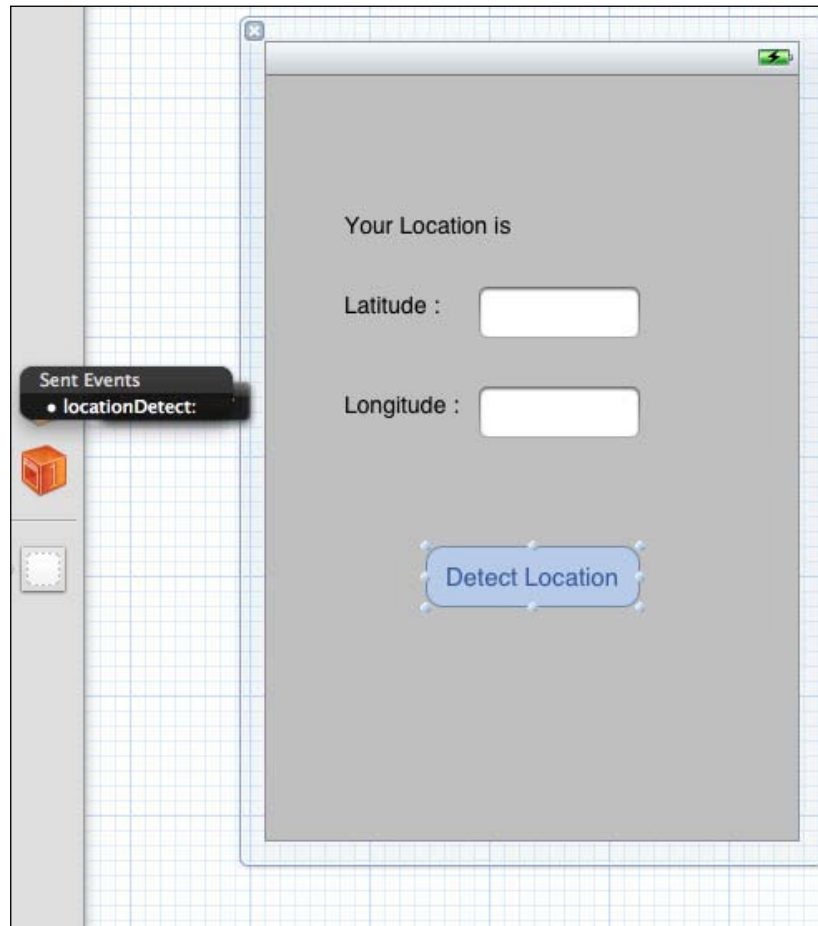
    latitudeText.text = latitudeTextData;
    longitudeText.text = longitudeTextData; }
```

- 15.** We capture the location values retrieved from the `LocationManager` Object and convert it to a `Double` value, and then finally, change the textbox values with the corresponding latitude and longitude values obtained.
- 16.** Now open the `Hello_LocationViewController.xib` file to connect the outlets and action, defined above. Press and hold the `Ctrl` key and click on the **File's Owner** item and drop it on the `UITextField`, which we designed to hold the Latitude value, as shown in the following screenshot, and select the `latitudeText` outlet.

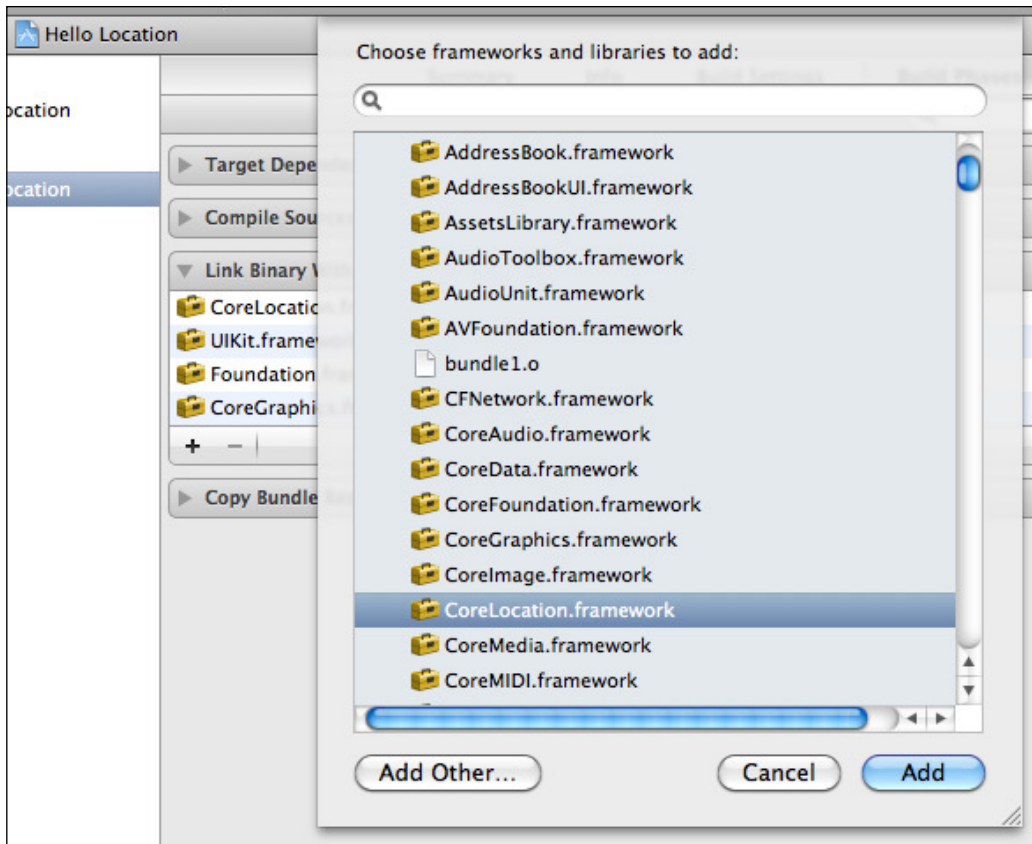


- 17.** Similarly, we connect to the `longitudeText` outlet from the **File's Owner** item.

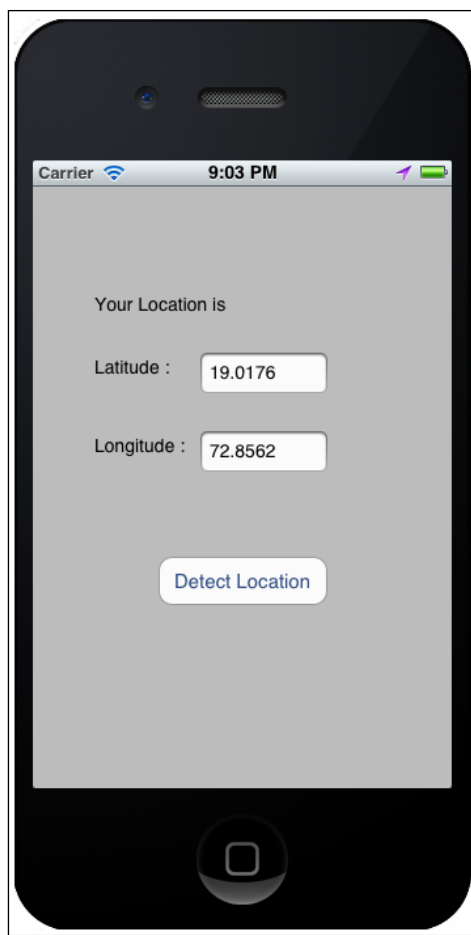
- 18.** Now let's connect the **Detect Location** button to its action – `locationDetect`; this is done by holding the *Ctrl* key, clicking on the **UIButton (Detect Location)** button, and dropping it on the **File's Owner** item. Note that connecting the Actions is a complete reverse of the way we connect the Outlets.



- 19.** You need to add the CoreLocation Library in your Build Path. To do so, navigate to **Target | Build Phases | Link Binary with Libraries** and add the Core Location Library.



- 20.** Build and run the application on the IOS Simulator (simulate the location by using the new Location Simulator feature of Xcode), or on the iPhone4, you should see an output, as shown in the following screenshot:



What just happened?

To summarize the `Hello Location` application, we created a Single View-based application, quickly added the `UILabels` and `UITextBoxes` to the UI, and created a `UIButton` that is used to trigger the location values and display on the textboxes created.

When the application is loaded, we start the `Location Manager Update` method to start reporting the location update values to the application. On button press event of the `Show Location` button, we fire an event that obtains the location values (latitude and longitude pair) and renders them on the textboxes.

The complete code for this application can be found on the book page at PacktPub's website <http://www.packtpub.com/iphone-location-aware-apps-beginners-guide/> book in a project named Hello Location.

Tools for the overnight coders: HTML5

If you have been following mobile apps development recently, then you might have heard about HTML5 and how it is being used for mobile apps development. Products such as PhoneGap, Appcelerator Titanium, and Adobe Dreamweaver offer the web designers a quick way for rapid mobile apps development using their existing Web Development skills.

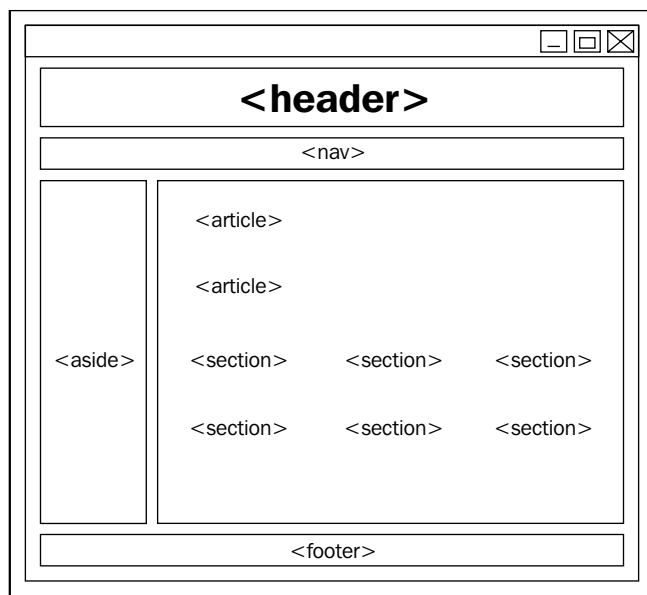
HTML5 combined with CSS3 and JavaScript has become the killer combination for easy-to-develop web-native apps for iPhone and Android. There are quiet discussions on the web on native apps versus web apps, but it seems web apps are the future. Technologies such as **WebGL**, offline storage are bridging the gap between native and web apps.

HTML5 is a new standard of Web Programming geared to make the Web more semantic and syntax aware. It also strives to have a common toolsets for all browsers, so as to remove the cross-browser issues. In short, HTML5 makes a developer's life easier, as it has been clearly thought of, considering decades of HTML usage and its drawbacks.

The major features of HTML5 are the addition of the **Canvas** element, **Video** element, Geolocation standard, Drag and Drop, Animations, Web Sockets (No more Ajax!!), and Offline storage. HTML5 also introduces new tags to easily structure the page; these are as follows:

- ◆ `<header>`
- ◆ `<nav>`
- ◆ `<article>`
- ◆ `<section>`
- ◆ `<aside>`
- ◆ `<footer>`

To illustrate how these tags are used for describing the structure and semantics of an HTML5 page, let us visualize an HTML5 page, which uses these new tags, as follows:



The HTML5 markup for such a page would be as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>HTML5 - Hello Location</title>
  </head>
  <body>
    <header>This is the header</header>
    <nav>Navigation Links and Menus here</nav>
    <aside>Ads, Sitemap and more links</aside>
    <article>This is Article 1 </article>
    <article>This is Article 2 </article>
    <section>
      <article> This is Article 3 </article>
    </section>
    <section>
      <article> This is Article 4 </article>
    </section>
    <section>
      <article> This is Article 5 </article>
    </section>
    <footer>Footer comes here</footer>
  </body>
</html>
```

The W3C – Worldwide Web Consortium has proposed a final deadline of 2014 for the complete specification and standardization of the HTML5 standard. However, most modern browsers already support HTML5 including Safari, Chrome, and Firefox. There are excellent write-ups on the net for HTML5 with examples and demos; here are some important links to get more details:

<http://www.html5rocks.com> - From Google

<https://developer.mozilla.org/en/HTML/HTML5> - Mozilla Foundation

<http://www.apple.com/html5/> - From Apple

The ease of use, native phone UI, and excellent new features of HTML5, as well as avoiding the steep learning curve for iPhone and Android app development, has made HTML5 a rich contender for building mobile phone applications over the past year. Products such as **PhoneGap** and **Appcelerator Titanium** are the front-runners in this space, each offering different solutions, centered around HTML5 and CSS3.

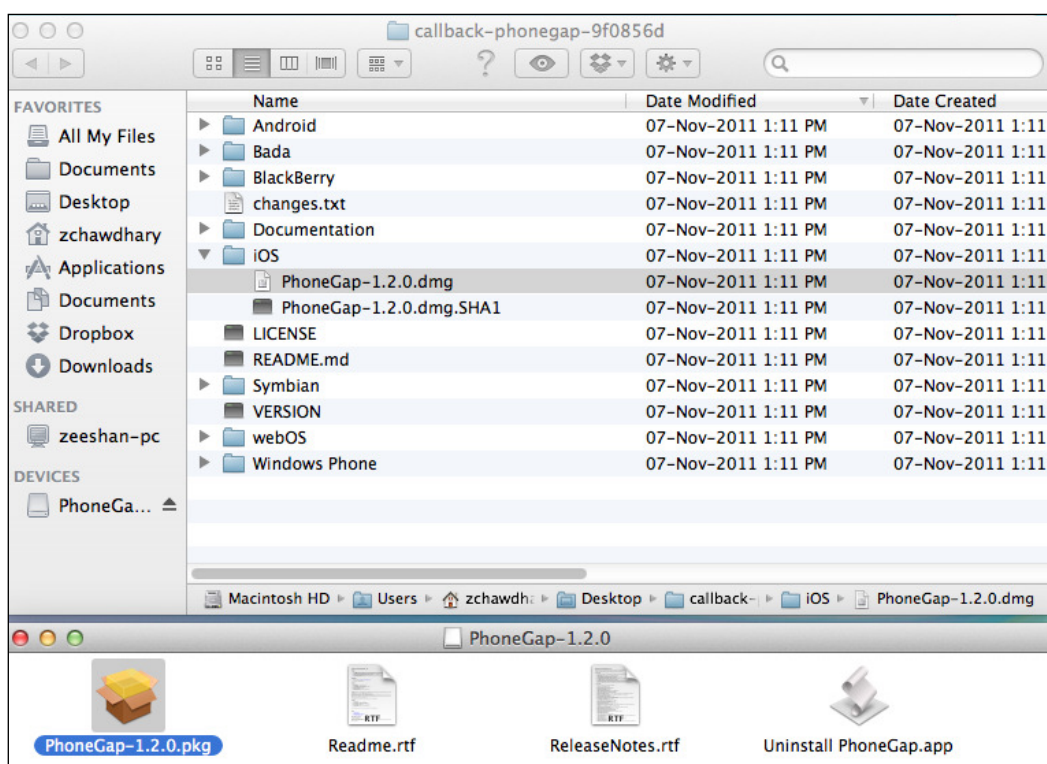
PhoneGap

PhoneGap is an open-source product that helps build cross-platform mobile apps for iOS, Android, BlackBerry, Nokia – Symbian, and HP-Palm WebOS platforms. The same HTML5, CSS, and JavaScript code can work on all the platforms, so no more tearing your hair porting your app from one platform to another. So it brings true the *Write Once, Run Anywhere* functionality to modern day mobile app development.

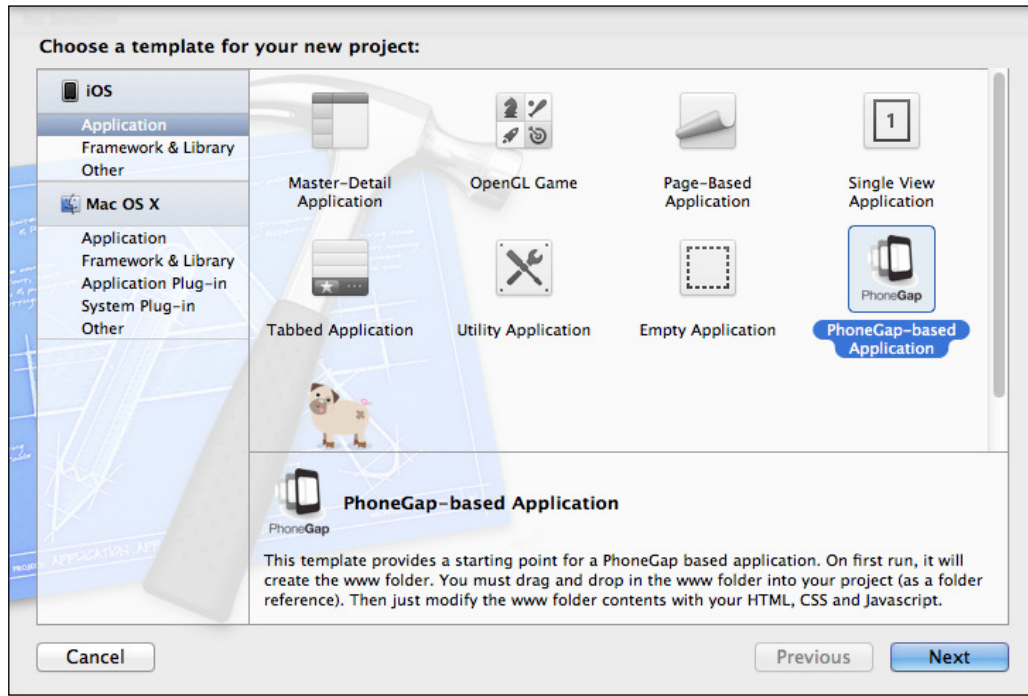
PhoneGap allows web developers to access the mobile camera, GPS, accelerometer, and contacts without writing platform-specific code, thus allowing true cross-platform development. Let's not waste time and dive directly into installing PhoneGap and writing our `Hello Location` app with it.

Time for action – using PhoneGap to build a Hello Location App

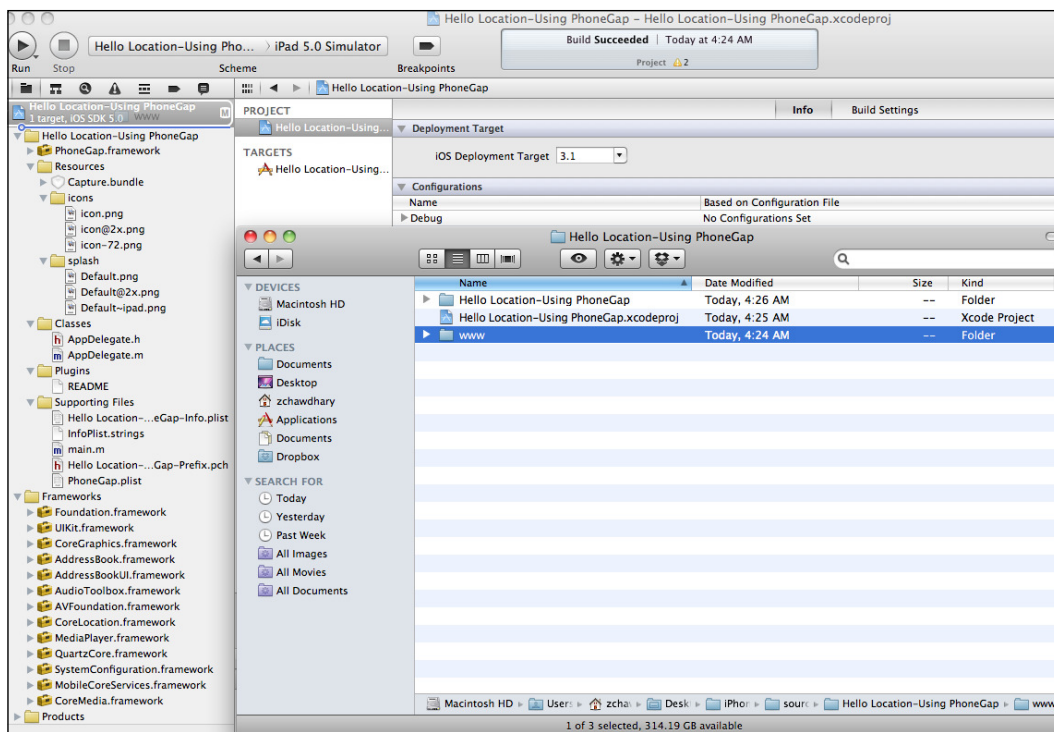
1. Go to <http://phonegap.com> and download the PhoneGap archive (version 1.2.0, as of this writing)
2. Unzip the PhoneGap ZIP file and traverse to the iOS folder in the unzipped folder (see following screenshot). Double-clicking on the PhoneGap-1.2.0.dmg file will mount the PhoneGap-1.2.0 folder on your desktop containing PhoneGap-1.2.0.pkg. Double-click on the .pkg file to run the phonegap add-in installer for Xcode 4.



3. Start Xcode 4 and set up a new phonegap project by selecting **File | New | New Project** from the **File** menu and then selecting **PhoneGap-based Application**.



4. Name your project **Hello Location-Using PhoneGap** and complete the creation of your project.
5. Build your project to generate the `www` folder, where your HTML code and JavaScript files will reside.
6. Browse to the folder where your project resides and drag-and-drop the `www` folder to add it to the project.



7. You will be prompted to add the `www` folder to the project in a couple of ways. Select **create folder references for any added folders** and finish adding the folder to the project.
8. The `index.html` file in the `www` folder is where all the PhoneGap action begins. To detect the user's location, PhoneGap has the `Geolocation` object. To get the user's current position, we will use the `geolocation.getCurrentPosition` of the PhoneGap API.
9. In `index.html` generated from the Xcode PhoneGap Plugin, we modify the `onDeviceReady` function to call the `geolocation.getCurrentPosition` function when the device is ready to listen for the PhoneGap specific API calls. The code for which is `navigator.geolocation.getCurrentPosition(onSuccess, onError)`; where `onSuccess` is the callback function when the `getCurrentPosition` succeeds in execution and the `onError` fires when the `getCurrentPosition` returns errors while executing.

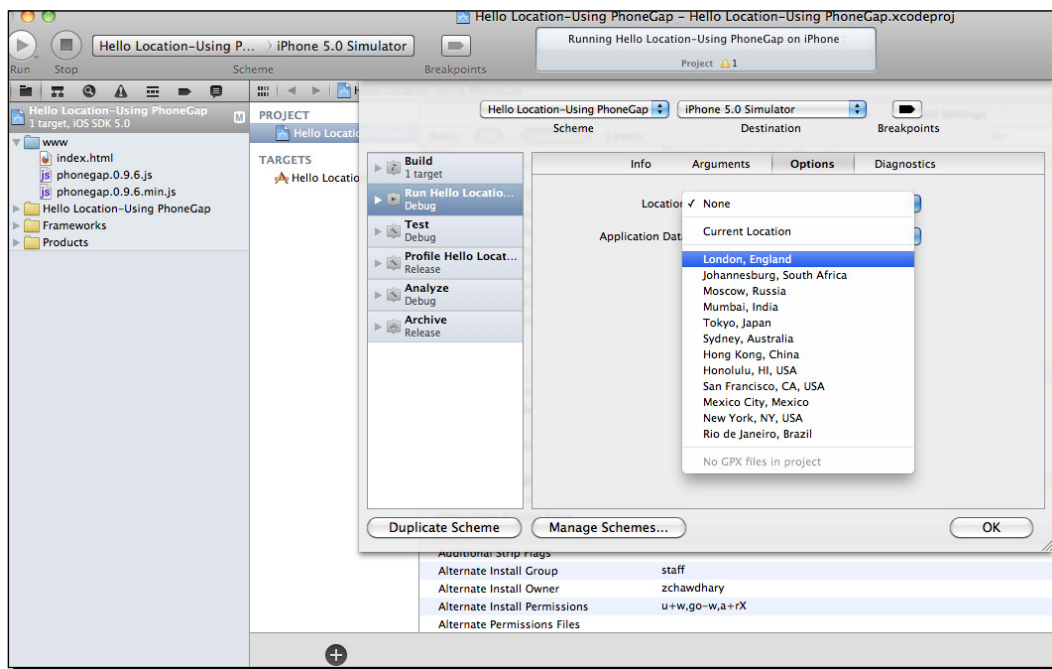
- 10.** Now, we define the `onSuccess` and `onError` functions, as shown in the following code snippet:

```
function onSuccess(position) {
    document.getElementById('latitude').innerHTML =
        position.coords.latitude ;

    document.getElementById('longitude').innerHTML =
        position.coords.longitude ;
}

function onError(error) {
    alert('message: ' + error.message + '\n');
}
```

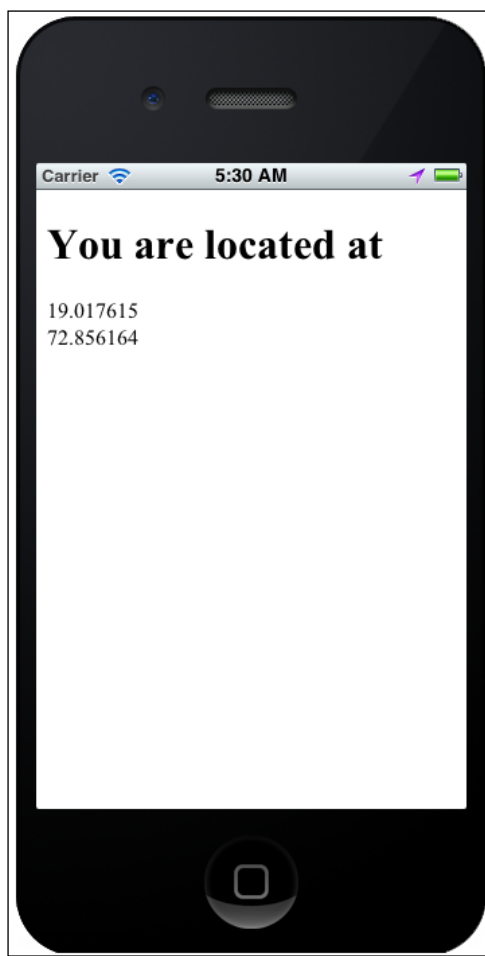
- 11.** Let's use the iOS 5 Location Simulation in our app! Go to **Product | Edit Scheme** in Xcode, with your Hello Location project open. From the **Run | Options** Settings Pane, select the location you want to simulate, in our case, I choose Mumbai.



- 12.** Modify the `index.html` `<body>` tag with the following content, which will receive the location values from the preceding `onSuccess` function:

```
<h1>You are located at </h1>
<div id='latitude'>Latitude Not Detected.</div>
<div id='longitude'>Longitude Not Detected</div>
```

- 13.** Run your app in the iPhone 5.0 Simulator. You will be presented with the latitude and longitude values of your simulated location, in this case Mumbai, as shown in the following screenshot:



What just happened?

We wrote a simple PhoneGap application using the Xcode plugin for PhoneGap and simulated the location using the new location simulation feature in Xcode 4.2 and iOS 5. The `index.html` file is the main file that runs in a PhoneGap application. When the `index.html` loads, we initialized PhoneGap framework on page load using `<body onload="onBodyLoad()">`, the `deviceready` is the PhoneGap function which is called when PhoneGap loads successfully. Next, we fire the `geolocation.getCurrentPosition` PhoneGap method to talk to the device hardware and get us the device's current position. This is returned via two functions, `onSuccess` and `onError`, which are pretty self-explanatory. The complete code for this application can be found on the book page at PacktPub's website <http://www.packtpub.com/iphone-location-aware-apps-beginners-guide/book> – in a project titled *Hello Location-Using PhoneGap*.

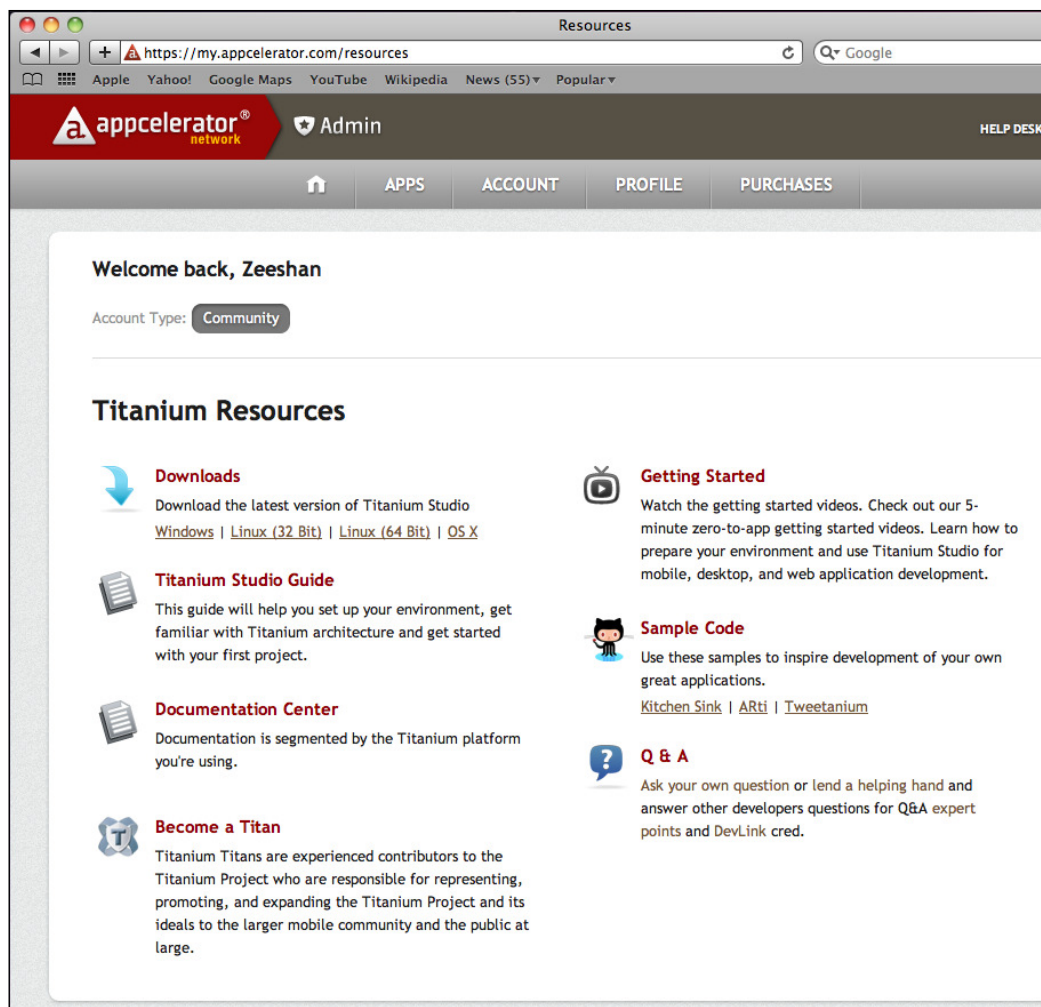
PhoneGap supports a JQuery Mobile JavaScript framework and Sencha Touch, which provides ready-to-use mobile-specific API calls such as Swipe, Touch, Zoom, Tap, Pinch to Zoom, Shake, and so on. The combination of PhoneGap and any of these frameworks makes mobile app development with HTML5 a breeze. Look no further for more information on PhoneGap because PacktPub has a dedicated book for the same; visit <http://www.packtpub.com/phonegap-beginners-guide/book> to read more and buy the book.

Time for action – using Titanium Appcelerator for building the Hello Location app

While PhoneGap is a good tool to build cross-platform mobile apps using HTML5 and CSS3 standards, the drawback of PhoneGap is that it is still a web-app wrapped in a native app. So, basically, it is non-native mobile app development. The controls and UI are all HTML-based. You don't get the native app look-and-feel for PhoneGap-based apps.

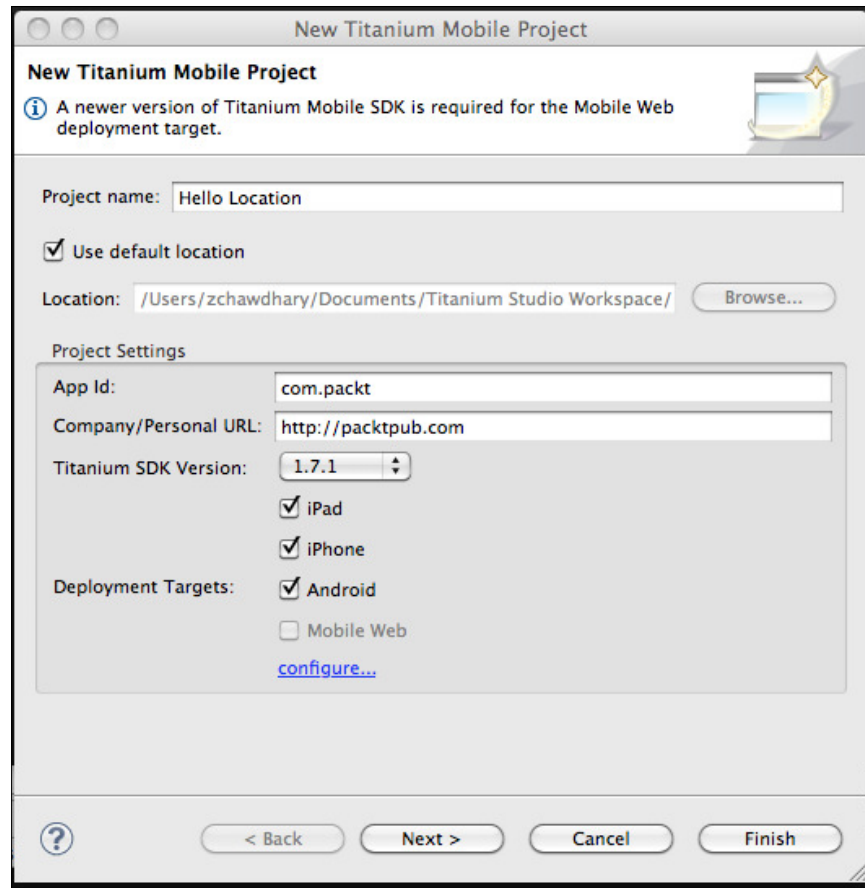
This is where Titanium Appcelerator excels; it converts your HTML5/CSS3/JS app into a complete native app through its extensive compilation and optimizations process. So the end result is a native UI look-and-feel for your app. As it is compiled to the device's architecture, it also performs faster. So let's get started with Appcelerator!


1. Go to <http://www.appcelerator.com/> and download the Titanium Studio community version for free (version 1.0, as of this writing). You will need to create an account at appcelerator.com before downloading. Select the Mac OS X version, as shown in the following screenshot:



2. Double-click on the downloaded `Titanium Studio.dmg` and drag-and-drop the Titanium Studio folder into your Applications folder on your Mac. The Titanium Studio is an Eclipse such as IDE that has all the plugins necessary for mobile development. Run the `TitaniumStudio` executable to start the IDE. On Launch, you will be presented with a Dashboard having links to examples, documentation, blog posts, and so on.

3. To create a new mobile project with Titanium Studio, go to **File | New | New Titanium Project**, as shown in the following screenshot. Deselect the **Create project from Template** checkbox.



 The Resources directory within the project folder holds the code and other files. The `app.js` file is the entry point of your Titanium application.

4. We begin by creating a new Window by using the `Titanium.UI.createWindow` method, which is part of the UI API provided by Titanium.

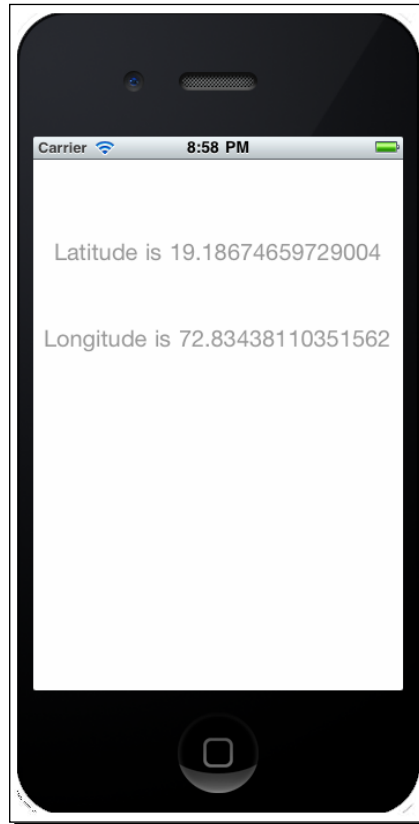
```
var win = Ti.UI.createWindow();
```
5. Next, we create a view using the `Ti.UI.createView` method

```
var view = Ti.UI.createView({backgroundColor:"white"});
```
6. We then add the view to the Window object and open it.

```
win.add(view); win.open();
```
7. Next, we call the `Ti.Geolocation.getCurrentPosition` method to determine the location of the user and the latitude and longitude values are displayed on the screen by adding two labels to the view created in step 6. Now the complete code for `app.js` is as follows:

```
var win = Ti.UI.createWindow();
var view= Ti.UI.createView({backgroundColor:"white"});
win.add(view);
Ti.Geolocation.getCurrentPosition(function(e) {
  if (e.error) {      Ti.API.error('geo - current position' +
    e.error);      return;    }
  var latitude = e.coords.latitude;
  var longitude = e.coords.longitude;
  var label1 = Titanium.UI.createLabel({
    color:'#999',
    text:'Latitude is '+latitude,
    font:{fontSize:20,fontFamily:'Helvetica Neue'},
    width:'auto', bottom:300});
  var label2 = Titanium.UI.createLabel({
    color:'#999',
    text:'Longitude is '+longitude,
    font:{fontSize:20,fontFamily:'Helvetica Neue'},
    width:'auto',      bottom:150}
  );
win.add(label1);
win.add(label2);
});
win.open();
```

8. Your app should look similar to the following on the iPhone Simulator:



What just happened?

Titanium Appcelerator follows a structured programming (or top-down programming) approach. We begin the app by creating a `Main Window` derived from the Titanium UI framework (`Ti.UI`) and then adding a view to the Window, having three labels to display the content. The `GeoLocation` values are obtained via the Titanium `GeoLocation` framework and eventually passed onto the labels for display.

The beauty of JavaScript programming is that you do not have to worry about type conversion, so the `GeoLocation` values obtained can be easily applied to the labels with minimum code and without worrying about the data type of the `GeoLocation` values obtained.

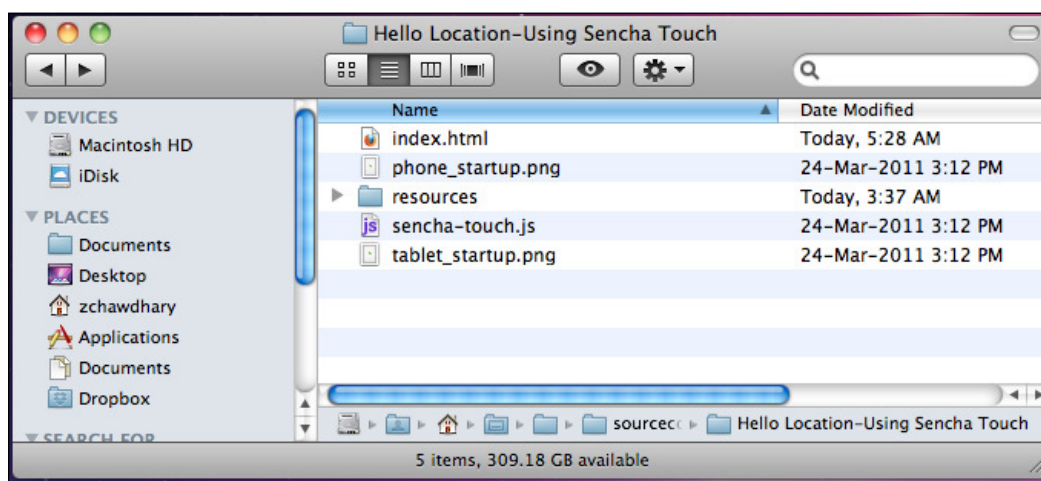
As with PhoneGap, PacktPub also has a book dedicated to Appcelerator Titanium; head to <http://www.packtpub.com/appcelerator-titanium-mobile-applications-development-for-smartphone-iphone-android-cookbook/book> to buy the book.

Time for action – Hello Location with Sencha Touch

Sencha Touch is a mobile web app framework, developed and maintained by the company behind the Ext JavaScript framework. It is built with HTML5, CSS3 with inbuilt data integration that allows developers to easily bind Data from XML, and JSON for HTML visual components. Sencha Touch also has enhanced touch events or swipe, double tap, pinch and zoom, rotate – all done with JavaScript.

Sencha Touch works very well with PhoneGap. It can also work for standalone HTML5 web apps.

1. Go to <http://www.sencha.com/> and download the Sencha Touch 1.1.0 web framework. Unzip the `sencha-touch-1.1.0.zip` folder.
2. You don't need to use all the files from this unzipped folder. Create a new folder for your application, let's say `Hello Location-Using Sencha Touch`, and copy `resources` and `sencha-touch.js` from the Sencha installation folder to this new folder.
3. You can copy the startup splash images from any of the examples. This can be used for your iPhone and iPad Builds. Copy the `phone_startup.png` and `tablet_startup.png` to the `Hello Location-Using Sencha Touch` folder.
4. Lastly, create a new `index.html`, which will hold your application code. Make sure your directory structure looks similar to the following screenshot:



5. In index.html, add the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello Location</title>
    <link href="resources/css/sencha-touch.css"
      rel="stylesheet" type="text/css">
    <script src="sencha-touch.js"></script>
    <script type="text/javascript"
      src="http://code.google.com/apis/gears/gears_init.js">
    </script>
    <script
      src="http://maps.google.com/maps/api/js?sensor=false">
    </script>
    <style>
      card1, .card2 {
        background-color: #376daa;
        text-align: center;
        color: #204167;
        text-shadow: #3F80CA 0 1px 0;
        font-size: 22px;
        padding-top: 100px;
      }
    </style>
  </head>
  <body>
    <script type="text/javascript">
      var latitude = "...";
      var longitude = "...";

      Ext.setup({
        tabletStartupScreen: 'tablet_startup.png',
        phoneStartupScreen: 'phone_startup.png',
        icon: 'icon.png',
        glossOnIcon: false,
        onReady: function()
        {
          var geo = new Ext.util.GeoLocation({
            autoUpdate: false,
            listeners: {
              locationupdate: function (geo) {
                latitude = geo.latitude;
                longitude= geo.longitude;
              }
            }
          });
        }
      });
    </script>
  </body>
</html>
```

```

        document.getElementById('latitude')
            .innerHTML=latitude;
        document.getElementById('longitude')
            .innerHTML=longitude;
    },
    locationerror: function (    geo,
    bTimeout,
    bPermissionDenied,
    bLocationUnavailable,
    message)
    {
        if(bTimeout){
            alert('Timeout occurred.');
```

}

```

        else
        {
            alert('Error occurred.');
```

}

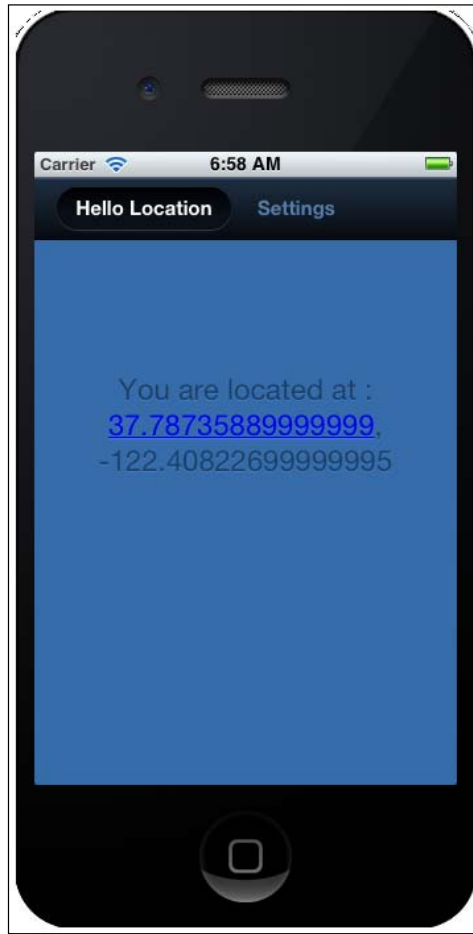
```

    }
}
});
geo.updateLocation();

new Ext.TabPanel({
    fullscreen: true,
    type: 'dark',
    sortable: true,
    items: [{
        title: 'Hello Location',
        html: "You are located at :
        <span id='latitude'>"+latitude
            + "</span>" + ",<br><span
            id='longitude'>"+longitude+"</span>" ,
        cls: 'card2'
    },
    {
        title: 'Settings',
        html: 'Will be added soon!',
        cls: 'card2'
    }
    ]
});
});
</script>
</body>
</html>

```


6. Now let's create a new PhoneGap Application and wrap this code in a PhoneGap application.
7. Once again, you will find the code on the book's website at <http://packtpub.com>. Here is how the application will look when run on the iOS simulator (note that iOS location simulation was used here as well):



What just happened?

We used the Sencha Touch library to create a `geolocation` object using the Sencha `Ext.util.Geolocation` package to determine the user's location and pass it on to a tab panel widget.

The same HTML code also works well on the browsers supporting the W3C GeoLocation API specifications. You can learn more about Sencha Touch at <http://www.sencha.com/>; don't forget to play around with the kitchen sink and sample apps to get a feel of the power of HTML5.

Exploring location-based SDKs/APIs

In the last four to five years, a lot of location-based startups have launched their products. Some of them have become a rage and a benchmark for current and future location-based applications. Foursquare, Gowalla, Yelp, and Wcities are some examples of companies that are looked upon for location-based places information.

On the entertainment side, Eventful, Last.fm, Wcities, Plancast, and Zvents are companies that provide location-based events and entertainment guides. For Movies, Fandango and Rotten Tomatoes are some good examples. Groupon and BView are good location-based deal providers. Another niche location-based company is SeeClickFix.com for providing social-governance functionality.

Almost all location-based companies these days have a location-based API to provide application developers with a geographical application interface to integrate these services into their own apps/websites.

Some companies, such as SimpleGeo, Factual, Location Labs, and SkyHook wireless, also provide the backend infrastructure necessary for startups to create and launch their products faster into the market, using their location expertise, often in a cloud-based **Software-as-a-Service (SAAS)** model. Often these companies provide an iOS/Android SDK, as well as RESTful APIs to integrate their services in third-party developer applications. Let's explore some popular location-based SDKs and APIs and understand how we can use them in our iOS applications.

Some open source projects such as openstreetmaps.org and creative commons-based GeoNames.org provide excellent community-driven efforts for Maps and Geo-Tagged information. We will use them soon, so keep reading.

Foursquare

Foursquare is a location-based social network that incorporates gaming elements and techniques for users and venue owners to create a new experience. Users **check-in** to a **venue** (bar, hotel, mall, airport lounge) and share their location with friends. The number of check-ins drive the user's badge from being a newbie to Mayor for a particular venue, the Mayors of these venues are often given free beer, pizzas, or hotel stays for socially promoting their venues on foursquare and among the user's friends.

Foursquare is available on the iPhone, Android, Blackberry, and other platforms including Symbian. On the developer side of things, foursquare has a rich API (V2 as of now) that provides a lot of functionality through the Web Service channel. There is also an Objective-C library developed by the foursquare API community members. We are concerned with the HTTP-based API calls, as we want to have better control and also learn as we build the same.

A visit to the foursquare developer's page at <https://developer.foursquare.com/> opens up three different APIs for developers, which are as follows:

1. Foursquare API V2
This is the generic HTTP-based and OAuth2-only RESTful API that any web, mobile app can use.
2. Foursquare Venues Project (in Beta)
This is intended to be used as a location database.
3. Client Resources
URIs for deep linking into the native iPhone app, so your iPhone app can link to a foursquare venue and clicking on it will open the foursquare iPhone app.

A typical use case for foursquare API consumption would be as follows:

- ◆ Search for Venues Nearby (using device location) or by Name
- ◆ Display the Venue lists and Venue details on selection
- ◆ Allow users to check-in to that venue and provide an interface for the users to leave tips (comments) or upload images
- ◆ Add photos and friends

A list of functions supported can be found at
https://developer.foursquare.com/docs/index_docs.html

Gowalla

Gowalla is quite similar to foursquare, with the addition of Trips, better eye candy and real-time updates for Spots (read Venues) via XMPP or PubSubHubbub protocols that follow a push pattern for Venue updates/check-ins.

Gowalla API also provides a Custom URL Scheme for direct linking within your iPhone app. This works like hyperlinks, but instead of webpages, it opens a pre-determined application; in this case, the Gowalla iPhone app, foursquare, works the same way as well.

Eventful and Last.fm API – some music is always good

Now let's move our focus to events and music APIs, namely, **Eventful** and **Last.fm**, which provide events and a music guide by location.

Eventful provides an extensive concert and event API via <http://api.eventful.com>, which allows third-party developers to:

- ◆ Integrate the rich and unique events into their apps
- ◆ Create and edit new events via the API
- ◆ Add images and comments
- ◆ Create, edit, or modify a venue
- ◆ Search for venues and events
- ◆ Get user information: user-created venues, events, events attended by the user
- ◆ Add/edit artists' information
- ◆ Get a list of event categories
- ◆ Search for demands and get the details of a demand

The demand feature is a unique concept for Eventful, which can be used by consumers to demand that their favorite artists come to their cities. The Events content is tagged in the following ways:

1. Events are tied to a location with the latitude and longitude, and mapped to a Venue(s) in a city.
2. Events are tied to Artists (if available).
3. Events are categorized into specific pre-defined categories, such as concerts, jazz, dance, and so on, however, with support for extra tags.
4. Events details include start date, end date, time of the event, ticket price, and so on.

The API is quite extensive. We will cover more details as we build our apps in the coming chapters.

Last.fm (<http://www.last.fm/api/>) is a music recommendation service that also includes events, artists' information, and so on, but with a good recommendation algorithm. In addition to the functionalities supported by Eventful API, last.fm also provides API calls for Album, Chart, Library, Playlist, and Tracks objects. It is more extensive and also supports XML-RPC in addition to REST. Our interest lies in the Geo API Object that has provisions for the following:

- ◆ Retrieving events by location
- ◆ Retrieving top artists by location
- ◆ Retrieving top tracks by location

We will explore more when we build the events app in *Chapter 6*, so stay tuned.

Still more tools: SimpleGeo and Factual

SimpleGeo (<http://simplegeo.com>) and Factual (<http://factual.com>) are Software-as-a-Service (SAAS) model-based products that charge developers based on the Data/API consumption. Both offer location-based services. SimpleGeo extends its location service and offers three services. They are as follows:

1. GIS-based cloud storage of location data
2. Contextual information of location such as weather and geographic boundaries
3. Location-based places information such as Foursquare and Gowalla

Factual, on the other hand, works on the model of tables – sets of data clubbed together by location. So you can query for *A list of all Restaurants in San Francisco* and get a table as the result. Now this table can be embedded in a web app, or the factual iPhone SDK, or can be read as a REST JSON output, giving the developers a lot of flexibility.

Other Notable APIs – YQL and Location Labs

Yahoo! Query Language (YQL) provides a lot of small and quick location-based API calls, including weather and events from upcoming.org, as well as finding out the geographic details of a city, country, and its bounding boxes. Almost all the Yahoo! Geo Technologies (<http://developer.yahoo.com/geo/>) are available in YQL under the Geo tables heading. Also available in YQL is the Yahoo! Local Search API that works only for the USA for now.

Location Labs is known for its Geo Fencing Product. The Location Labs iPhone SDK for GeoFencing is provided as a 30 day trial. However, with the advent of iOS 5 location reminders, the need for a third-party Geo Fencing is being weeded out. More on Location Labs at <https://geofence.locationlabs.com/index.html>.

Pop quiz – so you think you can Xcode

1. Can Xcode 4 and higher work on the PowerPC architecture?
 - a. Yes
 - b. No
2. What is the default compiler in iOS 5 SDK?
 - a. GCC
 - b. LLVM

Summary

In this chapter, we learned about Xcode 4 and iOS 5 and how to be futureproof with ARC memory management, as it is a major change in iOS5 SDK. Specifically, we covered:

- ◆ Xcode – installation and new features
- ◆ iOS 5's new features
- ◆ Writing a simple `Hello Location` app in Xcode and Objective C
- ◆ HTML5: Its beauty and ease of development
- ◆ PhoneGap, Appcelerator, and Sencha Touch – the Hello location app done in three different ways
- ◆ We also discussed foursquare, Gowalla, Last.fm, Eventful, and other important Location APIs briefly

We are now ready to dive deep into Xcode and Core Location, so keep turning the pages.

3

Using Location in your iOS Apps— Core Location

Having played around with an introductory location app – Hello Location, let us dive deeper into the iOS library that handles location – Core Location. Core Location provides all the delegations and functions to detect location via GPS, Wi-Fi, or Cell ID. However, the good part is that the end user need not worry about which location method to employ; the Core Location library handles it for the user.

In this chapter, we will examine the following topics:

- ◆ Overview of Core Location
- ◆ Starting and using the location service
- ◆ Receiving location updates
- ◆ Remembering a user's location with the core data
- ◆ Extending the Hello Location app
- ◆ Building an events app using the `eventful.com` and `Last.fm` API
- ◆ Building a Local Search app using Foursquare API
- ◆ Understanding the features of iOS 5 Location Simulator

So let's get on with it...

Core Location framework – an overview

The Core Location framework in the iOS SDK is an asynchronous API that uses delegation to report location information from the iOS device. Along with location information, Core Location also reports the Heading information (Heading here implies the direction in which a device is pointed), as well as allowing you to define geographic regions and monitor when you cross those regional boundaries.

Core Location implements all the three methods of location detection: GPS, Wi-Fi, and Cell Tower Triangulation. The developer can control location detection by only specifying the accuracy needed. Core Location then decides internally on which approach to use for actual location detection.

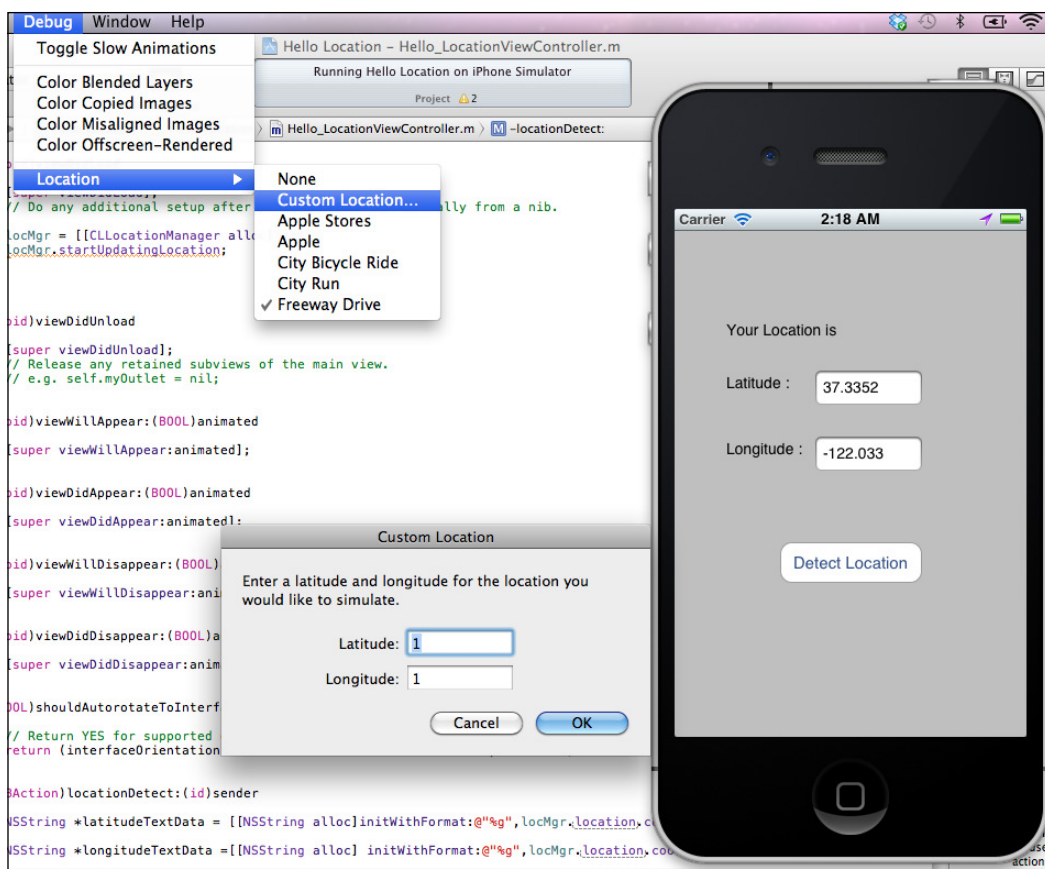
When creating an application that uses the Core Location framework, you need to first add it to your project in Xcode and include the `<CoreLocation/CoreLocation.h>` header files in your `.m` or `.h` file.

The new iOS SDK 5 and Xcode 4.2 includes a nifty location simulation and debugging capability. We have already visited the Location Simulation in the Hello Location examples in *Chapter 1*. Now we will understand how the location debugging feature helps you test your app by using different location values when your app is actually running in the simulator or a connected device.

Time for action – location debugging

Open the Hello Location application we created in *Chapter 1* and run the application.

1. If you are running your app on the iOS Simulator, then go to the **Debug | Location Menu** option where you can simulate multiple location inputs for your app, as shown in the following screenshot:



2. If you are running your app on a connected device, you need to go to the **Product | Debug | Simulate Location** menu option in the Xcode 4 menu bar.
3. Try changing to different locations and testing the app by clicking the **Detect Location** button in the app.

Location data is reported in your application via the Core Location's Delegate object, `CLLocationManagerDelegate`. Based on the location service type used in your app, the corresponding Core Location Delegate function has to be implemented by your application to catch the appropriate location change event. We will look at it as we inspect the different Core Location Services.

What just happened?

We simulated location information on our iPhone, using the new feature of "Location Simulation" in iOS 5 SDK and Xcode 4.2. This new feature helps us analyze our app's behavior in different locations. In the preceding example, we changed the location values one-by-one and clicked on the `Detect Location` button to echo the geo co-ordinates of our labels. iOS 5 Location simulation includes significant location updates, region monitoring, and continuous location updates via the GPX file support (a GPX file is an XML file format that contains a sequence of Geo Coordinates, typically for Tours or Navigational purposes).

Core location services

The Core Location framework provides the following services:

- ◆ Standard location
- ◆ Significant change
- ◆ Region monitoring
- ◆ Geocoding and reverse Geocoding – `CLGeocoder` (Added in iOS 5 SDK)
- ◆ Direction using heading information

Standard location

Standard Location is the latitude and longitude information retrieved from Core Location. The Core Location Manager (the `CLLocationManager` object in the iOS SDK) returns this information in the `CLLocation` object. Recall from the Hello Location example in *Chapter 2*, where we used the following code to retrieve the latitude information (which is the most common way to grab a user's location).

```
NSString *latitudeTextData = [[NSString alloc] initWithFormat:
    @"%g", locMgr.location.coordinate.latitude];
```


Here the location object is an instance of the `CLLocation` object that contains the latitude and longitude variables. Standard location service with the Core Location Manager is started with the `startUpdatingLocation` function. You can tell the Location Manager to stop updating the location with the `stopUpdatingLocation` function.

`distanceFilter` and `desiredAccuracy` are two properties that define how often you will receive the location updates and how much accuracy (in meters) is required by your app.

With `distanceFilter`, you will receive location information if the device has moved distance equal to or more than the value specified in the `distanceFilter` property.

Accuracy of the location detections can be chosen from the following `desiredAccuracy` values:


Constant Value	Definition
<code>kCLLocationAccuracyBestForNavigation</code>	Standard Accuracy intended for Navigational apps
<code>kCLLocationAccuracyBest</code>	Use highest accuracy available
<code>kCLLocationAccuracyNearestTenMeters</code>	10 meters accuracy
<code>kCLLocationAccuracyHundredMeters</code>	100 meters accuracy
<code>kCLLocationAccuracyKilometer</code>	Accuracy up to 1 kilometer
<code>kCLLocationAccuracyThreeKilometers</code>	Accuracy up to 3 kilometers

 Use/Try to use the lowest accuracy possible (the lowest accuracy your application can work with) to avoid more battery power consumption.

Significant change

With the Core Location framework, you can also request for location updates having significant location value changes only. This method provides excellent power saving options, as well as the ability of the device to send location updates even when your application is not running. This method uses Cellular Radio to detect the device location.

To use the significant change location service in your app, you need to use the `startMonitoringSignificantLocationChanges` and `stopMonitoringSignificantLocationChanges` functions.

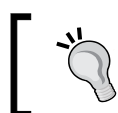
 Core Location framework caches the location data, it is a good idea to get the timestamp on the measurement objects to make sure your application receives the correct and updated location information.

Region monitoring

With the Region monitoring services, you can define geographical boundary-based tracking for your apps. Consider a simple example of a Weather app that can use Region Monitoring to detect the user's location based on physical boundaries and alert them if they cross a particular boundary, for example, if a user crosses a San Francisco city boundary towards San Jose, the app can trigger a boundary alert for the user and show him the new San Jose weather information.

The `startMonitoringForRegion` and `stopMonitoringForRegion` methods of the location framework are used to start and stop region monitoring in your application. Boundary entering and exiting are monitored by `locationManager:didEnterRegion` and `locationManager:didExitRegion`. Boundary crossing detection also requires an accuracy factor to determine the crossing factor needed to trigger the alert. This is done by the `startMonitoringForRegion:desiredAccuracy` method.

As with the Significant Change service, Region monitoring also works even if your application is not running. The most important part is that you need to register the Regions to be monitored with the device using the `monitoredRegions` property.



Use smart programming techniques to shut down location services when not required in order to conserve battery power. Another good idea is to turn off location if accuracy does not improve over a course of time.

Geocoding and reverse Geocoding – CLGeocoder

The `CLGeocoder` along with the `CLPlacemark` object provide the **Geocoding** and **Reverse Geocoding** functions in the Core Location framework. Note that these are new APIs added in the iOS 5.0 SDK.

The `MKReverseGeocoder` from the MapKit Framework (more on it in *Chapter 4*) has been deprecated. The `CLGeocode` object now handles the same. `CLGeocode` features as follows:

- ◆ Requests are asynchronous and support only one operation per request
- ◆ Supports multiple languages
- ◆ Supports Forward and Reverse Geocoding
- ◆ Does not require results to be displayed on a map
- ◆ Worldwide coverage

Geocoding is done by any of the following three methods:

1. `geocodeAddressString:completionHandler`: Geocodes a simple string, for example, *Mountain View, San Francisco*.
2. `geocodeAddressString:inRegion:completionHandler`: Geocodes a specified string using regional information. Think of this as searching for the String *Market Street* in region *San Francisco*.
3. `geocodeAddressDictionary:completionHandler`: Geocodes a specified address dictionary. This is a more structured geocoding request, usually providing the Address Street, Address City, and Address State fields in the `AddressBook` format. The following is an example code snippet for this function:

```

CLGeocoder *geocoder = [[CLGeocoder alloc] init];

NSDictionary *address=[NSDictionary dictionaryWithObjectsAndKeys:
    @"32 Lincoln RoadRoad",kABPersonAddressStreetKey,
    @"Birmingham",kABPersonAddressCityKey,nil ];

[geocoder geocodeAddressDictionary:address
    completionHandler:^(NSArray *placemarks, NSError *error)
    {

        for(CLPacemark *placemark in placemarks)
        {
            NSLog(@"Placemark %@",placemark);
        }
    }];

```

Don't forget to add the AddressBook framework in build phases in Xcode and import the header files required in your `Hello_LocationViewController.m` file from the Hello World example.

```

#import <AddressBook/AddressBook.h>
#import <AddressBook/ABPerson.h>

```

You can find the code at the book's page at <http://www.packtpub.com/iphone-location-aware-apps-beginners-guide/book> – in a project called Hello Location – Geocode. Run the application and click on the **Detect Location** button and observe the Debug Console in Xcode; you should see an output as follows:

```

Placemark 32 Lincoln Road, Solihull, England, B27 6, United Kingdom @
<+52.44378245,-1.81094734> +/- 100.00m

```

This is the result of the geocoding, along with the accuracy of 100 meters.

Reverse Geocoding is handled by the `reverseGeocodeLocation:completionHandler` method in the `CLGeocoder` class.

The `CLPlaceMark` object is returned for both Forward and Reverse Geocoding.

Direction using heading

Heading information in the Core Location service signifies the direction in which the device is oriented. This information is very critical for augmented reality, navigation, and gaming applications. The direction in which a device is pointing, reported by iOS devices with a magnetometer is known as **heading**, while direction in which the iOS device is moving, reported by the GPS hardware, is known as course.

The `CLHeading` object holds the heading data reported by the Location Manager. The `startUpdatingHeading` method in the Location Manager is used to start the heading update process, while `stopUpdatingHeading` is used to stop it.

The `CLHeading` object contains the following properties:

Property	Description
<code>magneticHeading</code>	Magnetic heading in degrees, relative to magnetic north
<code>trueHeading</code>	Heading in degrees, relative to the true north
<code>headingAccuracy</code>	The difference between the reported heading and true magnetic heading
<code>timestamp</code>	Time at which the heading was obtained
<code>x</code>	X-Axis difference from the magnetic fields tracked by the device
<code>y</code>	Y-Axis difference from the magnetic fields tracked by the device
<code>z</code>	Z-Axis difference from the magnetic fields tracked by the device

Core Location Manager – `CLLocationManager`

The `CLLocationManager` class controls all the Core Location services discussed above. The Core Location Manager class, `CLLocationManager`, handles all the location and heading-related events for your application.

Location and heading updates are delivered to associate delegate objects, which must conform to the `CLLocationManagerDelegate` delegate protocol.

To assess the different location services available on an iOS device, Core Location manager exposes the following methods in the `CLLocationManager` class:

- ◆ `locationServicesEnabled`
- ◆ `authorizationStatus`
- ◆ `significantLocationChangeMonitoringAvailable`
- ◆ `headingAvailable`
- ◆ `regionMonitoringAvailable`
- ◆ `regionMonitoringEnabled`

Time for action – checking for location service availability

Before we start using the iOS location framework, it is important to know whether location services are enabled on the user's device or not. If location is off, then we can prompt the user to switch it on.

Let's reuse the `Hello Location` example to check for location service availability:

1. Open the `Hello Location` example and modify the `viewDidLoad` method in the `Hello_locationViewController.m` file to look like the following code snippet:

```
locMgr = [[CLLocationManager alloc] init];
if ([CLLocationManager locationServicesEnabled])
{
    locMgr.startUpdatingLocation;
}
```
2. You can manage your iOS device's location settings at **Settings | Location Services**. Starting with iOS 5, location services are not turned **ON** by default, but you can choose to enable/disable it from the main phone's set up screens. We have kept it **ON**, as can be seen in the following screenshot:

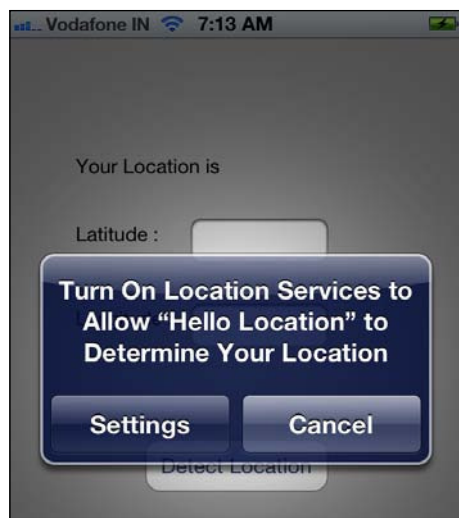


3. In iOS 5, you can also specify what system apps (apps that come inbuilt when you buy the iPhone) can access the location information. This is done via the **Settings | System Services** option.

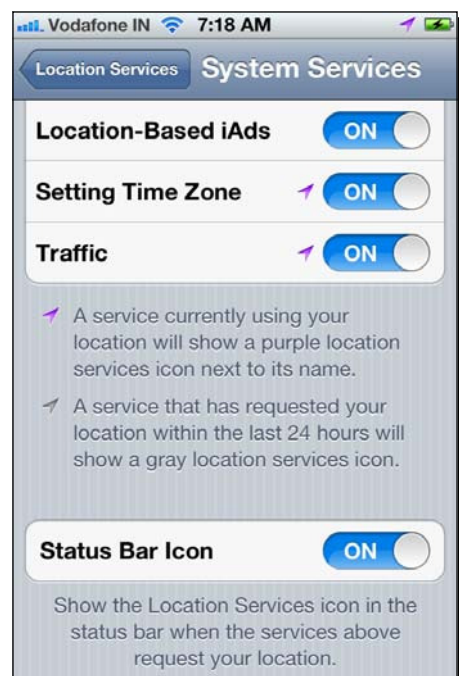


4. Let's turn it off to observe the Hello Location app's behavior now.
5. When you turn off the location settings and run the app, you will get a zero (0) value in the Latitude and Longitude text fields, when you click the **Detect Location** button.
6. Now modify the code, as shown in the following code snippet:

```
if ([CLLocationManager locationServicesEnabled]==TRUE)
{
    [locMgr startUpdatingLocation];
}
else
{
    [locMgr startUpdatingLocation];
}
```
7. Here, we call the Location Manager's `startUpdatingLocation` method, if location services are enabled or disabled.
8. If Location is enabled, then you get the location values as before. However, if Location is disabled and you still run the `startUpdatingLocation` method, then the application prompts you to enable Location Services from the **Settings** option, as follows:



9. With iOS 5, you can now easily identify what services and apps use location; from the **Settings | Location Services | System Services**, select the option **Status Bar Icon** and turn it **ON**. Now whenever an application or service uses location information, you will see a purple arrow in the header area of your iPhone, as shown in the following screenshot:



The source code for this example can be found at the books page at <http://www.packtpub.com/iphone-location-aware-apps-beginners-guide/book>, in a project titled Hello Location - Location Settings.



It is a good idea to reset your location warnings from the **Settings | General | Reset** option in your iOS, while testing your location-based applications.

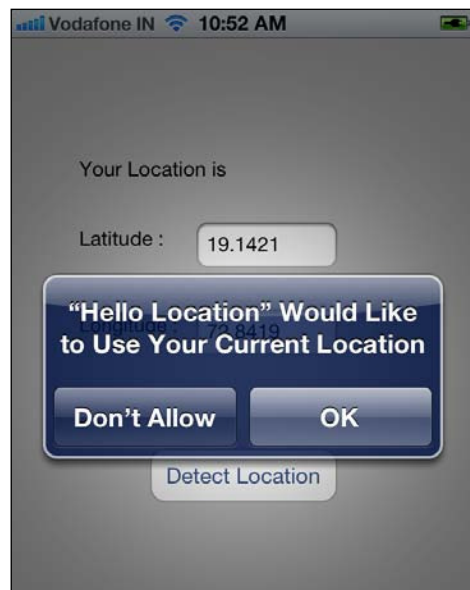
What just happened?

In this example, we continued on our **Hello Location** expedition and used the Location Manager object's `locationServicesEnabled` method to check if location services are enabled on the iPhone or not. If the location services are enabled, then we proceed to detect the location and allow the application to use the geocodes that were obtained. However, if the location services are not enabled, then we prompt the user to enable it via the **System Settings** and come back to our application.

User authorization

Having understood how Location Settings can be enabled/disabled on the iOS device, let's now move to User Authorization for Location.

So far, we have assumed that the user always allows the location pop-up in the application. For example, in our Hello Location application, you would see the first screen as follows:



However, what happens when the user clicks on **Don't Allow**?

Such explicit application authorization status can be obtained by using the `authorizationStatus` method of the `CLLocationManager` class. It returns any of the following statuses, depending on how the application is authorized to use location services:

Status	Definition
<code>kCLAuthorizationStatusNotDetermined</code>	The user hasn't made a choice yet
<code>kCLAuthorizationStatusRestricted</code>	The user is not authorized to use location services
<code>kCLAuthorizationStatusDenied</code>	The user has denied the use of location services for your application or all applications
<code>kCLAuthorizationStatusAuthorized</code>	The user has authorized your application for location services

Time for action – using Core Location with user authorization

Let's revise our Hello Location application with support for User Authorization. If the user has authorized your app for location, then the application will show the user's detected location; else we default it to Geocodes for San Francisco.

1. In your Hello Location application, open the `Hello_LocationViewController.h` file and add a `CLLocation` object `-userLocation` that stores the default San Francisco latitude and longitude values and an `NSString` object – a `message` that will be used to render custom user prompts based on the User Authorization level set by the user.


```
CLLocation *userLocation;
NSString *message;
```
2. Next, open the `Hello_LocationViewController.m` file and append the following lines in the `viewDidLoad` method:


```
userLocation = [[CLLocation alloc] initWithLatitude:37.33
        longitude:- 122.03];
message = [[NSString alloc] initWithString:@""];
```
3. We initialized the `userLocation` object with San Francisco's co-ordinates and created a new string object `message` to hold the User Authorization messages.

- 4.** Within the `viewDidLoad` method, we also check if location services are enabled or disabled:

```
if ([CLLocationManager locationManagerServicesEnabled]==FALSE)
{
    message = @"Location cannot be initialized.
               Please check settings";
    [locMgr startUpdatingLocation];
}

else if ([CLLocationManager locationManagerServicesEnabled]==TRUE)
{
    [locMgr startUpdatingLocation];
}
```

- 5.** Lastly, we modify the `locationDetect` function as follows:

```
- (IBAction)locationDetect:(id)sender
{
    latitudeText.text = @"0";
    longitudeText.text = @"0";

    if ([CLLocationManager locationManagerServicesEnabled]==TRUE)
    {
        if ([CLLocationManager locationManagerAuthorizationStatus]==
            kCMAuthorizationStatusNotDetermined)
        {
            message = @"User hasn't made a choice
                       yet. Defaulting to San Francisco";
            latitudeText.text = [[NSString alloc]
                                initWithFormat:@"%g", userLocation.coordinate.latitude];

            longitudeText.text = [[NSString alloc]
                                  initWithFormat:@"%g", userLocation.coordinate.longitude];
        }

        else if ([CLLocationManager locationManagerAuthorizationStatus]==
                  kCMAuthorizationStatusDenied)
        {
            message = @"User has denied use of location
                       services for your application or all
                       applications. Defaulting to San
                       Francisco";
            latitudeText.text=[[NSString alloc]
                              initWithFormat:@"%g", userLocation.coordinate.latitude];
            longitudeText.text=[[NSString alloc]
                               initWithFormat:@"%g", userLocation.coordinate.longitude];
        }
    }
}
```

```
else if([CLLocationManager authorizationStatus]==
    kCLErrorAuthorizationStatusAuthorized)
{
    message          = @"User has authorized your
    application for location services.";
    latitudeTextData = [[NSString alloc]
        initWithFormat:@"%g",
        locMgr.location.coordinate.latitude];

    longitudeTextData = [[NSString alloc]
        initWithFormat:@"%g",
        locMgr.location.coordinate.longitude];

    latitudeText.text = latitudeTextData;
    longitudeText.text = longitudeTextData;
}

else if([CLLocationManager authorizationStatus]==
    kCLErrorAuthorizationStatusRestricted)
{
    message          = @"Not authorized to use location
    services.Defaulting to San Francisco";
    latitudeText.text = [[NSString alloc]
        initWithFormat:@"%g", userLocation.coordinate.latitude];

    longitudeText.text = [[NSString alloc]
        initWithFormat:@"%g", userLocation.coordinate.longitude];
}
}
else
{
    if([CLLocationManager locationServicesEnabled]==FALSE)
    {
        message = @"Location cannot be initialized. Please check
        settings";
    }
}

UIAlertView *alert = [[UIAlertView
    alloc] initWithTitle:@"Location Authorization Tests"
    message:message
    delegate:self cancelButtonTitle:@"OK"
    otherButtonTitles:nil, nil];
[alert show];
}
```

What just happened?

We begin with checking if the location services are enabled or disabled in the `viewDidLoad` function. We start the Location updating Service nevertheless, as it will return the location information if location services are enabled and will prompt the user to enable location services if it is disabled on app load.

Next, on the button click function – `locationDetect`, we check for the various location services that User Authorization states and display location information accordingly. Note that if a location could not be retrieved, we have defaulted to San Francisco's co-ordinates, in this case, the user of `CLLocation` object. If Location is detected via the Location Manager, then we use the real device co-ordinates, change the `UITextView` values accordingly, and show an alert prompt with the User Authorization values obtained before.

Code for the sample can be downloaded from the book's website, from a project titled `Hello Location - User Authorization`.

The CLLocation object

The `CLLocation` object holds your location data; including the geographical co-ordinates (latitude and longitude) as well as the altitude. For iOS Devices with Navigation support, the `CLLocation` object also supports the **speed** and **course** property. Note, we have discussed the difference between course and heading before and it would be a good time to revisit it, if you haven't already.

An important property of the `CLLocation` object is the **timestamp** property, which lets us know the time at which location information was last fetched. The timestamp property can be used to make sure the device has the most updated value of the location. The timestamp property can constitute the **Smart Programming Technique** we discussed before to save a few location calls and conserve battery juice!

Latitude and longitude values are encased in the coordinate property, as seen in the `Hello Location` example code as well. Two other properties of the `CLLocation` object worth noting are `horizontalAccuracy` and `verticalAccuracy`. `horizontalAccuracy` is for getting information about the accuracy of the latitude and longitude values fetched, while `verticalAccuracy` provides accuracy information on the altitude.

The methods (functions) exposed by the `CLLocation` object are as follows:

Method	Description
<code>initWithLatitude:longitude:</code>	Initialize a location object with specified lat/lon pair values
<code>initWithCoordinate:altitude:horizontalAccuracy:verticalAccuracy:timestamp:</code>	Initialize a location object with lat/lon, altitude, horizontal and vertical Accuracy, along with the timestamp
<code>initWithCoordinate:altitude:horizontalAccuracy:verticalAccuracy:course:speed:timestamp:</code>	Initialize a location object with lat/lon, altitude, horizontal and vertical Accuracy, timestamp, along with course and speed values
<code>distanceFromLocation</code>	Calculates the distance to a destination location from the current location, in meters

Time for action – receiving location updates in your application

So far, in all our sample code, we have been using `Detect Location UIButton`, along with the `locationDetect` method to detect and use the location information from the device in our `Hello World` application. This was possible with the `startUpdatingLocation` method in the `CLLocationManager` class.

However, the common use case for location-based apps is that the location keeps updating in the background or when there is a significant change in location. The application should be able to catch it and notify the user for action to be taken.

The simplest way in which this can be done is by using the `locationManager:didUpdateToLocation:fromLocation` method of the `CLLocationManager` class.

Let's modify the `Hello Location` example to include the following method:

1. In your `Hello_LocationViewController.m` file, add the following method implementation:


```
(void) locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation *)newLocation fromLocation:
(CLLocation *)oldLocation
{
    NSString *newLatitude = [[NSString
        alloc] initWithFormat:@"%g", newLocation.coordinate.latitude];

    NSString *newLongitude = [[NSString
        alloc] initWithFormat:@"%g", newLocation.coordinate.longitude];
```



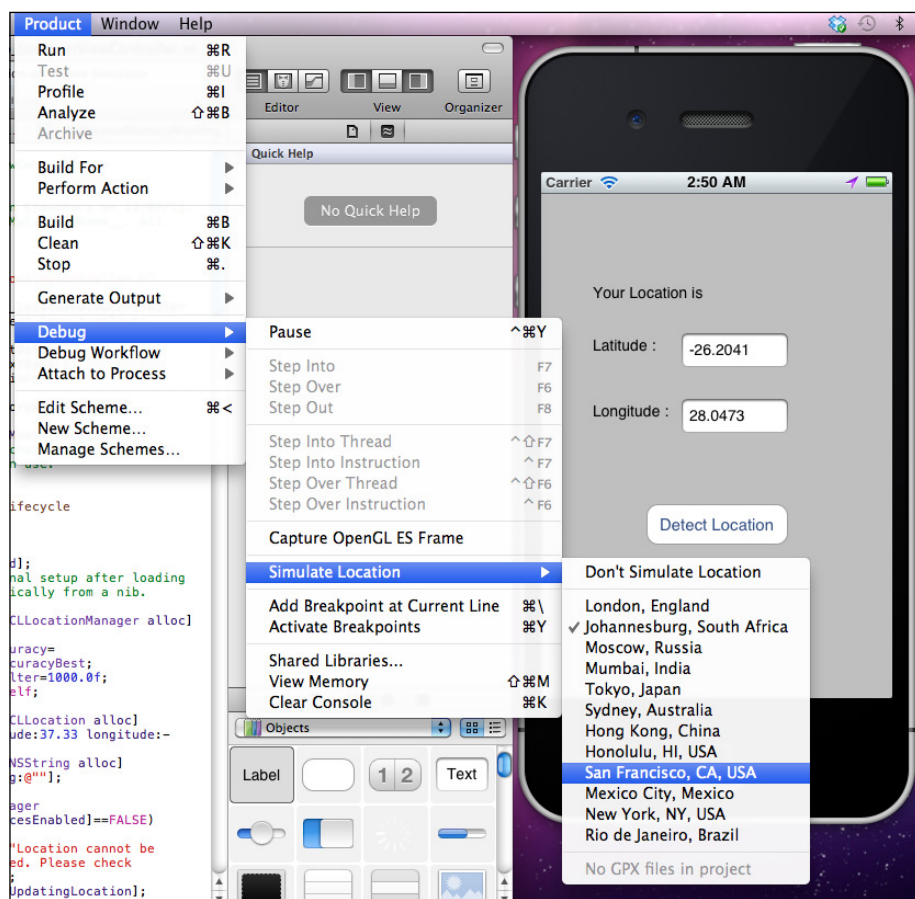
```
latitudeTextData    =    newLatitude;
longitudeTextData    =    newLongitude;

latitudeText.text    =    latitudeTextData;
longitudeText.text    =    longitudeTextData;
}
```

2. In the `viewDidLoad` method, change the `LocationManager` object we created before (`locMgr`) to the following:

```
locMgr    =    [[CLLocationManager alloc] init];
locMgr.desiredAccuracy= kCLLocationAccuracyBest;
locMgr.distanceFilter=1000.0f;
locMgr.delegate=self;
```

3. Run the application in the iOS Simulator and try changing the Simulated Location values from the **Product | Debug | Simulate Location** menu option.

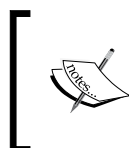


4. As soon as you change the simulated values in Xcode, you will see the values of the textfields objects change immediately.

What just happened?

1. We used the `didUpdateToLocation` method implementation of the location manager object to fire an event when new location information is available. If the location manager cannot find the location for whatever reason, then it fires the `didFailWithError` event.
2. We also created an `Info.plist` file and added the following properties to continuously monitor the location, even if the application is running in the background.

```
<key>UIBackgroundModes</key>
<array>
  <string>location</string>
</array>
```



Don't forget the battery life performance hit by using location services continuously in the background.

3. More on `info.plist` in *Chapter 7*; download the code for this example from the book's website, from the project titled `Hello Location - Location Updates`.
4. Note that we can also use the `startMonitoringSignificantLocationChanges` method from the Location Manager object to monitor location updates only when the location change is significant enough. This is done by monitoring the cell tower associated with the iPhone, as the user moves to a different location and the cell tower ID changes. It then becomes a significant location update call to the location manager. Here again, `didUpdateToLocation` is used to peruse the new location values obtained.

Time for action – boundary monitoring with Location Manager

Let's now move to the next Location Service provided by the iOS Location Manager – Region/Boundary Monitoring. We use the `CLRegion` class and its method, namely, `didEnterRegion` to monitor whether the user's position falls in the boundary.

1. Open the `Hello Location` project. In the `Hello_locationViewController.h` file, add the `CLRegion` definition as follows:

```
CLRegion *boundary;
```

- 2.** In the `Hello_LocationViewController.m` file, we create a circular boundary/region centered around San Francisco geo co-ordinates, with a radius of 1000 meters. We initialize the boundary variable as the following:

```
CLLocationCoordinate2D regionCords = CLLocationCoordinate2DMake
(37.78 , -122.408);

boundary = [[CLRegion
alloc] initWithCenter:regionCords
radius:1000.0f identifier:@"San Francisco"];
```
- 3.** Next in the `viewDidLoad` method, we tell the Location Manager to start monitoring the region using the `startMonitoringForRegion` method as follows:

```
[locMgr startMonitoringForRegion:boundary];
```
- 4.** To detect whether the device has entered the defined region, we implement the `didEnterRegion` method and alert the user in case he has entered the region (San Francisco boundary defined earlier)

```
- (void) locationManager:(CLLocationManager *)manager
didEnterRegion:(CLRegion *)region
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
@"You Entered San Francisco"
message:@"Welcome to San Francisco"
delegate:self cancelButtonTitle:@"OK"
otherButtonTitles:nil, nil];
    [alert show];
}
```
- 5.** When you run the application in the simulator and use location simulation to pass San Francisco co-ordinates to the application, you get an output as shown in the following screenshot:



What just happened?

We created a region of 1000 meters around the San Francisco Geo co-ordinates and monitored the user's device location against this. As soon as the user enters the specific region, an alert is displayed, welcoming the user to San Francisco.

For best results, start the application by setting the **Product | Setting | User Location** to say Moscow or Mumbai and then when running your application on the iOS Simulator or the iPhone, go to **Product | Debug | Simulate Location** and select San Francisco. You should see the alert on your device immediately.

The `didEnterRegion` and `didExitRegion` method of the `CLRegion` class are used to detect if the user's iPhone enters or leaves the region. This is the simplest form of Geo Fencing that can be accomplished by core iOS APIs.

Have a go hero – remembering a user's location with Core Data

As we have discussed so far, Location calls on the iOS device can be taxing on the battery. It is a good programming technique to store the user's last position on the device. It might be an application design requirement as well to store a user's location history, in case you are building a Travel Trip application or a Travelling Tour application.

Core Data allows iOS developers to store, retrieve, and manage their application's data in an object-oriented manner. Think of it as an **Object-relational mapping (ORM)** for iOS development.

Core Data is based on the Model View Controller software development methodology. Let's look at the key building blocks of Core Data:

- ◆ **Managed-object model:** Similar to "Tables" in an RDBMS Schema
- ◆ **Managed-object context:** Connector between the developer and the managed objects
- ◆ **Persistent object stores:** A single File or External Data store
- ◆ **Managed object:** A "Row" in a table (in the RDBMS context)

iOS 5 brings some new features in Core Data, namely:

- ◆ iCloud integration
- ◆ Incremental store
- ◆ Data protection (with encryption)
- ◆ Concurrency
- ◆ `UIManagedDocument`

A complete analysis of Core Data framework is beyond the scope of this book. However, let's use the iOS Core Data framework to store the user's location history on the iOS device. Try out building an application that uses Core Data to store the Location info.

Extending Hello Location for nearby events

The previous chapter introduced us to the Eventful events and also discussed its supported features. Now, let us dive into building a location-enabled Events app using the Eventful API (which is a third-party API. Read the API terms before you proceed with using it in your application). Let's explore the API a bit before we begin coding.

Important things to know before we begin

1. An API key: Register for one at <http://api.eventful.com/signup>
2. XML Parsing know – how to consume XML with the `NSXMLParser` class in iOS SDK
3. Fetching nearby Events using the **Events | Search** method from the API
4. Fetching the XML from the API and using it in our app – Using the `NSURLConnection` class

Time for action – extending Hello Location for nearby events

Let's begin writing our app now. Using the Hello Location application as the base of our project, we add a `UITableView`, which will be used to show the nearby events.

1. Open the Hello Location project. In the `Hello_locationViewController.h` file, create a `UITableView` variable that will be used to display a Table View in our app. We will also define a variable for connecting to the Eventful API URL via `NSURLConnection`. An XML Parser of the type `NSXMLParser` is created as well, and lastly, a variable to store the XML content retrieved from Eventful of the type `NSMutableData` is also created.
2. We also need to use the `NSXMLParserDelegate` in our class definition to use the methods implemented by the `NSXMLParser`. Create an outlet for the `UITableView` and name it `myDataTable`:

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

@interface Hello_LocationViewController : UIViewController
    CLLocationManagerDelegate, UITableViewDataSource,
    NSXMLParserDelegate
{
    CLLocationManager *locMgr;
    CLLocation *userLocation;
    UITableView *myDataTable;
    NSURLConnection *urlConnection;
    NSXMLParser *xmlParser;
    NSMutableData *xmlContent;
}

@property (nonatomic, retain) IBOutlet UITableView *myDataTable;

@end
```

3. In our `Hello_LocationViewController.m` file, synthesize the `myDatatable` variable. Create a `NSMutableArray` variable called `events` to store the events title received from Eventful API.
4. Create a `MutableString` variable, `titleText`, which will be used to parse the XML and send an element value to the `UITableView`. Create another string variable `currentXMLTitle` to store the current XML element name.
5. In the `viewDidLoad` method of our View Controller, we initialize the `xmlContent` and `events` variable. The following is what the `viewDidLoad` method now looks like:

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

    xmlContent = [[NSMutableData alloc] init ];

    locMgr      = [[CLLocationManager alloc] init];
    locMgr.desiredAccuracy= kCLLocationAccuracyKilometer;
    locMgr.distanceFilter=1000.0f;
    locMgr.delegate=self;

    userLocation = [[CLLocation alloc] initWithLatitude:37.33
        longitude:-122.03];

    if([CLLocationManager locationServicesEnabled]==FALSE)
    {
        [locMgr startUpdatingLocation];
    }

    if([CLLocationManager locationServicesEnabled]==TRUE)
    {
        [locMgr startUpdatingLocation];
    }
    events = [[NSMutableArray alloc] init];
}
```

- 6.** When the location gets updated, through the `didUpdateToLocation` method, we call the Eventful API via `NSURLConnection` and `NSURLRequest`. Note that you will need your own Eventful API key to get this example running:

```
- (void) locationManager:(CLLocationManager *)manager
  didUpdateToLocation:(CLLocation *)newLocation
  fromLocation:(CLLocation *)oldLocation
{
    xmlContent = [[NSMutableData alloc] init];
    events = [[NSMutableArray alloc] init];

    NSString *newLatitude = [[NSString
        alloc] initWithFormat:@"%g", newLocation.coordinate.latitude];
    NSString *newLongitude = [[NSString
        alloc] initWithFormat:@"%g", newLocation
        .coordinate.longitude];

    latitudeTextData = newLatitude;
    longitudeTextData = newLongitude;

    // Call EventFul API Now

    NSString *appKey = @"xxxxxxxxxxxxxxxxx";

    NSString *url = [NSString
        stringWithFormat:@"http://api.eventful.com/rest/events/search?
        keywords=concerts&location=%@,%@&app_key=%@&within=10",
        newLatitude, newLongitude, appKey];

    NSURL *urlToRequest = [[NSURL
        alloc] initWithString:url];
    NSURLRequest *request = [NSURLRequest
        requestWithURL:urlToRequest];

    urlConnection = [[NSURLConnection alloc]
        initWithRequest:request
        delegate:self startImmediately:YES];
}
```

- 7.** The `NSURLConnection` class calls the `didReceiveData` method when it receives a response successfully from the web service that was called for.

- 8.** In the `didReceiveData` method, we use the response data and pass it to an XML Parser to parse the XML received as follows:

```
- (void)connection: (NSURLConnection *) connection
    didReceiveData: (NSData *) data
{
    [xmlContent appendData:data];
    xmlParser = [[NSXMLParser alloc] initWithData:xmlContent];
    xmlParser.delegate = self;
    [xmlParser parse];
}
```

- 9.** The `NSXMLParser` class has three methods of traversing an XML, namely, `didStartElement`, `didEndElement`, and `foundCharacters`. The XML Parser calls the `didStartElement` method when it encounters a start of an XML element; similarly, it calls the `didEndElement` when it encounters close tags for an XML Element.

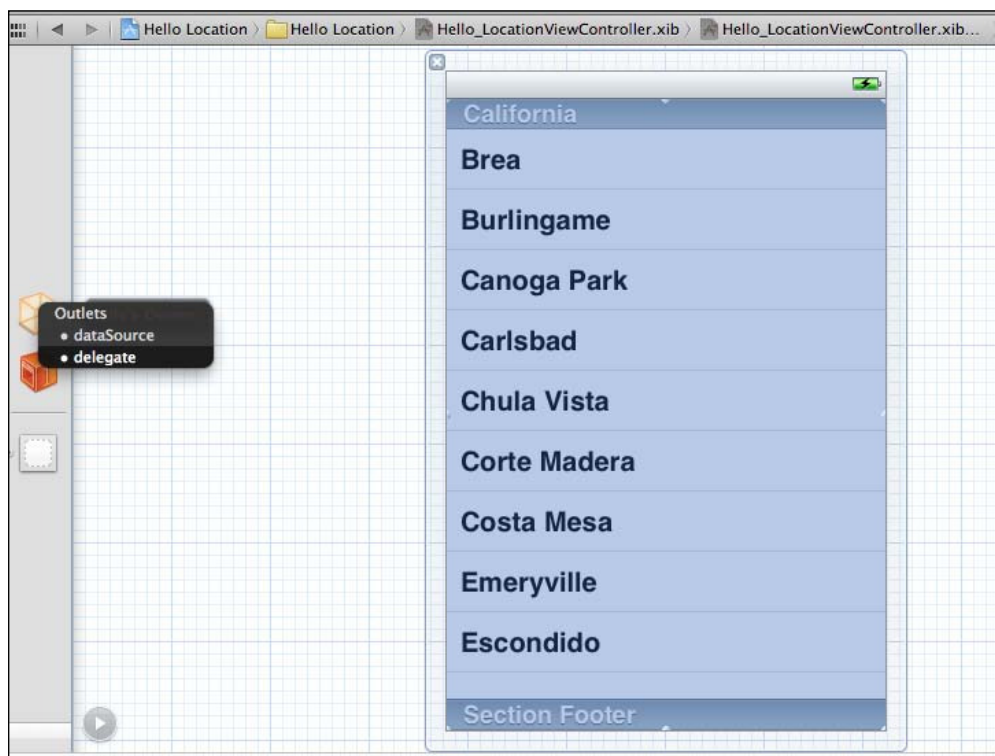
- 10.** Between the `StartElement` and `EndElement` call, the parser calls the `foundCharacters` if it finds textual content in the XML element. We use this method to fetch the events from Eventful API. The XML element for the event name is `event`, and the title stores the event name. So we use the title element of the response as follows and add it to the `UITableView`:

```
if([currentXMLTitle isEqualToString:@"title"])
{
    [titleText appendString:string];
}
```

- 11.** When the `EndElement` parser method is called, we add the title received to the `UITableView` via the `events` mutable array defined in the header, as shown in the following code snippet:

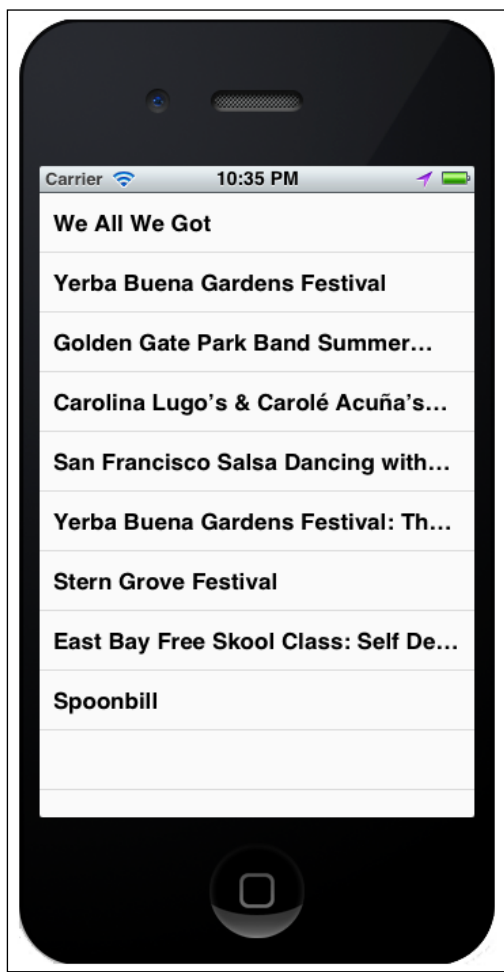
```
- (void)parser: (NSXMLParser *) parser didEndElement: (NSString *)
    elementName namespaceURI: (NSString *) namespaceURI
    qualifiedName: (NSString *) qName
{
    if([currentXMLTitle isEqualToString:@"title"])
    {
        if (![events containsObject:titleText])
        {
            [events addObject:titleText];
            titleText = [[NSMutableString alloc] init];
        }
    }
}
```

12. When the XML Parser completes the parsing via the parser `DidEndDocument` method, we reload the `UITableView` to reflect the addition of the event titles in the same.
13. It is a good time to get familiar with the `UITableView`. Open your NIB file and add the `UITableView` to it, as follows:



14. Don't forget to connect the `UITableView` with the file's owner – **dataSource** and **delegate**. The `UITableViewDataSource` contains the necessary methods to construct and modify a `UITableView`.

- 15.** Run the project in the simulator with Location Simulation; you should see an output as shown in the following screenshot:



The code for the Eventful API project can be found on the book's website, under *Chapter 3* – titled *Hello Location – Eventful*

What just happened?

We created a sample application that uses Location along with XML Parsing and Web Services API, as well as a UI TableView to display nearby events in our app. We used Eventful events API to fetch information, by passing the user's location in the API URL, as soon as there is an update in the user's location the API URL is called and the new events obtained.

We then parse the XML and displayed it in a simple UI Tableview. In *Chapter 6*, we will extend this example to create a complete `Events` app and even submit to the app Store!!

Time for action – Last.fm API in your app

Let's do a quick events call from the `Last.fm` API as well. From the Eventful API example, we are using the code to hit `Last.fm` as well. The only change we need to do is change the API URL to the following:

```
http://ws.audioscrobbler.com/2.0/?method=geo.getevents&lat=19.076&lon=72.8562&lon=&api_key=xxxxxxxxxxxxxxxx (where xxxxxxxxxxxxxxxx is your API key)
```

1. We added some UI Enhancements to the `tableView` by creating a header using the `titleForHeaderInSection` method of the `tableView Delegate`.

```
-(NSString *) tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    return @"Events by Last.fm";
}
```

2. A Calendar event icon is also added using the `UIImage` class as follows:

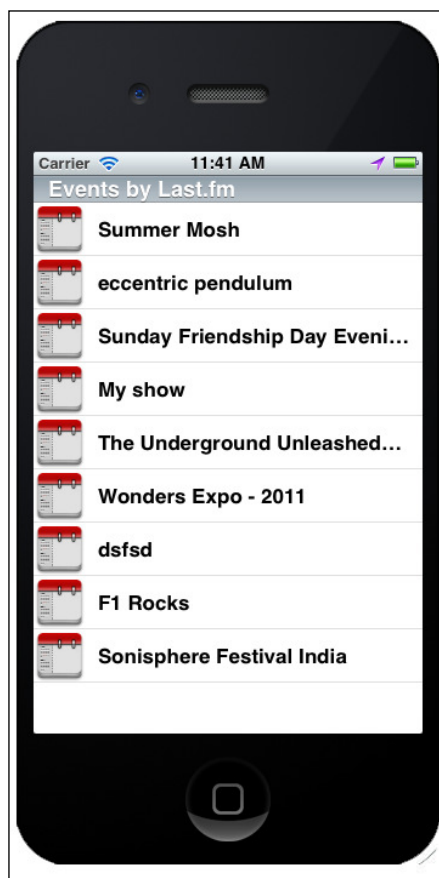
```
UIImage *cellImage = [UIImage imageNamed:@"Calendar.png"];
cell.imageView.image = cellImage;
```

3. The `UITableView didSelectRowAtIndexPath` method has been implemented to add interactivity to the application. When the user selects a table row, we call an alert box:

```
-(void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSString *eventClicked = [events objectAtIndex:indexPath
row];
    UIAlertView *alert = [[UIAlertView
alloc] initWithTitle:@"You selected the following event "
message:eventClicked
delegate:self
cancelButtonTitle:@"OK"
otherButtonTitles:nil];
    [alert show];
}
```

It is a good time to get a hang of `UITableView`, `NSURLConnection`, and the `XMLParser` classes to utilize the web services call and bind them to a `UITableView`. A lot of location-based applications are built this way, although with more features, which we will review in our sample apps in the forthcoming chapters.

4. The output now looks like the following screenshot:



What just happened?

We used the popular music website `Last.fm` as the source for our Events content this time and used the same logic as in the `eventful` example. However, this time we created a `UITableView Cell Clicked` event that shows an alert when a particular row in the `UITableView` is clicked on. This can be useful to show more information from the API. Typically, a `mapView` signifies the events location or more description of the event, including tickets, sample songs, and so on.

Extending Hello Location for local search

Having understood quite a few details about iOS Location APIs, let's move a bit further from the Hello Location paradigm to a real-world application usage scenario – **local search**.

Local search—implying location-based search for businesses including bars, cafés, restaurants, shopping malls, gas stations, pizza outlets—is the most common usage of location data, from simple content display apps to complex check-in, augmented reality-based apps. If you have an iPhone, then you would have surely used the foursquare application or the Google Places application.

Now let's dive into the foursquare developer site <https://developer.foursquare.com/docs/> and register your app at <https://foursquare.com/oauth/> to start using the foursquare API calls, as shown in the following screenshot:

The screenshot shows the Foursquare OAuth Consumer Registration page. The browser address bar displays <https://foursquare.com/oauth/register>. The page header includes the Foursquare logo, the user's location 'Zeeshan in Mumbai, Maharashtra', and links for 'Notifications', 'Apps', 'Help', 'Settings', and 'Log Out'. A navigation bar contains links for 'Me', 'History', 'Badges', 'Stats', and 'Friends', along with a search bar labeled 'Search places, people and tips'. The main content area is titled 'OAuth Consumer Registration' and features a form for 'APPLICATION DETAILS'. The form includes three input fields: 'APPLICATION NAME', 'APPLICATION WEB SITE', and 'CALLBACK URL'. Below these fields is a note: 'Your application must abide by our [acceptable use policy](#) and [trademark guidelines](#).' A green 'REGISTER APPLICATION' button is at the bottom of the form. To the right of the form is a box titled 'Why do we use OAuth?' with text explaining its importance for tracking API usage and data safety. The footer contains links for 'About', 'Blog', 'Contact', 'Help', 'Jobs', 'Privacy 101', 'Privacy Policy', 'Terms (New)', and 'Store', along with copyright information '© 2011 foursquare - English'.

Note down your Client ID and Client Secret. This will be required for the API calls in our application.

Important things to know before we begin

1. Client ID and Client Secret from Foursquare
2. `NSJSONSerialization` – New JSON API in iOS 5
3. `NSDictionary` and/or `NSArray` implementation details

Time for action – building a local search app with foursquare

We use the example code from Eventful and the Last.fm example. However, this time we do not use XML; instead, we use JSON along with the new `NSJSONSerialization` class in iOS 5. The `NSJSONSerialization` class is useful to convert JSON-to-Core Foundation objects and Core Foundation objects to JSON. In simpler words, you can convert JSON retrieved from web services to `NSArray`, `NSDictionary` types easily and use in your application.

1. We begin by creating a `UITableView` in our `Hello_LocationViewController.xib` file and exposing an outlet, as in the Eventful example before.
2. Next in the `didReceiveData` method of `NSURLConnection`, we define an object of the type `NSDictionary` as `NSDictionary *dictionary;`, which will hold our JSON data from the foursquare API. This is after converting the received data into JSON format. We also add the `NSJSONReadingAllowFragments` option to allow objects that are not of the types `NSArray` or `NSDictionary` to be converted into an appropriate JSON format:

```
- (void)connection: (NSURLConnection *)connection
didReceiveData: (NSData *)data
{
    NSError *jsonError;
    NSDictionary *dictionary;
    NSArray *items;

    dictionary= [NSJSONSerialization JSONObjectWithData:data
                options:NSJSONReadingAllowFragments error:&jsonError];

    items      = [NSArray arrayWithObject:[dictionary
                objectForKey:@"response"]objectForKey:@"groups"]];

    NSUInteger count    = [[[[items objectAtIndex:0]
                objectForKey:@"items"]count];

    for(NSInteger i=0;i<count-1;i++)
    {
```

```

NSString *titleText = [[[[[items objectAtIndex:0]
    objectAtIndex:0] objectForKey:@"items"] objectAtIndex:i]
    objectForKey:@"name"];
if (![venues containsObject:titleText])
{
    [venues addObject:titleText];
}
}

[myDataTable reloadData];
}

```

3. We created a dictionary object and converted the JSON data received into an array for easier parsing and adding the name of the *Nearest Venue* to the UITableView via the venues array.
4. The `JSONObjectWithData` method converts the data from the Foursquare API (which is in JSON format) to a Foundation object. In this case it is an instance of `NSDictionary`, which we eventually convert into an array.
5. We then loop through the array of the `items` object (that holds the venue information details) and use the name of the venue to pass on to the venue array for the UITableView.
6. Run the application in the iOS Simulator. You should see an output like the one in the following screenshot:



What just happened?

We used the foursquare venues API to fetch nearby venues by passing the geo location coordinates from our iPhone to the foursquare API. The result of the search is a JSON payload, which we convert into Core Foundation objects using the `NSJSONSerialization` class in iOS 5.

Before iOS 5, JSON parsing was available via third-party add-ons, some of which are JSONKit, JSON-framework. However, with the addition of the `NSJSONSerialization` class, third-party JSON frameworks are not required. The twitter framework in iOS 5 also uses `NSJSONSerialization`. Find the code for this example on the book's website, in a project titled *Hello Location - Foursquare*.

Pop quiz – Location, Location, and Location

1. What is the name of the class that holds the Location information in the Core Location Framework?
 - a. `CLLocationManager`
 - b. `CLLocation`
2. Name the class responsible for forward and reverse Geocoding in iOS 5
 - a. `CLLocationManager`
 - b. `CLGeocoder`
3. What is the method used to check whether location services are enabled on your iOS device or not?
4. How do you enable background location in your app.
 - a. Keep the app running in the background by pressing the home button
 - b. Enable Location from the iPhone's settings pages
 - c. Use `UIBackgroundModes` with the key 'location'
5. Name the JSON API in iOS 5 SDK
 - a. `NSJSONSerialization`
 - b. `JSONKit`

Summary

In this chapter, we learned how iOS SDK handles a location with the Core Location framework, along with sample apps for showing nearby venues and events.

Specifically, we covered:

- ◆ Starting the Location Manager to receive location data
- ◆ Getting Location data updates in your application
- ◆ New iOS 5 API calls for Geocoding and Reverse Geocoding
- ◆ Location Simulation in the iOS 5 using Xcode 4.2
- ◆ Region monitoring with Core Location
- ◆ User Authorization for Location data security

We also discussed XMLParser and the new JSON API – NSJSONSerialization. In total, we used both the XMLParser and the JSON API to manipulate data from third-party APIs.

Now that we know how to handle Location in iOS 5, we now move further into the Maps territory by using the MapKit API in the next chapter.

4

Using Maps in your iOS apps—MapKit

*Maps provide a great visual experience for location-based services. Apple iOS SDK includes a dedicated API for maps, via the **MapKit** framework.*

In this chapter, we will have a look at the following topics:

- ◆ Overview of the MapKit framework
- ◆ Understanding map geometry
- ◆ Working with map gestures – panning, zooming, and pinch zoom
- ◆ Annotating maps
- ◆ Draggable and custom map annotations in your apps
- ◆ Map overlays
- ◆ Working with the OpenStreetMaps-based CloudMade SDK for iOS
- ◆ User tracking Modes – iOS 5's new features

So let's get on with it...

Overview of the MapKit framework

The MapKit framework provides iOS developers with the ability to display, annotate, and overlay information on maps using Google maps data. Maps are now a default feature on most location-based applications, as it provides a good graphical overview of the user's location and his or her distance/nearness from the information he/she seeks in a location context.

With custom pin markers and directions, it also helps users navigate to the destination easily. Most importantly, maps give the user a feeling that *this place is around the next block, north from where I am standing*, so decision-making happens quickly. Whether the user needs to catch a taxi or a bus or if it is easier to walk, all of it happens quickly in the user's head. As the map provides an intuitive information overlay that helps the user take this decision in a fraction of the time compared to analyzing texts of information that guide users to do step 1, step 2, step 3, and so on.

This can be correlated with studies done on how the human mind works. It seems **Google Maps** functions similar to the way our brains process map information. For the science geek, here is the link – <http://www.sciencedaily.com/releases/2009/10/091007081528.htm>; no wonder people love Google Maps!

The iOS MapKit framework provides us with the following capabilities:

- ◆ Add a map view to your app (using MKMapView)
- ◆ Add annotations (read markers) with draggable, custom annotations support
- ◆ Show a user's location on a map
- ◆ Overlays
- ◆ Tracking modes (new in iOS 5)



Tracking modes is a new feature that specifies how the user's location updates affect the map's positioning. So if tracking modes are turned on, your map display will always be updated with the user's current location. Another option also rotates the map display based on the heading values.

Understanding map geometry

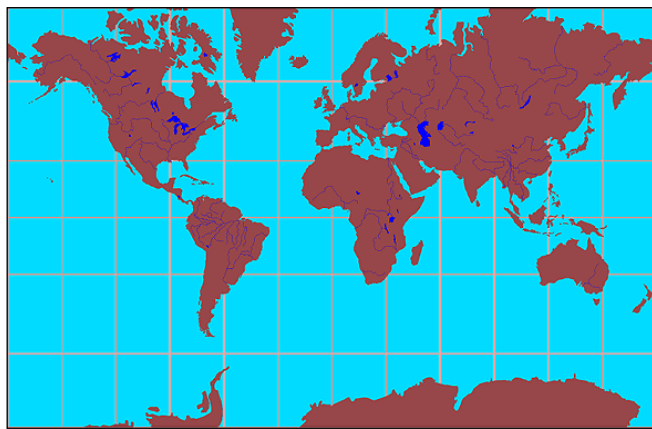
Before we delve into the methods, actions, and outlets of MapKit, it is a good idea to get acquainted with the background map geometry and how it works for Google maps. If you have a background in Computer Science, then you would be aware of keywords such as projection, trajectory, co-ordinate systems, raster, and scalable graphics. In fact, most of the Computer Graphics stuff you learned in school would relate here. If you are not from a Computer Science background, then a basic understanding of the Google Maps co-ordinate system will be good enough to begin working with MapKit. You can find the same at http://code.google.com/apis/maps/documentation/javascript/v2/overlays.html#Google_Maps_Coordinates

Google maps, and hence MapKit, use the **Mercator projection** model of converting the Earth's Sphere into a corresponding flat surface grid-based, parallel map. In such a projection, the longitude lines are parallel, and hence landmass further from the equator tends to be distorted. However, Mercator projection works well for navigational purposes, and therefore, despite the drawbacks, it is still used today.

The following images should give you a good idea about the Mercator projection:



Earth's surface as a sphere—Image courtesy - Michael Pidwirny from <http://www.eoearth.org/article/Maps> and <http://www.physicalgeography.net/fundamentals/2a.html>





Mercator projection of the Earth's surface—Image courtesy - Michael Pidwirny from <http://www.eoearth.org/article/Maps> and <http://www.physicalgeography.net/fundamentals/2a.html>

MapKit supports three co-ordinate systems to point to a location on the map:

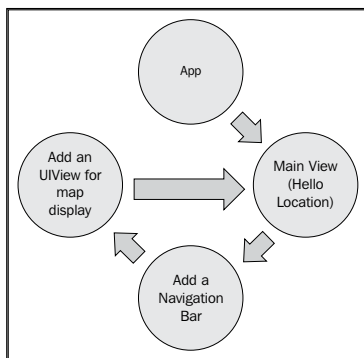
1. Map co-ordinate system: Regular latitude/longitude values
2. A map point: x and y values on the Mercator map projection
3. A point: A unit associated with the co-ordinate system of a UIView Object

The Map co-ordinate system is the best, accurate, and portable way for storing location data. We can convert from any of these co-ordinate systems, back and forth, using the MapKit conversion functions. Let's have a quick glance at them.

Conversion From	Conversion To	Conversion Functions
Map Co-Ordinates	Points	<code>convertCoordinate:toPointToView</code> <code>convertRegion:toRectToView</code> MKMapView methods
Map Co-Ordinates	Map Points	<code>MKMapPointForCoordinate</code>
Map Points	Map Co-Ordinates	<code>MKCoordinateForMapPoint</code> <code>MKCoordinateRegionForMapRect</code>
Map Points	Points	<code>pointForMapPoint</code> <code>rectForMapRect</code> MKOverlayView methods
Points	Map Co-Ordinates	<code>convertPoint:toCoordinateFromView</code> <code>convertRect:toRegionFromView</code> MKMapView methods
Points	Map Points	<code>mapPointForPoint</code> <code>mapRectForRect</code> MKOverlayView methods

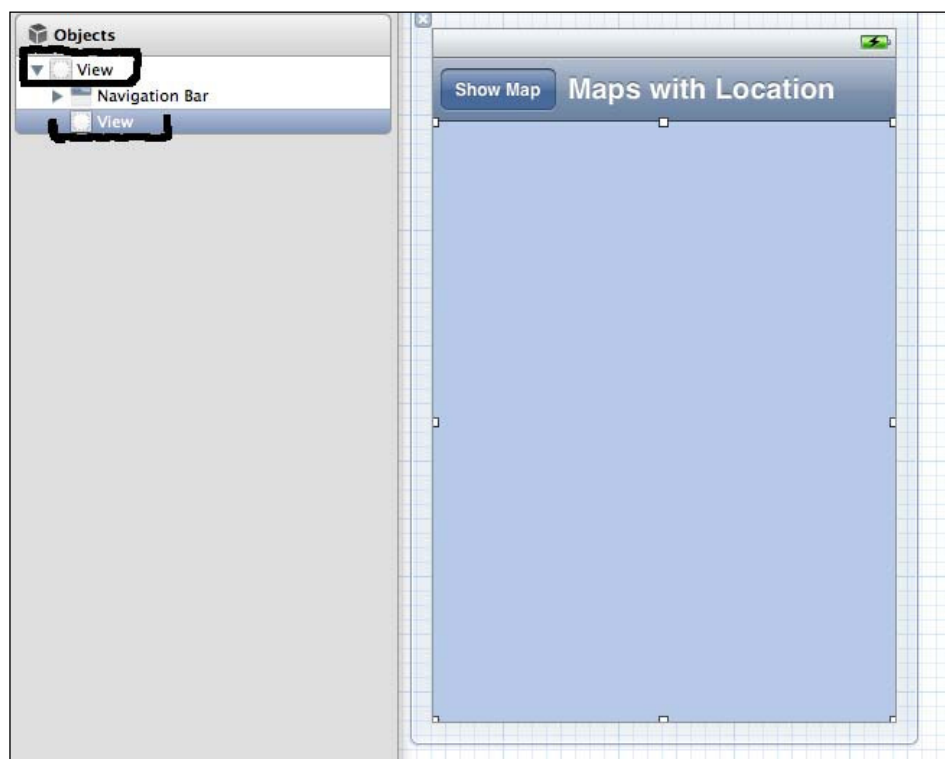
Time for action – using MapKit in your app

We will use the `Hello Location` application we saw in *Chapter 3*, the one that uses location updates in our app, so that we can change the map display as the location changes. The following diagram should give you an idea of our app behavior:

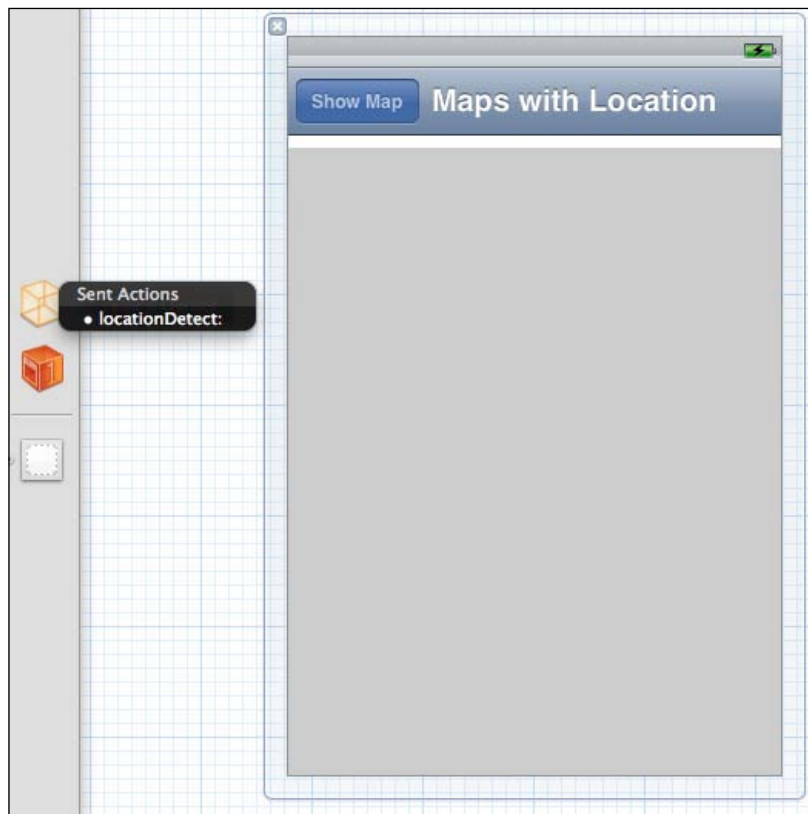


We modify the `Hello_LocationViewController.xib` file to make our app look more professional as follows:

1. Add a **UINavigationController** to your application's **NIB** file. Create another view by dragging it from the **Toolbox** onto the **NIB** file. Your app should now look like the following screenshot; do not get confused with the **UIView** already present. What we will be learning is superimposing another **UIView** (for this example, it will contain the map) onto the **Main UIView** of the application.



2. Now we map the Navigation Bar's **Show Map** button to the `locationDetect` action.



3. In your `Hello_LocationViewController.h` file, import the **MapKit Framework** and define the `MKMapView` object, as well as the Map View you defined in step 1. Your code should now look like following snippet:

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

@interface Hello_LocationViewController : UIViewController
<CLLocationManagerDelegate>
{
    CLLocationManager *locMgr;
    CLLocation *userLocation;
    NSString *message;
    MKMapView *map;
    UIView *mapView;
}
```

```

    }

    @property (retain, nonatomic) MKMapView *map;
    @property (strong, nonatomic) IBOutlet UIView *mapView;

    - (IBAction)locationDetect:(id)sender;

@end

```

4. In your `Hello_LocationViewController.m` file, synthesize the map and mapview objects:

```

@synthesize map;
@synthesize mapView;

```

5. Furthermore, create an `MKCoordinateRegion` object that will hold the portion of the map to display as follows:

```

MKCoordinateRegion region;

```

6. In the `viewDidLoad` method, we create the map object and bind it to the new View we created to hold the map; the map type is the default Google Maps standard view, which is as follows:

```

map = [[MKMapView alloc] initWithFrame:self.mapView.bounds];
map.mapType = MKMapTypeStandard;

```

7. We define the region to show on the map from the user's initial location (San Francisco, if the location is not found; the actual location otherwise). The zoom level is controlled by the span variable of the region. Next, we enable the map zooming. The complete `viewDidLoad` method is as shown in the following code snippet:

```

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

    locMgr = [[CLLocationManager alloc] init];
    locMgr.desiredAccuracy= kCLLocationAccuracyBest;
    locMgr.distanceFilter=1000.0f;
    locMgr.delegate=self;
    userLocation = [[CLLocation alloc] initWithLatitude:37.33
        longitude:-122.03];
    message = [[NSString alloc] initWithString:@""];

    if([CLLocationManager locationServicesEnabled]==NO)

```

```
{
    message=@"Location cannot be initialized. Please check
        settings";
}
[locMgr startUpdatingLocation];
map = [[MKMapView alloc]
initWithFrame:self.mapView.bounds];
map.mapType = MKMapTypeStandard;
region.center = userLocation.coordinate;
region.span.latitudeDelta = 0.1;
region.span.longitudeDelta = 0.1;
map.zoomEnabled = TRUE;
[map setRegion:region animated:TRUE];
}
```

- 8.** When the location changes, we use the new location values and pass it to the region variable and update the map in the Location Manager's `didUpdateToLocation` method as follows:

```
(void) locationManager:(CLLocationManager *)
manager didUpdateToLocation:(CLLocation *)newLocation
fromLocation:(CLLocation *)oldLocation
{
    region.center = newLocation.coordinate;
    [map setRegion:region animated:TRUE];
}
```

- 9.** On the **Show Map** button click, we defined the `locationDetect` method to be called. Based on the user's authorization, we define the region of the map accordingly, and then, finally updated the map. The map is not rendered on the View unless we add it as a subView to the `mapView` we created in step 1, 2, and 3, as follows:

```
[self.mapView addSubview:map];
```

- 10.** The following is the updated `locationDetect` method:

```
- (IBAction)locationDetect:(id)sender
{
    if ([CLLocationManager locationServicesEnabled]==TRUE)
    {
        if ([CLLocationManager authorizationStatus]==
            kCLAuthorizationStatusNotDetermined)
        {
            message = @"User hasn't made a choice yet.
                Defaulting to San Francisco";
        }
    }
}
```

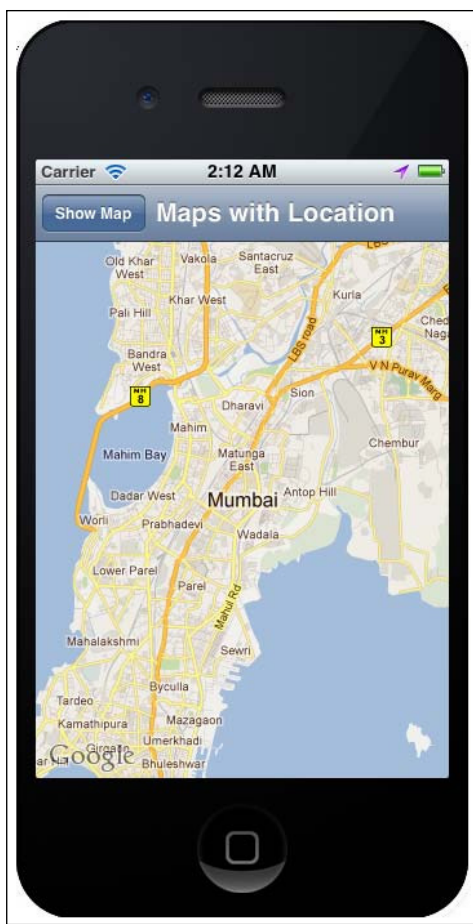
```
        region.center    =    userLocation.coordinate;
    }

    else if([CLLocationManager authorizationStatus]==
        kCLErrorAuthorizationStatusDenied)
    {
        message = @"User has denied use of location
                    services for your application or
                    all applications.Defaulting to San Francisco";
        region.center    =    userLocation.coordinate;
    }

    else if([CLLocationManager authorizationStatus]==
        kCLErrorAuthorizationStatusAuthorized)
    {
        message = @"User has authorized your
                    application for location services.";
        region.center    =    locMgr.location.coordinate;
    }
    else if([CLLocationManager authorizationStatus]==
        kCLErrorAuthorizationStatusRestricted)
    {
        message = @"Not authorized to use location
                    services.Defaulting to San Francisco";
        region.center    =    userLocation.coordinate;
    }

    [map setRegion:region animated:TRUE];
    [self.mapView addSubview:map];
}
else if([CLLocationManager locationServicesEnabled]==FALSE)
{
    message =    @"Location cannot be initialized. Please check
                    settings";
}
}
```

- 11.** Run the application on the iOS simulator by using Location Simulation. You should see the following result when you click on the **Show Map** button on the navigation bar:



What just happened?

We created a simple app that detected a user's location updates and displayed a map. We combined Core Location and MapKit functionality to do so. Try changing the location values via the Location Simulation feature in iOS 5 in the **Product | Debug | Simulate Location** menu option and see the map change to the simulated location. Note the map should be rendered once to observe this change.

Do not forget to include the MapKit Library reference in your project. You can find the code for this example on the book's website, project titled *Hello Location - With Maps*.

Time for action – using map gestures – panning and zooming

Panning and **Zooming** are two of the basic gestures on a Map. Panning on a MapView can be achieved in two ways. They are as follows:

- ◆ Use the `centerCoordinate` property of `MKMapView`
- ◆ Use the `setCenterCoordinate` method

Similarly, zooming can be controlled by performing the following:

- ◆ Modifying the `span` value in the `region` property of the `MKMapView`
- ◆ Using the `setRegion` method

To make the map zoomable, we need to use the `zoomEnabled` property of `MKMapView`; remember from our previous example where we used it as follows:

```
map.zoomEnabled = TRUE;
```

Zooming can be controlled by the `region` property of `MKMapView` and by the `setRegion:animated:` method. We also used the `region.center` property before. Now we will use the `map.centerCoordinate` to pan the map. **The difference being that the `region.center` property changes the zoom level of the map as well. However, the `map.centerCoordinate` does not change the zoom level of the map, so it enables true Panning.**

1. From the last example, open the `Hello_LocationViewController.m` file and update the `didUpdateLocation` method of the location manager to use the `map.centerCoordinate` method of panning the map to the new location as follows:

```
- (void) locationManager:(CLLocationManager *)manager
  didUpdateToLocation:(CLLocation *)newLocation
  fromLocation:(CLLocation *)oldLocation
{
    CLLocationCoordinate2D newCenter;
    newCenter.latitude = newLocation.coordinate.latitude;
    newCenter.longitude = newLocation.coordinate.longitude;
    map.centerCoordinate= newCenter;
}
```

2. Change the `locationDetect` method to:

```
- (IBAction) locationDetect:(id) sender
{
    CLLocationCoordinate2D newCenter;

    if ([CLLocationManager locationManagerServicesEnabled]==TRUE)
    {
```

```
if([CLLocationManager authorizationStatus]==
    kCLAuthorizationStatusNotDetermined)
{
    message = @"User hasn't made a choice yet.Defaulting
               to San Francisco";
    newCenter.latitude =   userLocation.coordinate.latitude;
    newCenter.longitude =  userLocation.coordinate.longitude;
    map.centerCoordinate = newCenter;
}

else if([CLLocationManager authorizationStatus]==
    kCLAuthorizationStatusDenied)
{
    message =  @"User has denied use of location services
               for your application or all applications.
               Defaulting to San Francisco";
    newCenter.latitude =   userLocation.coordinate.latitude;
    newCenter.longitude =  userLocation.coordinate.longitude;
    map.centerCoordinate = newCenter;
}

else if([CLLocationManager authorizationStatus]==
    kCLAuthorizationStatusAuthorized)
{
    message = @"User has authorized your application
               for location services.";
    newCenter.latitude =   locMgr.location.coordinate.latitude;
    newCenter.longitude =  locMgr.location.coordinate.longitude;
    map.centerCoordinate =   newCenter;
}

else if([CLLocationManager authorizationStatus]==
    kCLAuthorizationStatusRestricted)
{
    message = @"Not authorized to user location
               services.Defaulting to San Francisco";
    newCenter.latitude =   userLocation.coordinate.latitude;
    newCenter.longitude =  userLocation.coordinate.longitude;
    map.centerCoordinate =   newCenter;
}

[self.mapView addSubview:map];
}
```

```

else if([CLLocationManager locationServicesEnabled]==FALSE)
{
    message = @"Location cannot be initialized. Please check
    settings";
}
}

```

3. Run the application in the simulator and zoom to a particular location. Now change the location via the Location Simulator to a different city or any custom location. You will observe that the zoom level remains the same, only the map has been panned to the new location.

The code for this example can be found on the book's website, in a project titled *Hello Location - Maps with Pan and Zoom*.

Annotating Maps – an overview

Annotations are to iOS SDK what markers are to Google Maps API. Annotations are used to display a single geocoded entity on a map; this entity could be a local restaurant, or a bus stop, or a cinema hall. Annotations such as the `MapView` are `UIView` elements that can be rendered on the iOS device.

The `MKAnnotationView` class, along with the `MKAnnotation` protocol, is responsible for managing annotations on a map. MapKit allows the following functionality for Annotations:

- ◆ Adding and Displaying Annotations on the map
- ◆ Draggable Annotations
- ◆ Custom Map Annotations

Time for action – adding annotations to your maps

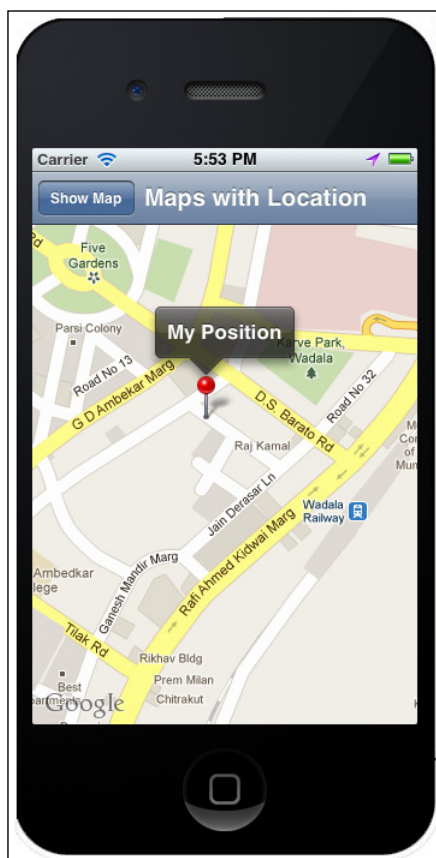
Let's begin extending the map application we created before by showing an Annotation object on the map, signifying the user's current location on the map.

1. We begin by creating an `MKPointAnnotation` object, define it in the `Hello_LocationViewController.h` file as `MKPointAnnotation *annotation;`, and expose it as a property `@property (retain, nonatomic) MKPointAnnotation *annotation;`

2. In the `Hello_LocationViewController.m` file, synthesize the annotation object, and in the `viewDidLoad` method, initialize and add the annotation to the map by using the following code:

```
annotation          = [[MKPointAnnotation alloc] init];
annotation.title     = @"My Position";
annotation.coordinate = userLocation.coordinate;
[map addAnnotation:annotation];
```
3. This is a simple annotation object added to the map. We change the annotation's position in the Location Manager's `didUpdateToLocation` method as

```
annotation.coordinate = newLocation.coordinate;
```
4. Similarly, we change the annotation's co-ordinates in the `locationDetect` method as well.
5. Run the application; you should see an annotation on the map, positioned at your location.



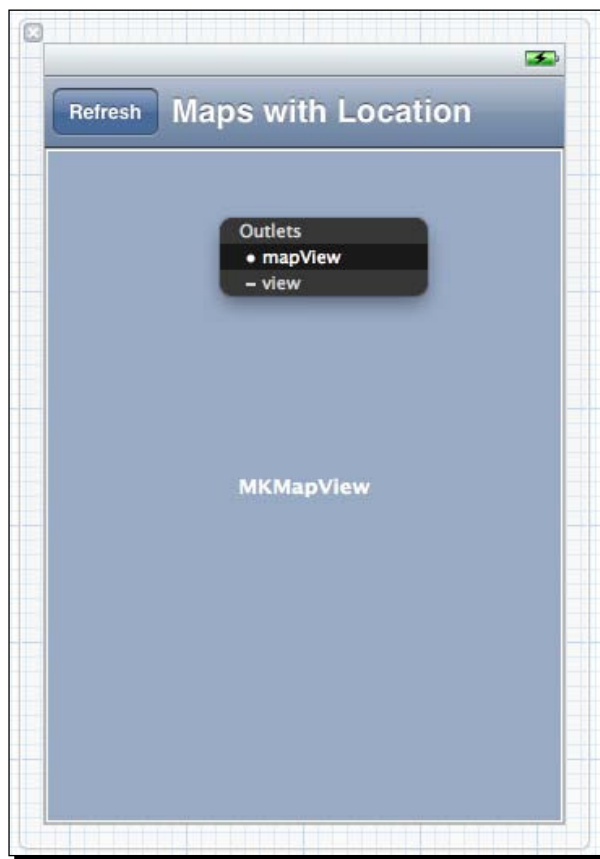
The code for this example can be found on the book's website, in the project titled *Hello Location - Annotations*

We will now look at how to create draggable and custom annotations. This time, we will use better annotation management techniques using the `MKAnnotationView` class.

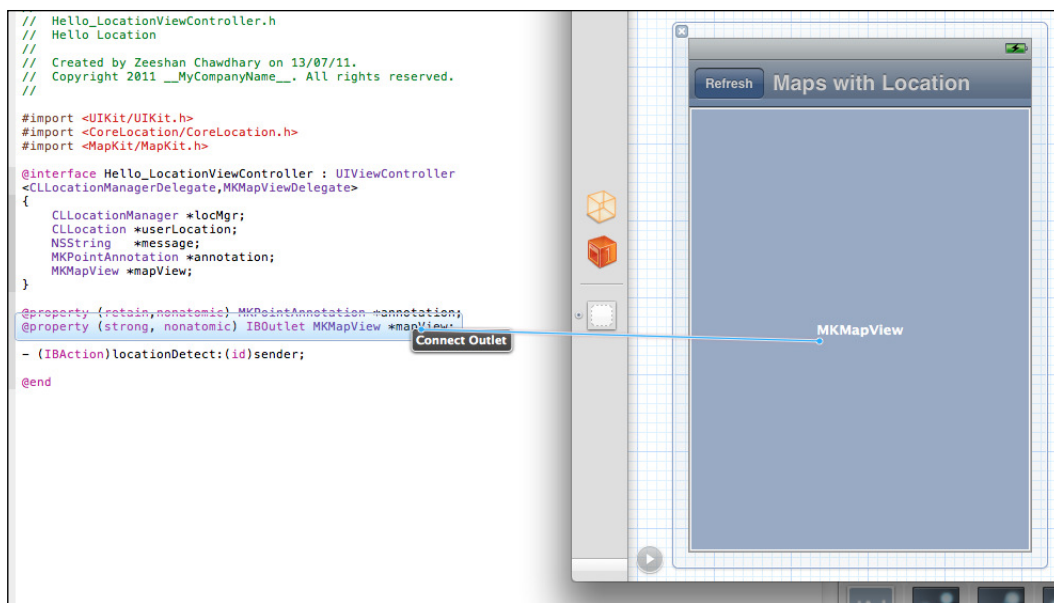
Time for action – draggable annotations

We now look at adding draggable annotations to our Maps.

1. We reuse the example from before, but rearrange the UI to look like the following image. This time, we add the `mapView` directly using the UI ToolBox.



2. We also create an IBOutlet as @property (strong, nonatomic) IBOutlet MKMapView *mapView; and connect the mapView from the Nib file to the outlet by keeping *Ctrl* clicked and dragging the MapView to the outlet.



3. As before, we define an MKPointAnnotation object annotation in our Hello_LocationViewController.h file. The following is the complete code:

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

@interface Hello_LocationViewController : UIViewController
    <CLLocationManagerDelegate, MKMapViewDelegate>
{
    CLLocationManager *locMgr;
    CLLocation *userLocation;
    NSString *message;
    MKPointAnnotation *annotation;
    MKMapView *mapView;
}

@property (retain, nonatomic) MKPointAnnotation *annotation;
@property (strong, nonatomic) IBOutlet MKMapView *mapView;

- (IBAction)locationDetect:(id)sender;

@end
```

4. In the `Hello_LocationViewController.m` file, we no longer need to create a `mapView` and call it implicitly, when we added the `mapView` to the `Nib` file via the UI Toolbox. Xcode automatically adds the code to show the map accordingly.
5. We implement the `mapView:viewForAnnotation` delegate function in our `.m` file. In this function, we use the `dequeueReusableCellWithIdentifier` method of an `MKPinAnnotationView` to check if the Annotation View can be reused. If not, then we create the new `MKPinAnnotationView`. We then make the `MKPinAnnotationView` **draggable** and change the pin color to green.

```
(MKAnnotationView *)mapView:(MKMapView *)mapView
viewForAnnotation:(id <MKAnnotation>)annotation
{

MKPinAnnotationView *annotationView=(MKPinAnnotationView *)
[self.mapView dequeueReusableCellWithIdentifier:
@"My Location"];

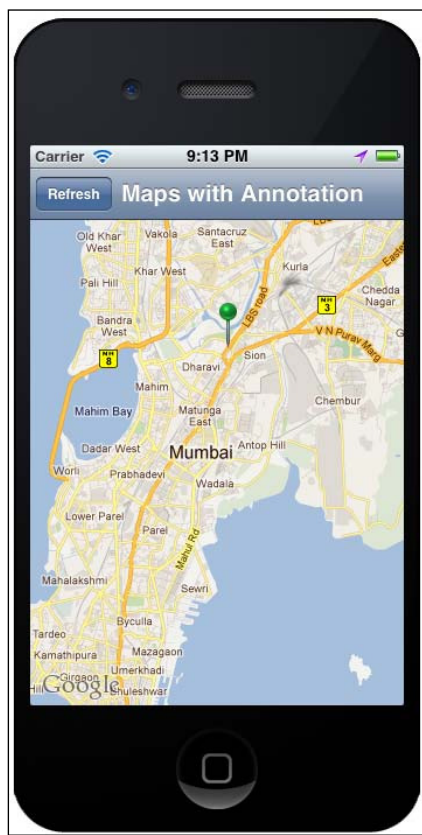
if (annotationView == nil)
{
    annotationView = [[MKPinAnnotationView alloc]
initWithAnnotation:self.annotation
reuseIdentifier:@"My Location"];
}
else
{
    annotationView.annotation = self.annotation;
}

    annotationView.draggable=TRUE;
    annotationView.canShowCallout=YES;
    annotationView.pinColor = MKPinAnnotationColorGreen;

return annotationView;
}
```

6. The `MapView` calls the `viewForAnnotation` method when an annotation is to be displayed. The `MKMapViewDelegate` delegate of `MKMapView` does this; make sure your class implements this delegate to use Annotation Views.

7. Run the application. You should see an output like one shown in the following screenshot:



What just happened?

We created a simple UI with a MapView on the `viewDidLoad` function. We added an annotation to the map. As the annotation was added to the mapView, the mapView called the `viewForAnnotation` delegate method, wherein we checked to see if we could reuse an existing `AnnotationView`. This is a good practice for better memory management. This is helpful in scenarios when we have a large number of annotations added to the mapView. We then create or reuse the current `AnnotationView` with the annotation object and make the `AnnotationView` draggable and change the default red annotation pin's color to green. Find the code for this example on the book's website, in a project titled *Hello Location - Draggable Annotations*.

Time for action – custom map annotations

Now that we have good hands-on experience with Annotations and Annotation Views, let's explore the full power of Annotation Views by creating our own custom Marker. We will be creating flag-based markers for cities, so if you create an annotation with Mumbai's latitude and longitude values, the annotation will show the Indian Flag, and when we create an Annotation with a San Francisco co-ordinate, it will show the American flag.

1. We extend from the preceding example and declare one more Annotation object in our class. In the `Hello_LocationViewController.h` file, add a `MKPointAnnotation` object as `MKPointAnnotation *mumbaiAnnotation;` and expose it as a property `@property (retain, nonatomic) MKPointAnnotation *mumbaiAnnotation;`
2. Furthermore, add two flag PNG images to your project, one for the US flag and another for the Indian Flag. Name them `usa-flag.png` and `india-flag.png`, respectively.
3. In the `viewDidLoad` method of our `Hello_LocationViewController.m` file, we instantiate a Mumbai Location object that holds Mumbai's Geo co-ordinates, as follows:

```
CLLocation *mumbaiLocation = [[CLLocation alloc]
    initWithLatitude:19.02 longitude:72.85];
```

4. Next, we set the title and co-ordinates for the `mumbaiAnnotation` object and add it to the `mapView`, as follows:

```
mumbaiAnnotation.title = @"Mumbai";
mumbaiAnnotation.coordinate=mumbaiLocation.coordinate;

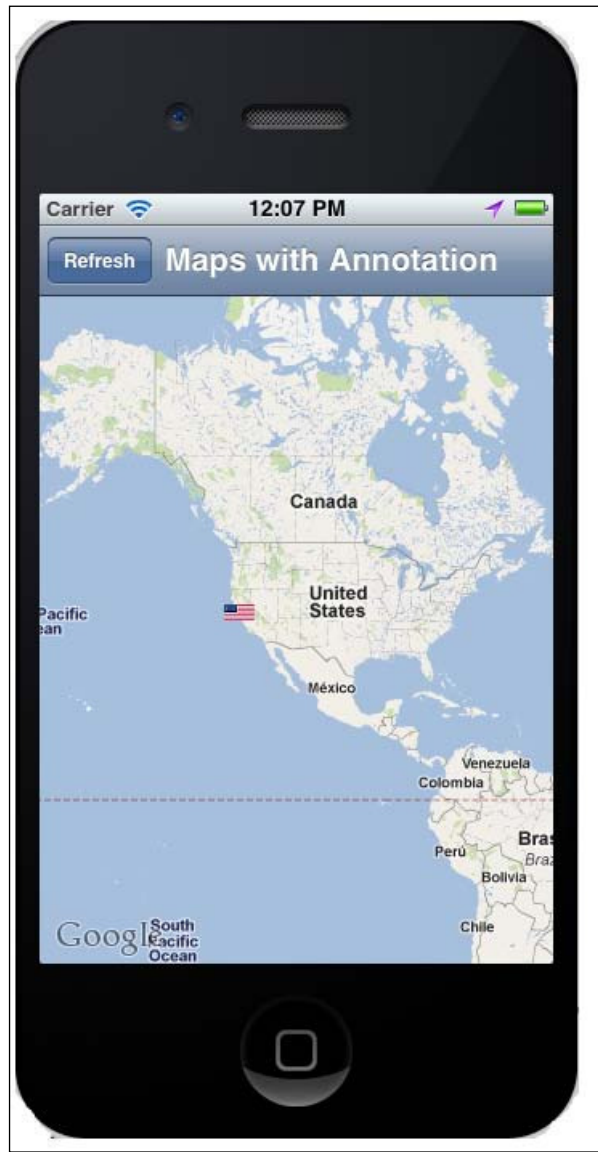
[mapView addAnnotation:mumbaiAnnotation];
```

5. In the `viewForAnnotation` delegate method implementation, this time we use the `MKAnnotationView` instead of the `MKPinAnnotationView`. We then check the annotation's title and assign the flag images to them accordingly.

```
if([annotationMapView.title isEqualToString:@"Mumbai"])
{
    annotationView.image = [UIImage imageNamed:@"india-
        flag.png"];
    annotationView.draggable=TRUE;
    annotationView.canShowCallout=YES;
}
else if ([annotationMapView.title isEqualToString:@"Detected
    Location"])
{
```

```
annotationView.image = [UIImage imageNamed:@"usa-flag.png"];  
annotationView.draggable=TRUE;  
annotationView.canShowCallout=YES;  
}
```

6. Run the application by having Location Simulation set to "San Francisco". You should see the following results:



7. Scroll the application to your right. You should also see the Mumbai Annotation with the Indian Flag, as shown in the following image:



What just happened?

We created two custom annotations, displaying the country flags for two cities, based on their annotation title (Mumbai or Detected Location). Note, we purposely simulated this application with the San Francisco co-ordinates, but this app can run without any location simulation. All we need to do is change the `if` condition where the annotation title is compared.

The code for this example can be found at the book's website, in a project titled *Hello Location - Custom Annotations*.

Have a go hero – use `CLRegion` to detect a user's city

Remember our `CLRegion` example from *Chapter 3*, where we introduced a region of 1,000 meters around San Francisco? Use the same to detect if the user entered the San Francisco region and show a map annotation with the US flag; do the same for Mumbai.

Map overlays – an overview

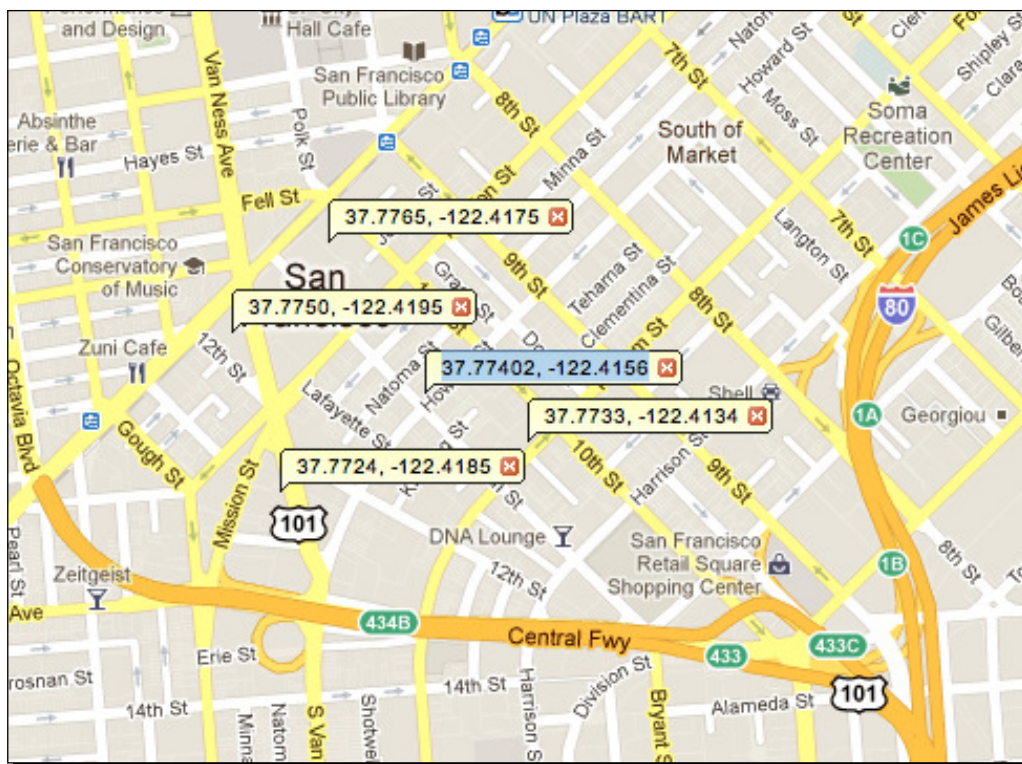
An **overlay** is a layer of multiple map co-ordinates, used to represent activity or information over a significant geographical region. While annotations are single map co-ordinates, overlays are a group or set of co-ordinates layered on a map surface, their size being connected to the zoom level of the map. Overlays help in analyzing a region for certain behavior or accessing and representing statistical data on a region on the map.

Overlays can be used to show city boundaries on a map, a good way to check this in action is at Flickr's Geo API Explorer - <http://www.flickr.com/places/info/12587707> for San Francisco.

Overlays are quite similar to annotations – in the sense that both are superimposed layers on top of the map view. Our application needs to provide an object that conforms to the `MKOverlay` protocol and the `MKOverlayView` that draws the overlay on the map; this is similar to the use of the `MKAnnotation` protocol and `MKAnnotation` used in our annotation examples earlier. iOS provides some built-in overlays that can be used to draw a circle, polygon, or a polyline. These are subclasses of the `MKOverlay` protocol, namely, `MKCircle`, `MKPolygon`, and `MKPolyline`. Similarly, `MKOverlayView` has the subclasses `MKCircleView`, `MKPolygonView`, and `MKPolylineView`.

Time for action – customizing map annotations

Let us quickly see an example of an overlay in action. We create a polygon overlay over the San Francisco region. The values of the polygon are obtained from Google Maps, as shown in the following image:



1. From the earlier example, we are using the code for custom annotations and adding an overlay to the same. In the `Hello_LocationViewController.h` file, declare an `MKPolygon` that will hold our polygon values as `MKPolygon *yourArea;` and expose it as a property.
2. In the `viewDidLoad` method, we create an array of `CLLocationCoordinate2D` that will hold our polygonal values. We then initialize the array with the four latitude/longitude values obtained from Google maps as follows:

```
CLLocationCoordinate2D area[4];
area[0]=    CLLocationCoordinate2DMake(37.7750, -122.4195);
area[1]=    CLLocationCoordinate2DMake(37.7765, -122.4175);
area[2]=    CLLocationCoordinate2DMake(37.7733, -122.4134);
area[3]=    CLLocationCoordinate2DMake(37.7724, -122.4185);
```

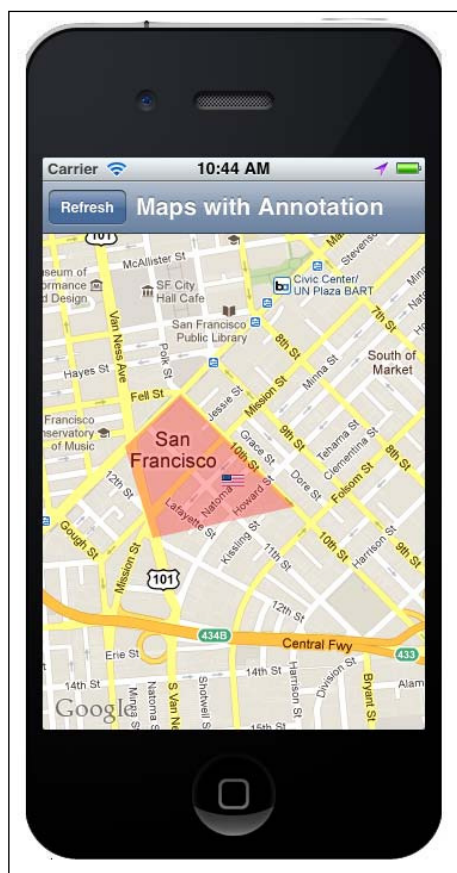
3. We create the `MKPolygon` with these coordinates, assign it a title, and add it to the map as follows:

```
yourArea    =    [MKPolygon polygonWithCoordinates:area count:4];
yourArea.title    =    @"San Francisco Central Area";
[mapView addOverlay:yourArea];
```

4. Next, we implement the `viewForOverlay` delegate method and fill the overlay with the color red, adding some alpha component to make the layer a bit transparent.

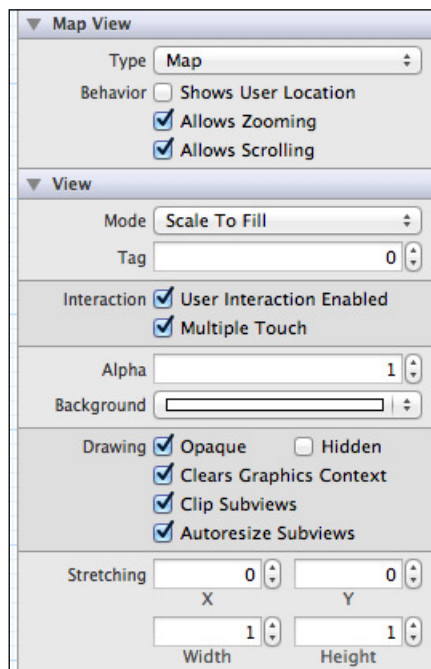
```
- (MKOverlayView *)mapView:(MKMapView *)mapView
viewForOverlay:(id<MKOverlay>)overlay
{
    MKPolygonView* overlayView = [[MKPolygonView alloc]
initWithPolygon:(MKPolygon*)overlay];
    overlayView.fillColor = [[UIColor redColor]
colorWithAlphaComponent:0.3];
    return overlayView;
}
```

5. Run the application, this time without location simulation, as we are hardcoding the polygon values, so it is a good idea to center it to a location we know to understand the output. We have centered the map at San Francisco (37.77402, 122.4156). The overlay on the map should be as shown in the following output:



User tracking modes

One of the default behaviors of maps is detecting and showing the user's location. The `MKMapView` component has an easy way of turning it on. When in the **Interface builder mode** in Xcode, you can select the `mapView` component on your `Nib` file and then navigate to the **Attributes** inspector to enable the **Show User Location** checkbox.



iOS 5 also introduces User Tracking modes via `mapView`, using the `setUserTrackingMode` method of `MKMapView`. This allows the `MapView` to track a user's location via two tracking modes, which are as follows:

- ◆ `MKUserTrackingModeFollow` – the map is updated as the user's location is updated.
- ◆ `MKUserTrackingModeFollowWithHeading` – the map updates its position from the user's location and rotates based on the heading value.

Bonus – offline maps in your app

Google maps, as well as Bing Maps, work well for network-connected iOS devices, but there are no options available yet (**Offline Maps** are available on Android, but not iOS). Enter **CloudMade** - <http://cloudmade.com/>, a company offering solutions for building location-based map apps, using the OpenStreetMaps as the mapping solution.

There are other apps for iOS using **OpenStreetMaps**. You can check out a comprehensive list at http://wiki.openstreetmap.org/wiki/Apple_iOS. This should give you an idea on how OpenStreetMaps data can be consumed in iOS apps. Coming back to CloudMade, the company offers a comprehensive suite of Mapping APIs and SDKs. Our interest lies in the iPhone and iPad SDK – <http://cloudmade.com/products/iphone-sdk>.

The CloudMade iPhone SDK allows more functionality than Google Maps, including support for the following:

- ◆ Offline maps
- ◆ Customizable map styles
- ◆ Vehicle and pedestrian routing
- ◆ Data Market Place, free and paid datasets to be consumed by map-based apps – <http://datamarket.cloudmade.com/>
- ◆ Forward and Reverse Geocoding
- ◆ Location-based advertising

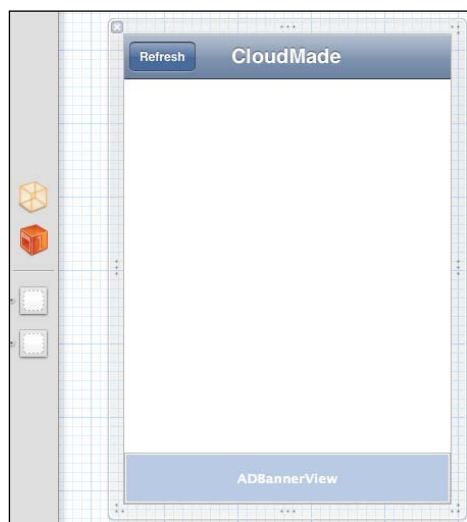
Developers need to sign up for access and get an API to begin building apps using CloudMade iOS SDK. There are limitations with the free API. A comparison of free and paid services can be found at <http://cloudmade.com/select/web>. After you have downloaded the `iphone-sdk`, a good place to start building your apps using CloudMade API can be found at the following websites:

- ◆ [http://developers.cloudmade.com/wiki/iphone-sdk/Examples - Getting started](http://developers.cloudmade.com/wiki/iphone-sdk/Examples-Getting-started)
- ◆ [http://support.cloudmade.com/forums/iphone-sdk/posts/104/show - Offline maps](http://support.cloudmade.com/forums/iphone-sdk/posts/104/show-Offline-maps)

Time for action – using OpenStreetMaps with CloudMade API

Let's build a sample OpenStreetMaps-based app quickly using the CloudMade iPhone Framework.

1. Before you can start using the CloudMade API in your iOS app, you need to register for a key as well as download the CloudMade iPhone Library. See http://developers.cloudmade.com/wiki/iphone-sdk/How_to_get_development_environment_and_download_the_latest_version_the_CloudMade_iPhone_Library
2. Next, we need to add this library to our Xcode project and set up Xcode to work with the newly-added CloudMade Library. A detailed step-by-step guide is available at http://developers.cloudmade.com/wiki/iphone-sdk/How_to_setup_Xcode_to_work_with_CloudMade_iPhone_Library, don't tear your hair out if you have difficulty compiling and configuring the same, as you can always download this example from the book's website.
3. We begin our app by using the getting started app from the CloudMade Wiki http://developers.cloudmade.com/wiki/iphone-sdk/Creating_the_simplest_application_displaying_the_map as the base for our app. We also add our Hello Location process to the same.
4. We added two UIViews and changed the second UIView to the class type `RMMapView` from the CloudMade library and this new MapView is superimposed on our parent view. We also added an iAd placeholder to the `mapView` to show relevant advertisement on the app (don't forget to include the iAd Framework in your Project Build Settings). Our MapView now looks like the following screenshot:

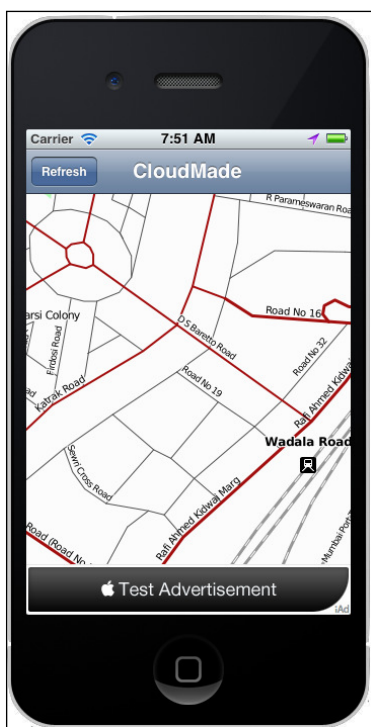


5. In the `Hello_Location_CloudMadeViewController.h` file, we declare the CloudMade MapView as `RMMMapView* mapView;`. To use this, we need to import the appropriate class from the CloudMade library – `#import "RMMMapView.h"`
6. In our implementation, we change the `viewDidLoad` method and instantiate the CloudMade MapView as follows:

```
id cmTilesource = [[RMCloudMadeMapSource alloc] initWithAccessKey:
    @"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" styleNumber:2];

[[RMMMapContents alloc] initWithView:mapView tilesource:
    cmTilesource];
```
7. Replace `xx` with your own key from CloudMade.
8. The CloudMade MapView uses a simpler method to change its position and zooming, which is based on popular JavaScript Map notations (Google Maps JS API):

```
[mapView moveToLatLng: userLocation];
[mapView.contents setZoom: 16];
```
9. Run the application on the iOS Simulator or on the iPhone. You should see an output as follows:



What just happened?

We used the CloudMade `mapView` class and rendered it in our application using the methods exposed by the same. We also created an `iAd` in our application that can be used to display location-based advertisements. Remember our discussion in *Chapter 2*? `iAds` can also track a user's location to show relevant ads, thereby leading to more clicks and hence revenues for the app developers.

CloudMade iPhone SDK has a lot more potential than the basic example we saw earlier. For more comprehensive examples of CloudMade iPhone SDK, visit <http://developers.cloudmade.com/wiki/iphone-sdk/Examples>. A lot of companies have used the CloudMade SDK to create compelling offline-based applications. Once again, you can find the code for this example on the book's website, in the project titled *Hello-Location-CloudMade*.

Have a go hero – creating an offline map

Push yourself up to coach and try to make your app work offline by using offline maps. For a start, see <http://support.cloudmade.com/answers/offline-maps>

The advantages of offline maps are aplenty. First of all, it is a great user experience and provides an increase in loading speed. Secondly, it saves the user's mobile billing charges by serving maps from the offline store. So no hits to the Google Maps API or Bing Maps API to fetch a new map image. Thirdly, the app can work even in remote locations where 3G or other network connections are sparse.

Pop quiz – Map Mania

1. Which projection model does Google Maps use?
 - a. What is a projection model?
 - b. Gall–Peters projection
 - c. Mercator projection
2. What is the difference between `region.center` and `map.centerCoordinate` properties?
 - a. No difference, both achieve the same result
 - b. The former changes the zoom level, but the later does not
3. What is the difference between `MKPinAnnotationView` and `MKAnnotationView`?
 - a. `MKAnnotationView` is used for custom markers, while `MKPinAnnotationView` does not support custom images for markers
 - b. `MKPinAnnotationView` is the standard view and used for custom markers for the Maps

Summary

In this chapter, we learned about the MapKit Framework of iOS SDK. We also understood map geometry.

Specifically, we covered:

- ◆ MapKit and maps geometry
- ◆ Using MapKit in our applications
- ◆ Adding annotations, custom annotations, and draggable annotations
- ◆ Adding Overlays on a map
- ◆ Introduction to CloudMade SDK for Maps

Now that we know how to handle Location and Maps in iOS 5, let's create a real world *Weather app* using all that we have learned so far.

5

Weather App—WeatherPackt

A Weather app is a nifty app for mobile phone devices. It is a default app that is bundled with most phones. We will learn how to build our own Weather App for iOS devices, using the WeatherBug API. You need to register for a key at the following URL: <http://weather.weatherbug.com/desktop-weather/api.html>.

ProgrammableWeb lists a collection of Weather APIs. You can choose any Weather API provider from: <http://www.programmableweb.com/apis/directory/1?apicat=Weather&sort=date>

In this chapter, we will cover the following topics:

- ◆ Storing and retrieving the user's location data with SQLite
- ◆ Converting location data into city name, using GeoNames API
- ◆ Consuming the WeatherBug API in your app
- ◆ Building your Weather app
- ◆ Customizing Weather content display
- ◆ App settings pages
- ◆ Using PhoneGap to build WeatherPackt
- ◆ Bonus: text to speech

So let's get on with it...

Storing and retrieving the user's location with SQLite

We will use an SQLite database to store and retrieve the user's location. In addition to this, we will also store the **place** information through the GeoNames API. We can use the Core Data framework of the iOS SDK for similar purposes, but since our data will not be overwhelmingly large, SQLite is a good choice of storing the same. If you have used databases, such as MySQL or Postgres before, you will find the SQL statements among them to be similar. For performance-based apps, you should read the *Core Data Performance Guide* at the Apple developer site: <http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreData/Articles/cdPerformance.html>.

This should help you best prepare for using Core Data versus SQLite in your applications.

Time for action – storing and retrieving the user's location with SQLite

We use the Hello Location – Location update example from *Chapter 3, Using Location in your iOS Apps – Core Location*, to demonstrate the SQLite functionality.

1. Open the Hello Location – Location update example, and add the SQLite library to your project. The library is named `libsqlite3.0.dylib`. Alternatively, you can include the `libsqlite3.dylib` library (which is a symbolic link to the `libsqlite3.0.dylib` library).
2. In the `Hello_LocationViewController.h` file, we include the `sqlite3` header by importing the `sqlite3.h` file as follows:

```
#import <sqlite3.h>
```
3. Next, we declare a variable database of type `sqlite3`, and a string to hold the full filename with the following path:

```
sqlite3      *database;  
NSString     *sqliteFileName;
```
4. Next, we define a method named `getDatabaseFullPath` that returns the full path to the user's Document folder on his iPhone. This path will be used to store the SQLite database that we will create and use in our application.

```
-(NSString *) getDatabaseFullPath;
```
5. We then create an `IBAction` that will use the SQLite database to retrieve the user's last position, by reading it from the SQLite database using the SQL statements.

```
-(IBAction)getSqliteLocation:(id)sender;
```

- 6.** In the `Hello_LocationViewController.m` file, we declare a character variable that holds the errors received by SQLite within our application code.

```
char *sqliteError;
```

We also declare a variable to hold the database table name.

```
NSString *tableName = @"user_position";
```

- 7.** Before creating or using our database, we need to define the full path where the database file will reside on the iOS device. For this purpose, we use the `NSDocumentDirectory` foundation data type, which is declared in `Foundation.h` file.
- 8.** We define the `getDatabaseFullPath` method as follows:
- ```
{
 NSArray *directoryPath = NSSearchPathForDirectoriesInDomains
 (NSDocumentDirectory, NSUserDomainMask, TRUE);
 NSString *documentsDirectory = [directoryPath objectAtIndex:0];
 return [documentsDirectory stringByAppendingPathComponent:@"location.db"];
}
```
- 9.** The `NSArray` `directoryPath` variable contains the list of directory search paths, in this case, the directory path of the user's Documents directory, specified by the `NSDocumentDirectory` parameter in the `NSSearchPathForDirectoriesInDomains` method. We use this path, and append the filename of our SQLite database `location.db`, and return it back to the calling method.
- 10.** Next, in our `didUpdateToLocation` method, we open the SQLite file, create the table to store the user's location, and start inserting rows (holding the user location information variables `newLatitude` and `newLongitude`).
- 11.** The `sqlite3_open` method is used to open a database. If the database is present, this method will open the database, otherwise if there is no database, it will create one and open it. The function will return a response type to indicate the status of the database asked for. The results code also applies for other SQLite methods.

|                              |   |                                     |
|------------------------------|---|-------------------------------------|
| <code>SQLITE_OK</code>       | 0 | Successful result                   |
| <code>SQLITE_ERROR</code>    | 1 | SQL error or missing database       |
| <code>SQLITE_INTERNAL</code> | 2 | Internal logic error in SQLite      |
| <code>SQLITE_PERM</code>     | 3 | Access permission denied            |
| <code>SQLITE_ABORT</code>    | 4 | Callback routine requested an abort |
| <code>SQLITE_BUSY</code>     | 5 | The database file is locked         |
| <code>SQLITE_LOCKED</code>   | 6 | A table in the database is locked   |

|                   |     |                                                |
|-------------------|-----|------------------------------------------------|
| SQLITE_NOMEM      | 7   | A malloc() failed                              |
| SQLITE_READONLY   | 8   | Attempt to write a readonly database           |
| SQLITE_INTERRUPT  | 9   | Operation terminated by<br>sqlite3_interrupt() |
| SQLITE_IOERR      | 10  | Some kind of disk I/O error occurred           |
| SQLITE_CORRUPT    | 11  | The database disk image is malformed           |
| SQLITE_NOTFOUND   | 12  | Unknown opcode in sqlite3_file_control()       |
| SQLITE_FULL       | 13  | Insertion failed because database is full      |
| SQLITE_CANTOPEN   | 14  | Unable to open the database file               |
| SQLITE_PROTOCOL   | 15  | Database lock protocol error                   |
| SQLITE_EMPTY      | 16  | Database is empty                              |
| SQLITE_SCHEMA     | 17  | The database schema changed                    |
| SQLITE_TOOBIG     | 18  | String or BLOB exceeds size limit              |
| SQLITE_CONSTRAINT | 19  | Abort due to constraint violation              |
| SQLITE_MISMATCH   | 20  | Data type mismatch                             |
| SQLITE_MISUSE     | 21  | Library used incorrectly                       |
| SQLITE_NOLFS      | 22  | Uses OS features not supported on host         |
| SQLITE_AUTH       | 23  | Authorization denied                           |
| SQLITE_FORMAT     | 24  | Auxiliary database format error                |
| SQLITE_RANGE      | 25  | 2nd parameter to sqlite3_bind out of range     |
| SQLITE_NOTADB     | 26  | File opened that is not a database file        |
| SQLITE_ROW        | 100 | sqlite3_step() has another row ready           |
| SQLITE_DONE       | 101 | sqlite3_step() has finished executing          |

- 12.** Once the database is created, we create a table called `user_position`, having the following columns: `position_id`, `latitude`, `longitude`, and `placeName`. We create this using the `sqlite3_exec()` method, and on success, we insert the location values obtained through the location manager. Here is the code for the full `didUpdateToLocation` method:

```
- (void) locationManager: (CLLocationManager *)
manager didUpdateToLocation: (CLLocation *)newLocation
fromLocation: (CLLocation *)oldLocation
{
 NSString *newLatitude = [[NSString alloc] initWithFormat:@"%g",
newLocation.coordinate.latitude];
 NSString *newLongitude= [[NSString alloc] initWithFormat:@"%g",
newLocation.coordinate.longitude];
```

---

```

latitudeTextData = newLatitude;
longitudeTextData = newLongitude;

latitudeText.text = latitudeTextData;
longitudeText.text = longitudeTextData;

if(sqlite3_open([sqliteFileName UTF8String],
&database)==SQLITE_OK)
{

 NSString *sql = [[NSString alloc] initWithFormat:
@"CREATE TABLE IF NOT EXISTS '%" ('position_id' INTEGER PRIMARY
KEY,'latitude' DOUBLE, 'longitude' DOUBLE, 'placeName'
VARCHAR)",tableName];

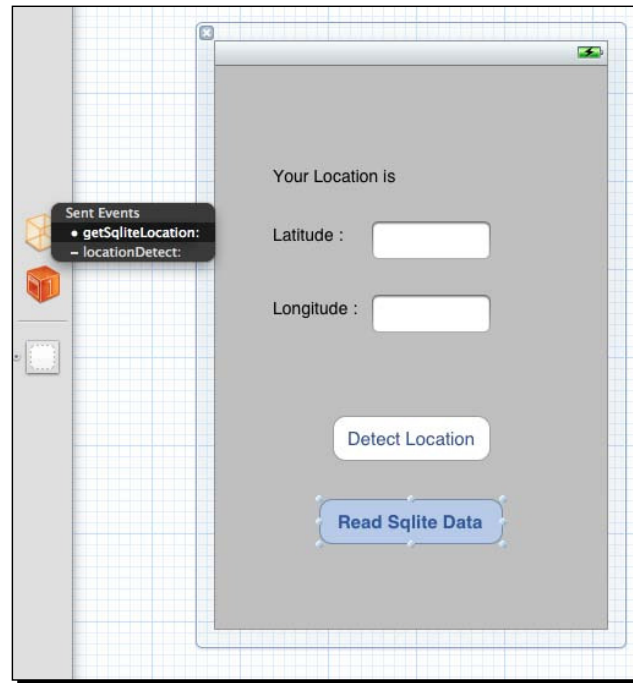
 if(sqlite3_exec(database, [sql UTF8String], NULL, NULL,
&sqliteError)==SQLITE_OK)
 {
 insertStatement = [[NSString alloc] initWithFormat:
@"INSERT OR REPLACE INTO '%" ('%', '%')
VALUES('%', '%')",tableName,@"latitude",@"longitude",
newLatitude,newLongitude];

 if(sqlite3_exec(database, [insertStatement UTF8String],
NULL, NULL, &sqliteError)==SQLITE_OK)
 {
 NSLog(@"Location Inserted");
 }
 }
}
}

```

- 13.** Now that we have successfully inserted the values in the database table, let's write some code to retrieve these values.

- 14.** Open your `Hello_LocationViewController.xib` file, create another round rect button, name it `Read Sqlite Data`, and connect it to `IBAction getLocation` by pressing `Control` key, and clicking-and-dragging the pointer to file's owner in the Interface Builder.



- 15.** We declare the `getLocation` method as follows:

```
- (IBAction) getLocation: (id) sender
{
 NSString *sql = [[NSString alloc] initWithFormat:
 @"SELECT * FROM '%@' where position_id =
 (select max(position_id) from '%@')", tableName, tableName];

 sqlite3_stmt *sqlStatement;
 if (sqlite3_prepare_v2(database, [sql UTF8String], -1,
 &sqlStatement, NULL) == SQLITE_OK)
 {
 while (sqlite3_step(sqlStatement) == SQLITE_ROW)
 {
 double latitudeData = sqlite3_column_double(sqlStatement, 1);
 double longitudeData = sqlite3_column_double(sqlStatement, 2);
 }
 }
}
```

---

```

 NSString *returnLat = [NSString stringWithFormat:
 @"Your double value is %f", latitudeData];
 NSString *returnLon = [NSString stringWithFormat:
 @"Your
double value is %f", longitudeData];

 NSLog(returnLat);
 NSLog(returnLon);

 }
}
}

```

- 16.** We retrieve the user's last location inserted in the table, by using a nested SQL query that retrieves the last row. The combined `sqlite3_prepare` and `sqlite3_step` methods are used to loop through the results of the SQL query. The `sqlite3_column_text` method retrieves the column specified. We use the first and second columns' values for retrieving the latitude and longitude values. Note the table structure, since the column numbers (starting with index 0) will be defined by the sequence of your SQL `Create` statement.
- 17.** Run the app in the iOS simulator, try changing a couple of location values through the **Product | Debug | Simulate Location** menu option, and observe the values in the **Debug** window.

```

2011-08-28 17:05:47.643 Hello Location Sqlite[1651:f203] Location
Inserted
2011-08-28 17:05:49.978 Hello Location Sqlite [1651:f203] 37.7874
2011-08-28 17:05:49.980 Hello Location Sqlite [1651:f203] -122.408
2011-08-28 17:05:59.681 Hello Location Sqlite [1651:f203] Location
Inserted
2011-08-28 17:06:01.626 Hello Location Sqlite [1651:f203] 19.0176
2011-08-28 17:06:01.628 Hello Location Sqlite [1651:f203] 72.8562
2011-08-28 17:06:15.131 Hello Location Sqlite [1651:f203] Location
Inserted
2011-08-28 17:06:17.145 Hello Location Sqlite [1651:f203] -33.8634
2011-08-28 17:06:17.147 Hello Location Sqlite [1651:f203] 151.211

```

Find the code for this example on the book's website: project titled *Hello Location - Location Updates - SQLite*.



## ***What just happened?***

We extended the **Hello Location** app again, and created a SQLite database - `location.db` that resides on our iOS device user's Document directory, and holds the table called `user_position`, which in turn, contains the user's raw location values.

When we read the SQLite database, we retrieve the user's last location, since this will be the user's last and most updated position, and it makes good app behavior to continue from there.

Our approach to storing the location is similar to the `consolidated.db` approach that Apple took with its location tracking Fiasco. This was detected by *Pete Warden*, and published at: <http://petewarden.github.com/iPhoneTracker/>. It caused a lot of security uproar for Apple, so it is a good idea to encrypt this file, and keep it private to your application alone.

## **Converting location data into city name – using Geonames API**

Now that we have the user's location stored in our iOS device, we assume that the user does not change his location often, or does not often move out of city. We use the GeoNames API to convert the user's position into a meaningful city name or area name, as returned by the GeoNames API. We could also use the reverse Geocoding method provided by the new `CLGeocoder` class in iOS 5. Time to revisit the Geocoding example we did in *Chapter 3*, where we covered forward geocoding. Now, we will look at reverse geocoding and converting latitude/longitude values to meaningful address.

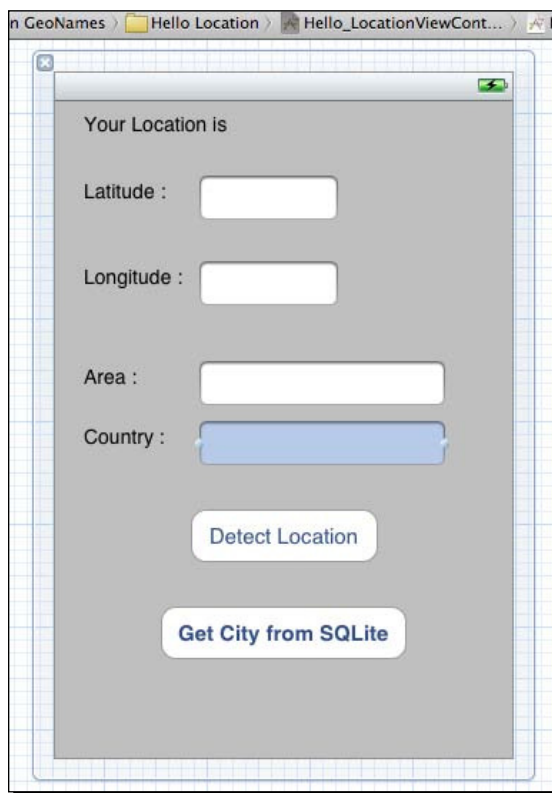
## **A bit on GeoNames**

GeoNames is a worldwide geographical database, with a creative common license, containing more than 10 million geographical names that could include city, street, administrative areas, mountains, lakes, canals, and so on . A full list is available at: <http://www.geonames.org/export/codes.html>. The database is available for download, and there is a web service as well.

## Time for action – converting location data into city name

To display the city in which the user is interacting with our `Weather` application, we need to convert the latitude/longitude pairs to an appropriate city or local area name, so that it makes visual sense to the end user.

1. Before we can use the GeoNames API, you need to register and get your own username with `GeoNames.org` through <http://www.geonames.org/login>. The documentation for the web service can be found at: <http://www.geonames.org/export/ws-overview.html>.
2. We begin extending the SQLite example discussed previously, by adding two `UILabels` for city and country to our `Hello_LocationViewController.xib` file, as well as by adding two `UITextFields` that will be use to render the city and country text values.



- 3.** We will need an `XMLParser` object to parse the XML response from GeoNames. We created a similar example when we used the Last.FM API in *Chapter 3*, and we will reuse most of the code here as well. Our `Hello_LocationViewController.h` file now looks as follows:

```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>
#import <sqlite3.h>

@interface Hello_LocationViewController : UIViewController
<CLLocationManagerDelegate, NSXMLParserDelegate>
{
 IBOutletUITextField *latitudeText;
 IBOutletUITextField *longitudeText;
 CLLocationManager *locMgr;
 CLLocation *userLocation;
 NSString *message;
 sqlite3 *database;
 NSString *sqliteFileName;
 NSXMLParser *xmlParser;
 NSURLConnection *urlConnection;
}

@property (retain, nonatomic) IBOutletUITextField *latitudeText;
@property (retain, nonatomic) IBOutletUITextField *longitudeText;
@property (strong, nonatomic) IBOutletUITextField *area;
@property (strong, nonatomic) IBOutletUITextField *country;

- (NSString *) getDatabaseFullPath;

- (IBAction)locationDetect:(id)sender;
- (IBAction)getSqliteLocation:(id)sender;

@end
```

- 4.** The `area` and `country` variables are required to hold the values of city and country, respectively. The `getSqliteLocation` method will be used to retrieve the user's last location and city/country values.

5. In our `Hello_LocationViewController.m` file, as soon as the `didUpdateToLocationEvent` is called when the device location is updated, we do the following:
  - ❑ Call the GeoNames API through a `NSURLRequest`. We use the `http://api.geonames.org/findNearbyPlaceName` GeoNames API to find the place nearest to the latitude/longitude values provided.
  - ❑ On success of the place nearest to the latitude/longitude values provided, the `XMLParser` takes over, using the `didStartElement` and `didEndElement` method, to parse the XML data for city and country values contained in the fields `name` and `countryName`.
  - ❑ After the XML parsing finishes, we use the SQLite `insert` statements to insert the user's latitude, longitude, place (read city), and country values into the SQLite database table named `user_position`.
6. Open the XIB file in Interface Builder, and *Control*+drag the mouse pointer from the **Get City from SQLite** button to the **File's Owner**, and select the event `getSqliteLocation`.
7. We now define the actions for `getSqliteLocation` as follows:

```
- (IBAction)getSqliteLocation: (id) sender
{
 NSString *sql = @"SELECT * FROM user_position where position_id
 =(select max(position_id) from user_position)";
 sqlite3_stmt *sqlStatement;

 if(sqlite3_prepare_v2(database, [sql UTF8String], -1,
 &sqlStatement, NULL)==SQLITE_OK)
 {
 while(sqlite3_step(sqlStatement)==SQLITE_ROW)
 {
 const unsignedchar *latitudeData =
 sqlite3_column_text(sqlStatement, 1);

 const unsignedchar *longitudeData =
 sqlite3_column_text(sqlStatement, 2);

 const unsigned char *placeData =
 sqlite3_column_text(sqlStatement, 3);
```

```
NSString *returnLat = [[NSStringalloc] initWithFormat:
 @"",latitudeData];

NSString *returnLon = [[NSStringalloc] initWithUTF8String:
 longitudeData];

NSString *returnPlace = [[NSStringalloc] initWithUTF8String:
 placeData];

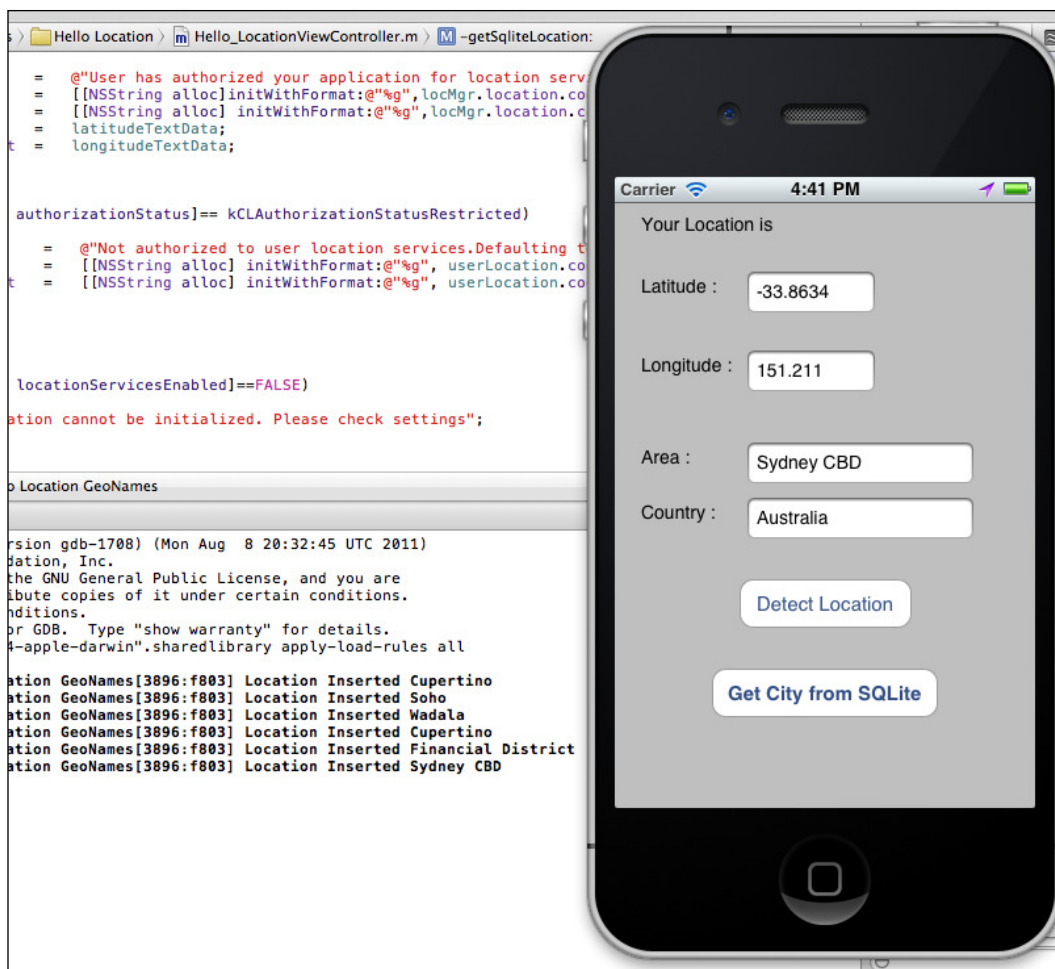
NSLog(returnLat);
NSLog(returnLon);
NSLog(returnPlace);

 }
}
}
```

8. Run the app in the iOS simulator, and use a couple of location values to simulate your app behavior. You should see the following response in the **Debug** window:

```
2011-09-04 16:40:09.421 Hello Location GeoNames[3896:f803]
Location Inserted Cupertino
2011-09-04 16:40:33.977 Hello Location GeoNames[3896:f803]
Location Inserted Soho
2011-09-04 16:40:42.230 Hello Location GeoNames[3896:f803]
Location Inserted Wadala
2011-09-04 16:40:48.889 Hello Location GeoNames[3896:f803]
Location Inserted Cupertino
2011-09-04 16:40:55.913 Hello Location GeoNames[3896:f803]
Location Inserted Financial District
2011-09-04 16:41:04.692 Hello Location GeoNames[3896:f803]
Location Inserted Sydney CBD
```

You can find the code for this example on the book's website: project titled *Hello Location - Location Updates with GeoNames*.



The GeoNames API that we used (<http://api.geonames.org/findNearbyPlaceName>), return to us the closest place, which need not necessarily be a city name; it could be the name of a street, locality, or another administrative area. If you need the city name compulsorily, then the new `CLGeocoder` class in iOS 5, specifically the reverse geocoder would be helpful. It does the same task as the GeoNames API, but returns a better place information through the `CLPlacemark` object.

## What just happened?

We enhanced the `Hello Location` SQLite example, by not only storing the user's latitude and longitude pair, but also by converting the same into readable city and country values. These values can further be used in our `Weather App` project.

We also extended the SQLite database by adding the place and country fields in the `user_position` table. So, a row in the database table now contains the user's latitude, longitude, place name, and country name.



You can verify the data inserted in this table, by using the **Xcode | Organizer** tool. With the new features of iOS 5, you can download your app data, modify it, and then re-insert it on your device. With your device selected in **Organizer**, select the **Application** name; in our example it should be `Hello Location Sqlite 1.0`, and then in the **Documents Tree View**, you should see the `location.db` file.

Use the **Download** button to download the file on your desktop; the file should be named as `com.packt.Hello-Location-Sqlite 2011-09-04 17.46.40.751.xcappdata`. Open the file by right-clicking the **Context Menu** and selecting the **Show Package Contents** option. Find the `location.db` file in the **AppData | Documents** folder, and modify it with the **Firefox SQLite manager** (available at <https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/>) or any SQLite database editor. Once done, just upload the `com.packt.Hello-Location-Sqlite 2011-09-04 17.46.40.751.xcappdata` file to your iOS device again.

## Consuming the WeatherBug API

Now that we have secured our foundation for building the `Weather` app, it's time to look at the `WeatherBug` API, understand the API calls, and understand how we can build our app around it. For the purpose of our `Weather App`, we are considered about the following `Weather` API calls: **Live Weather**, **Forecast**, and **Alerts**. Let us look at how the `WeatherBug` API solves our requirement for the three mentioned `Weather` queries.

## Important things to know before we begin

Register for the `WeatherBug` API at <http://weather.weatherbug.com/desktop-weather/api-register.html>.

The documentation can be found at <http://weather.weatherbug.com/desktop-weather/api-documents.html>.

Keep the API key handy.

The following Weather services are offered by WeatherBug:

|                        |                       |                |
|------------------------|-----------------------|----------------|
| Location search        | Weather Camera search | Live Weather   |
| Compact Live Weather   | Forecast              | Weather Alerts |
| Weather Station search |                       |                |

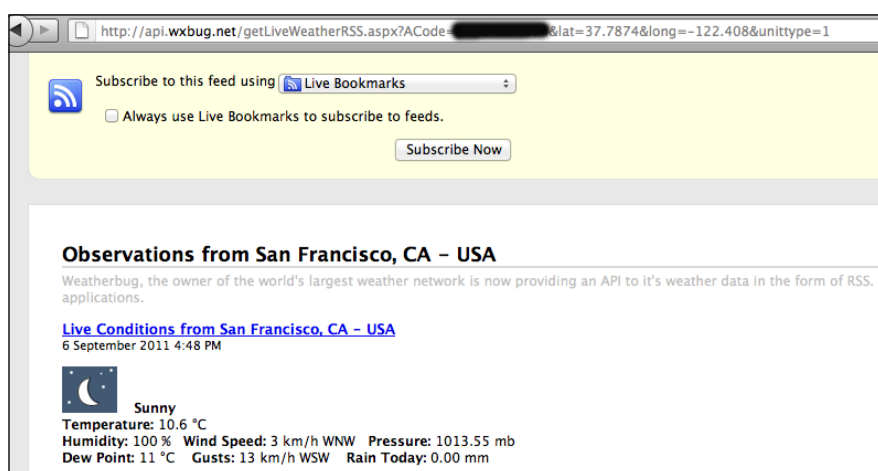
We will focus on the API calls for

- ◆ **Live Weather:** `http://api.wxbug.net/getLiveWeatherRSS.aspx?ACode=xxxxxxxxx&lat=latitude&long=longitude&unitttype=0|1`
- ◆ **Forecast:** `http://api.wxbug.net/getForecastRSS.aspx?ACode=xxxxxxxxx&lat=latitude&long=longitude&unitttype=0|1`
- ◆ **Weather Alerts:** `http://api.wxbug.net/getAlertsRSS.aspx?ACode=xxxxxxxxx&lat=latitude&long=longitude`

Here

- ◆ xxxxxxxxxxxx is the API key
- ◆ latitude is our iOS device's latitude value
- ◆ longitude is our iOS device's longitude value
- ◆ unitttype=0 is for Fahrenheit display
- ◆ unitttype=1 is for Celsius display

You can check out the API URLs in a browser. Since the API output is RSS, it should render well on any browser; the idea is to get a rough image on how the output is organized. We ran the API URL on Firefox to constantly monitor and compare it with the data that we receive in the app. This helped us verify our application logic as well.





## Time for action – using WeatherBug API

Let's create a barebones app that uses the three WeatherBug API discussed in the previous section. We will use the `Hello Location` example again, but this time, we make the UI a bit of a run-of-the-mill iOS app UI.

We will create an app that will detect the location, convert the location to city name, fetch weather information, and forecast and display the same in a `UITableView`.

1. We begin declaring the variables that will be used in our app, open `Hello_LocationViewController.h`, and add a variable, `weatherTable`, for the `UITableView - UITableView` `*weatherTable;`

2. Next, we declare the variables that will be used to hold the Live Weather information:

```
NSString *weatherIcon;
NSString *weatherConditions;

NSString *weatherTemperature;
NSString *weatherTemperatureUnit;

NSString *weatherHumidity;
NSString *weatherHumidityUnit;

NSString *weatherWindSpeed;
NSString *weatherWindSpeedUnit;

NSString *weatherPressure;
NSString *weatherPressureUnit;

NSString *weatherDewPoint;
NSString *weatherDewPointUnit;

NSString *weatherGusts;
NSString *weatherGustsUnit;

NSString *weatherRainToday;
NSString *weatherRainTodayUnit;
```

3. We then declare an array of the type `NSMutableArray`, which will hold the weather information in an array format to be rendered on the `UITableView`.

```
NSMutableArray *weatherdataArray;
```

4. For the Forecast data, we declare the following variables:

```
NSString *dayTitle;
NSString *dayPrediction;

NSString *mondayForeCast;
NSString *tuesdayForeCast;
NSString *wednesdayForeCast;
NSString *thursdayForeCast;
NSString *fridayForeCast;
NSString *saturdayForeCast;
NSString *sundayForeCast;
```

5. We declare individual functions for Live Weather, Forecast, and Alerts as follows:

```
-(IBAction) showLiveWeather: (id) sender;
-(IBAction) showForeCast: (id) sender;
-(IBAction) showAlerts: (id) sender;
```

6. Do not forget to include the `NSXMLParserDelegate` and the `UITableViewDataSource` delegate in your header file.
7. Now, let's design our UI. Open the `Hello_LocationViewController.xib` file. To make our app visually appealing, we incorporate some icons. We have used icons designed by *Joseph Wain*: <http://glyphish.com>. He has put up some nice icons for both: the iPhone and the iPad applications. Download the free icons with *Creative Commons License* from <http://glyphish.com/download/glyphish-icons.zip> and, and unzip it in your application's main folder.

8. Next, we create a toolbar at the header of the View, and add four bar button items, one button each for refresh, Live Weather, forecast, and Alert, respectively. We also place a `UITableView` that will hold our dynamic Weather data, as well as some labels and textboxes to show the location information. Here is how our final app should look like:



9. Time to connect the dots; *Control*+drag the mouse pointer from the refresh button to the File's Owner, and select the `getSqliteLocation` action. Similarly, connect the Live Weather button to the `showLiveWeather` action, the Forecast button to the `showForeCast` action, and Alert button to the `showAlerts` action. Notice the **Bar Item** carefully, where we have used the icons we downloaded before. Xcode automatically allows you to choose from the icons downloaded. So, for the Forecast button, we choose the `25-weather.png` file, and similarly for the other buttons.



- 10.** Having created the UI, we now proceed to write the implementation for the `showLiveWeather`, `showForeCast`, and `showAlert` methods. Open the `Hello_LocationViewController.m` file to detect which method of the Weather we are accessing. We create three Boolean variables to keep a track, as well as add one variable for the WeatherBug API key.

```
NSString *apiCode=@"xxxxxxxxxxx";

bool inGeoNames = FALSE;
bool inLiveWeather = FALSE;
bool inForeCast = FALSE;
```

- 11.** In the `ViewDidLoad` method, we initialize the variables needed for the weather display.

```
weatherIcon = [[NSString alloc] initWithString:@""];
```

- 12.** In the `didUpdateToLocation` event method in the `LocationManager`, we make a request to the WeatherBug API for Live Weather conditions. This is the default home screen for our app. We initiate `UIActivityIndicatorView`, to show the **loading...** effect, while our app fetches the Weather Info over the Web service. We also change its visibility to hidden, when we stop the `loadingIcon` after our Web service calls completes successfully.

```
loadingIcon.hidesWhenStopped=TRUE;
[loadingIconstartAnimating];
[selfshowLiveWeather:self];
```

- 13.** In the `showLiveWeather` method, we initialize the `weatherDataArray`, and set the `inLiveWeather` flag to true. We then call the Weather Bug Live Weather API, open a `NSURLConnection`, and proceed to complete the URL request.

```
-(IBAction) showLiveWeather: (id) sender
{
 [loadingIconstartAnimating];
 weatherDataArray = [[NSMutableArrayalloc] init];
 [weatherTablereloadData];

 inGeoNames = FALSE;
 inLiveWeather= TRUE;
 weatherBugUrl= [[NSStringalloc] initWithFormat:
@"http://api.wxbug.net/getLiveWeatherRSS.aspx? ACode=%@&lat=%@&lon
g=%@&unittype=1",
 apiCode,latitudeText.text,longitudeText.text];

 NSURL *urlToRequest = [[NSURLlalloc]
```

```
initWithString:weatherBugUrl];
NSURLRequest *request = [NSURLRequest
requestWithURL:urlToRequest];

urlConnection = [[NSURLConnection alloc]
initWithRequest:request
delegate:self startImmediately:YES];

}
```

- 14.** We use the `didStartElement`, `didEndElement`, and `foundCharacters` method of the `NSXMLParser`, to populate our variables used for the weather display, and pass it on to the `WeatherdataArray`, which in turn renders it on the `UITableView`.

```
-(void)parser:(NSXMLParser *)parser didStartElement:(NSString
*)elementName namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName attributes:(NSDictionary *)
attributeDict
{
currentXMLTitle=[elementName copy];
if([currentXMLTitle isEqualToString:@"name"]
{
cityText = [[NSMutableString alloc] init];
}

if([currentXMLTitle isEqualToString:@"countryName"])
{
countryText = [[NSMutableString alloc] init];
}

// In Live Weather
if(inLiveWeather)
{
if([currentXMLTitle isEqualToString:@"aws:current-condition"])
{
currentConditionsText = [[NSMutableString alloc] init];
weatherIcon=[attributeDict objectForKey:@"icon"];
}

if([currentXMLTitle isEqualToString:@"aws:temp"])
{
weatherTemperatureUnit=[attributeDict objectForKey:@"units"];
}
```

---

```

 weatherTemperatureUnit=[weatherTemperatureUnit
 stringByReplacingOccurrencesOfString:@"°"
 withString:@"°"];

 }

 // Parse other RSS fields for humidity,wind-speed,etc

 }

 // End of Live Weather
}

-(void)parser:(NSXMLParser *)parser didEndElement:(NSString
*)elementName namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName
{
 if([currentXMLTitle isEqualToString:@"name"])
 {
 area.text = cityText;
 cityText = [[NSMutableStringalloc]init];
 }

 if([currentXMLTitle isEqualToString:@"countryName"])
 {
 country.text = countryText;
 countryText = [[NSMutableStringalloc]init];
 }

 if([currentXMLTitle isEqualToString:@"aws:current-condition"])
 {
 currentConditionsText = [[NSMutableStringalloc]init];
 }

}

-(void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)
string
{
 if([currentXMLTitle isEqualToString:@"name"])
 {
 [cityTextappendString:string];
 }
}

```

```
 }

 if ([currentXMLTitle isEqualToString:@"countryName"])
 {
 [countryTextappendString:string];
 }

 // For Live Weather
 if (inLiveWeather)
 {
 inForeCast=FALSE;
 if ([currentXMLTitle isEqualToString:@"aws:current-condition"])
 {
 weatherConditions=string;
 }

 if ([currentXMLTitle isEqualToString:@"aws:temp"])
 {
 weatherTemperature=[string
 stringByTrimmingCharactersInSet:[NSCharacterSet
 whitespaceAndNewlineCharacterSet]];
 }

 if ([currentXMLTitle isEqualToString:@"aws:humidity"])
 {
 weatherHumidity=[string
 stringByTrimmingCharactersInSet:[NSCharacterSet
 whitespaceAndNewlineCharacterSet]];;
 }

 }
 // End of Live Weather

 // In ForeCast
 if (inForeCast)
 {
 inLiveWeather=FALSE;
 if ([currentXMLTitle isEqualToString:@"aws:forecast"])
```

```

 {
 }

 if([currentXMLTitle isEqualToString:@"aws:title"])
 {
 dayTitle = [string
 stringByTrimmingCharactersInSet:[NSCharacterSet
 whitespaceAndNewlineCharacterSet]];
 }

 if([currentXMLTitle isEqualToString:@"aws:short-prediction"])
 {
 dayPrediction=[string
 stringByTrimmingCharactersInSet:[NSCharacterSet
 whitespaceAndNewlineCharacterSet]];

 if([dayTitle isEqualToString:@"Monday"])
 {
 mondayForeCast = [[dayTitlestringByAppendingFormat:@"%: "]
 stringByAppendingFormat:dayPrediction];
 }

 if([dayTitle isEqualToString:@"Tuesday"])
 {
 tuesdayForeCast = [[dayTitlestringByAppendingFormat:@"%:
 "] stringByAppendingFormat:dayPrediction];
 }

 }
}

// End of ForeCast
}

- (void)parserDidEndDocument:(NSXMLParser *)parser
{
 if(inLiveWeather)
 {

```



```
 [loadingIconstartAnimating];
 NSString *temp = weatherConditions;

 if (![weatherdataArraycontainsObject:temp])
 {
 [weatherdataArrayaddObject:temp];
 }

 temp=[@"Temperature: "
stringByAppendingFormat:[weatherTemperature
stringByAppendingFormat:weatherTemperatureUnit]];

 if (![weatherdataArraycontainsObject:temp] ||
 ![weatherTemperatureisEqualToString:@""])
 {
 [weatherdataArrayaddObject:temp];
 }

 temp=[@"Humidity: "stringByAppendingFormat:[weatherHumidi
ty
stringByAppendingFormat:weatherHumidityUnit]];

 if (![weatherdataArraycontainsObject:temp] ||
 ![weatherHumidityisEqualToString:@""])
 {
 [weatherdataArrayaddObject:temp];
 }
 // Prepare similar statement for Pressure, Wind Speed,etc

 [loadingIconstopAnimating];

 [weatherTablereloadData];

 }

 if(inForeCast)
 {
 [loadingIconstopAnimating];
 if (![mondayForeCastisEqualToString:@""])
 {
```

---

```

 [weatherdataArrayaddObject:mondayForeCast];
 }

 if (![tuesdayForeCastisEqualToString:@""])
 {
 [weatherdataArrayaddObject:tuesdayForeCast];
 }
 // Add forecast variables for wed, thurs, Friday, etc

 [loadingIconstopAnimating];
 [weatherTablereloadData];
}

```

- 15.** And finally, we define the `showForeCast` method that loads the forecast data from the WeatherBug API as follows:

```

- (IBAction) showForeCast: (id) sender
{
 inLiveWeather = FALSE;
 weatherIcon = [[NSStringalloc] initWithString:@""];
 weatherConditions = [[NSStringalloc] initWithString:@""];

 weatherTemperature= [[NSStringalloc] initWithString:@""];
 weatherTemperatureUnit = [[NSStringalloc] initWithString:@""];

 weatherHumidity = [[NSStringalloc] initWithString:@""];
 weatherHumidityUnit = [[NSStringalloc] initWithString:@""];

 weatherWindSpeed = [[NSStringalloc] initWithString:@""];
 weatherWindSpeedUnit = [[NSStringalloc] initWithString:@""];

 weatherPressure = [[NSStringalloc] initWithString:@""];
 weatherPressureUnit = [[NSStringalloc] initWithString:@""];

 weatherDewPoint = [[NSStringalloc] initWithString:@""];
 weatherDewPointUnit = [[NSStringalloc] initWithString:@""];

 weatherGusts = [[NSStringalloc] initWithString:@""];
 weatherGustsUnit = [[NSStringalloc] initWithString:@""];

 weatherRainToday = [[NSStringalloc] initWithString:@""];
}

```

```
weatherRainTodayUnit = [[NSStringalloc]initWithString:@""];

[loadingIconstartAnimating];
weatherdataArray = [[NSMutableArrayalloc]init];

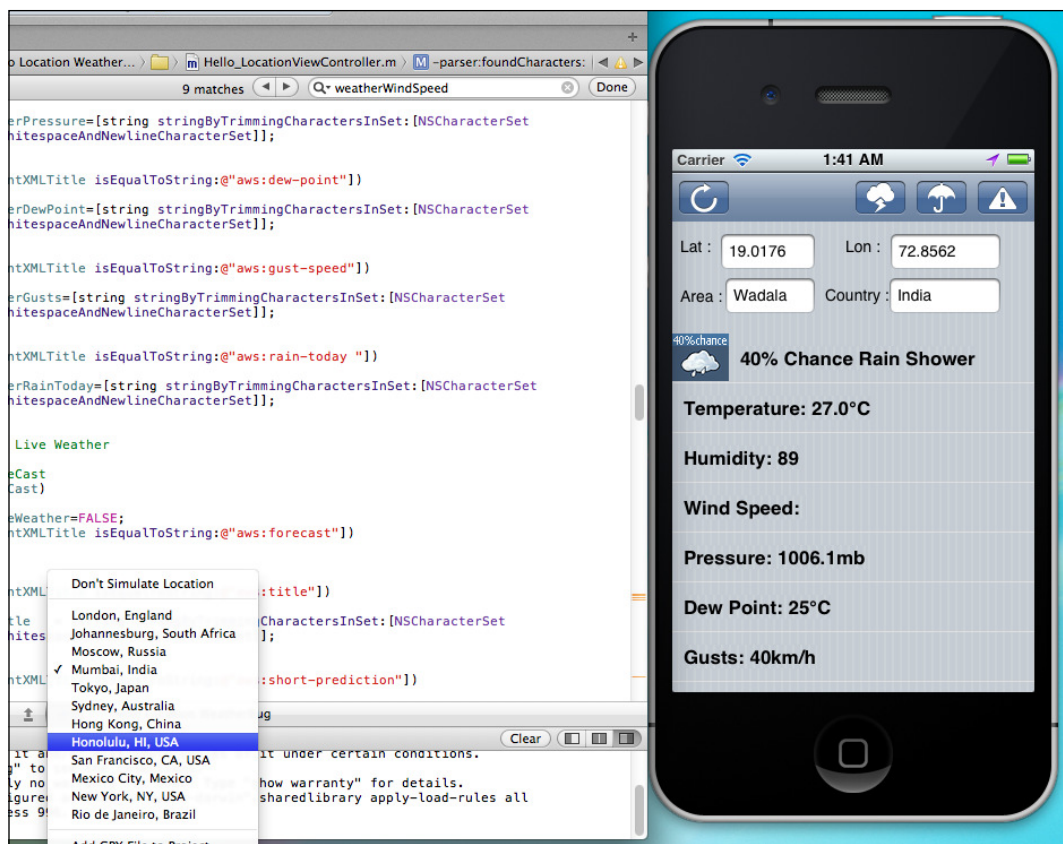
[weatherTablereloadData];

inGeoNames = FALSE;
inForeCast = TRUE;
weatherBugUrl = [[NSString
 alloc]initWithFormat:@"http://api.wxbug.net/getForecastRSS.
aspx?
 ACode=%@&lat=%@&long=%@&unitttype=1",apiCode,latitudeText.text,
 longitudeText.text];

NSURL *urlToRequest = [[NSURL
 alloc]initWithString:weatherBugUrl];
NSURLRequest *request = [NSURLRequest
 requestWithURL:urlToRequest];

urlConnection = [[NSURLConnectionalloc]
 initWithRequest:request
 delegate:selfstartImmediately:YES];
}
```

- 16.** Run the app in the emulator. Try changing a couple of location values through the **Location** icon over the **Debug** window. Your output should be similar to the following screenshot:



## What just happened?

We created a nifty Weather app using GeoNames and WeatherBug. We also used some free icons for our app UI. We learnt how to use the WeatherBug API calls for Live Weather and Forecast, and how to call them dynamically.

We also learnt how to control the XML parsing with flags in our code. The `inForecast` and `inLiveWeather` flags used in the code, helps us keep track of the XML parsing for elements, and help us parse the XML smartly. This does away the purpose of having tree-driven XML parsing in our apps (which is memory hogging).

You can find the code for this example on the book's website: project titled `Hello Location WeatherBug`.

## Have a go hero – creating the Weather Alert function

Try your hands at adding the Weather Alert in the app. From our initial discussion, use the following API URL: `http://api.wxbug.net/getAlertsRSS.aspx?ACode=xxxxxxxxxxx&lat=latitude&long=longitude`.

## Building your Weather App: WeatherPackt

Having looked at the Live Weather and the Forecast API, and understanding how to parse the XML response received from the WeatherBug API, let's use our previous example to build our final WeatherPackt application. Once done building the app, we will also submit our app to the Apple iTunes Store!!

For our WeatherPackt app, we will use the iOS 5 reverse geocoder as the primary source of converting the latitude/longitude to city and country name. You can use the GeoNames API as well, but the `CLGeocoder` class included in iOS 5 helps us obtain the same results easily, in a better-organized format.

Also, keep in mind the response format of the Weather API. Depending upon your country or location, the appropriate metric of display (Celsius or Fahrenheit) should be used. We accomplish this by using a **Settings** page in our app, which gets registered in the global Settings app on your iOS device.

## Start a new Xcode project

We begin our first iOS app by creating a new Xcode project, unlike reusing the `Hello Location` template as before. We will also use the Google AdMob Ads iOS SDK to show mobile ads in our application. Follow the as-simple-as-always Google documentation for the same at: <http://code.google.com/mobile/ads/docs/ios/fundamentals.html>.

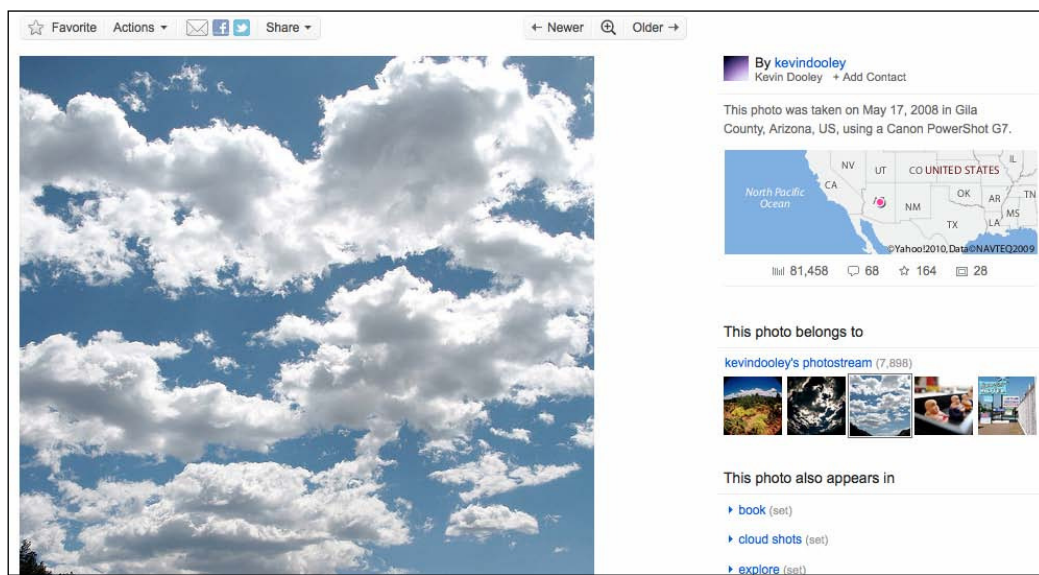
Before we begin our app coding, we need a couple of things to make our app presentable. These include having a nice app icon and a splash image, which will be used as the default app launch image. We source our images from the World Wide Web, using the free-to-use and Creative Commons images in our app. Go through the image licenses carefully; we are using images that are either Creative Commons or free, and have the license to be modified and used commercially.

Images that we will use for our Weather app (henceforth referenced as WeatherPackt) are as follows:

- ◆ **App icon:** We use this image by Jackie Tran (<http://365psd.com/day/2-139/>), and crop the PSD in Photoshop to include on the icon with the rainbow over it as the main app icon. Make sure that each icon fits the size defined by Apple for iPhone display, iPhone retina display, and iPad display. Similarly, for the Splash images, you need to have the right sizes and resolution.



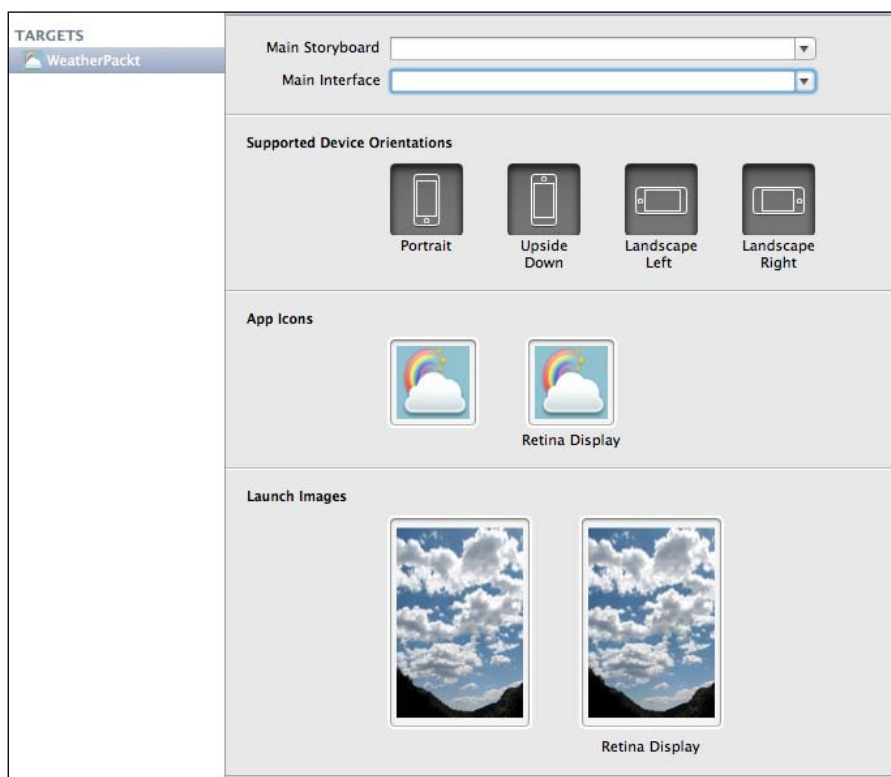
- ◆ **Default background images (launch images):** We use the photo taken by *Kevin Dooley* (<http://www.flickr.com/photos/pagedooley/2511369048/>). We could use any other image, if needed. A Google advanced search for *Clouds*, gave us Kevin's wonderful image, and it looks great for our app requirement.





A bit of image editing experience would be required to crop the images, as needed. The source code has the right sizes for the icons and the splash image, both for iPad and iPhone, including retina display.

We will be using the **Tabbed Application** template from the **Xcode | New Project Area**. For our *WeatherPackt* application, the first tab will be utilized for the main Weather information, the second tab will be used for app **Settings**. It is a good time to try out the Tabbed application template, to get a hang of the project template and tabs management. Our *WeatherPackt* app's **Summary** page in Xcode, should look as follows:



The app icon should look similar to this iPhone Simulator:



## Define the Home screen

We create a new Tabbed application with Xcode, however, the default template for the Tabbed application has only two screens. For our Weather app we need three screens: one for the Live Weather, another for the Forecast, and the third for Weather Alerts. So, we add another `UIViewController` subclass to our project, and name it `ThirdViewController`. We also add an extra XIB file for the iPad template, since the `UIViewController` subclass addition to our project only gives us the iPhone XIB file.

### Time for action – defining the Home screen

1. Open the `AppDelegate.m` file in your newly created WeatherPackt application project. Modify the `didFinishLaunchingWithOptions` method to include the third view controller in the `TabBarController`.

```
if ([[UIDevicecurrentDevice] userInterfaceIdiom] ==
 UIUserInterfaceIdiomPhone)
{
 viewController1 = [[FirstViewControlleralloc] initWithNibN
ame:@"FirstViewController_iPhone"bundle:nil];

 viewController2 = [[SecondViewControlleralloc] initWithNib
Name:@"SecondViewController_iPhone"bundle:nil];
```



```
 viewController3 = [[ThirdViewControlleralloc] initWithNibName:@"ThirdViewController_iPhone"bundle:nil];

 }
 else {
 viewController1 = [[FirstViewControlleralloc] initWithNibName:@"FirstViewController_iPad"bundle:nil];

 viewController2 = [[SecondViewControlleralloc] initWithNibName:@"SecondViewController_iPad"bundle:nil];

 viewController3 = [[ThirdViewControlleralloc] initWithNibName:@"ThirdViewController_iPad"bundle:nil];
 }
}
```

- 2.** In each individual ViewController main file, for example in your SecondViewController.m file, add the following code for the AdMob ads integration in your viewDidLoad method:

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

 deviceType = @"iPhone";

 NSString *model= [[UIDevicecurrentDevice] model];
 NSRange range = [model rangeOfString:@"iPhone"];

 if(range.location == NSNotFound)
 {
 deviceType = @"iPad";
 }
 else
 {
 deviceType = @"iPhone";
 }

 responseData = [[NSMutableDataalloc] init];

 // AdMob Code Starts
 // Create a view of the standard size at the bottom of the screen.
 if([deviceTypeisEqualToString:@"iPhone"]){
 bannerAdView = [[GADBannerViewalloc]
```

```

initWithFrame:CGRectMake(0.0,43.0,
GAD_SIZE_320x50.width,GAD_SIZE_320x50.height)];
 }else{
bannerAdView = [[GADBannerViewalloc]
initWithFrame:CGRectMake(20.0,43.0,
GAD_SIZE_728x90.width,GAD_SIZE_728x90.height)];
 }

bannerAdView.adUnitID = @"xxxxxxxxxxxxxxxxx ";

bannerAdView.rootViewController = self;
 [self.viewaddSubview:bannerAdView];

// Initiate a generic request to load it with an ad.
 [bannerAdViewloadRequest:[GADRequestrequest]];

// AdMob Code Ends
}

```

Where xxxxxxxxxxxxxxxx is your publisher ID from <http://admob.com>.

Based on the device type and the app UI, we adjust the AdMob code accordingly.

- 3.** Time to use the icons for the tab bars in the `SecondViewController.m` file. Modify the `initWithNibName` method to include the icon images from <http://glyphish.com>. Note that you also need to add the `glyphish-icons` folder to your Xcode project.

```

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
nibBundleOrNil
{
self = [superinitWithNibName:nibNameOrNil bundle:nibBundleOrNil];
if (self) {
self.title = NSLocalizedString(@"Forecast", @"Forecast");
self.tabBarItem.image = [UIImageimageNamed:@"99-umbrella"];
}
returnself;
}

```

4. If all goes well, your Home screen should now look as the one shown in the following screenshot:



5. We create a similar layout for the Forecast page in the `SecondViewController_iPhone.xib` file.

## Set up a default location

To make sure that our app works well in case the user has not allowed the location services or in case the user's location has not been determined or could not be determined, we initiate the app loading process with a default location; in this case, we load San Francisco by using the following code:

```
newLatitude = @"37.33";
newLongitude = @"-122.03";

locationManager = [[CLLocationManager alloc] init];
locationManager.desiredAccuracy = kCLLocationAccuracyBest;
locationManager.distanceFilter = 1000.0f;
locationManager.delegate = self;
userLocation = [[CLLocation alloc] initWithLatitude:
[newLatitudedoubleValue] longitude:[newLongitudedoubleValue]];
```

As the user's new location is detected, we update the `userLocation` variable to always hold the updated values.

## Formatting the Weather API for display

As we saw in the previous examples, we used the `didStartElement`, `foundCharacters`, `didEndElement`, and `parserDidEndDocument` methods of the `XMLParser`, to fetch and display the weather information. However, the logic we used before was not perfect. It assumed that the XML response is streamlined, and each method calls the subsequent method sequentially. But in the real-world scenario, this would be a bit different. Depending upon your `NSURL` and network connections, the `foundCharacters` method of the `NSXMLParser` class can be called multiple times for the same XML tag. This would create problems in our earlier examples, but we will refactor the code to take care of this issue as well.

In our `foundCharacters` method, we keep appending the string response received to a temporary variable until the `didEndElement` method confirms the end of the XML tag, and resets the temporary variable.

```
- (void)parser: (NSXMLParser *)parser
 foundCharacters: (NSString *)string{
 if (!currentXMLValue)
 {
 currentXMLValue = [[NSMutableStringalloc]init];
 }
 [currentXMLValueappendString:string];
}
```

## The settings page

We want our `WeatherPackt` app to have a settings page, registered in the main `Settings` app of your iOS device, which can be used to flip the display from Celsius to Fahrenheit, and vice versa. We do this by adding a `Settings` bundle to our application. The `Settings` bundle helps us manage preferences from within the `Settings` application.

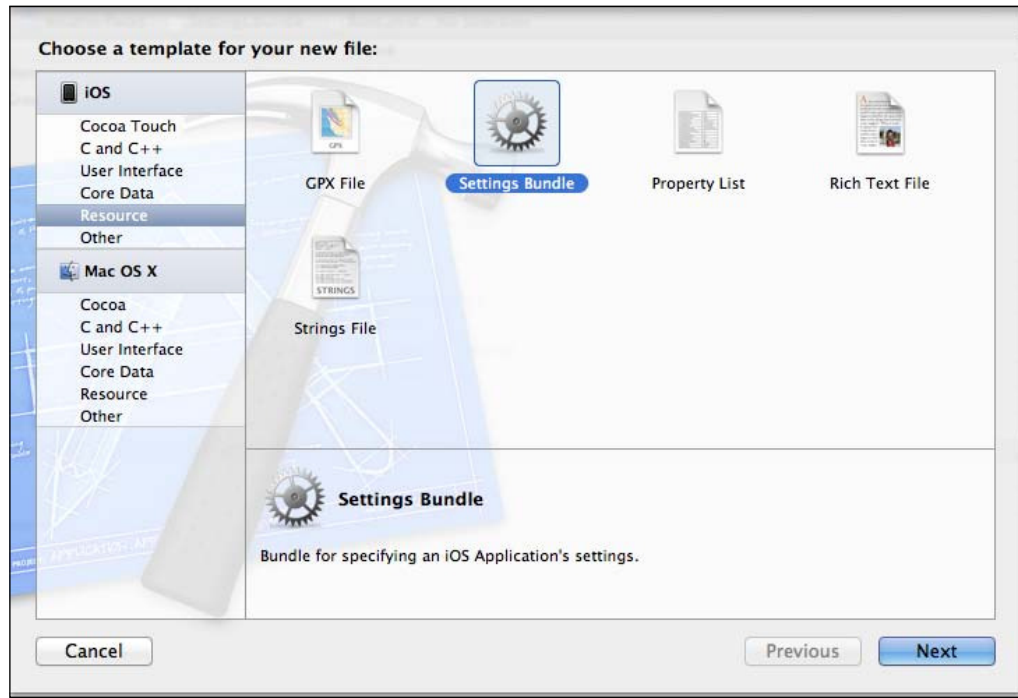
The `NSUserDefaults` class is used to access the settings/preference values. The type of settings we can incorporate in our `Settings` page could be the following:

- ◆ A slider
- ◆ A text field
- ◆ A Title
- ◆ A toggle switch (we will use this for Celsius display on/off)
- ◆ A group
- ◆ A child pane
- ◆ A multi value

More details can be found at Apple's rich documentation available at:

<http://developer.apple.com/library/ios/#DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/Preferences/Preferences.html>.

We will create a simple settings page for our app. Using the **File | New File** option, and selecting the **Resources** tab from the **Modal** window, we add the setting bundle, name the file as `Settings Bundle`, and save it in your project.



The `Settings Bundle` is just a collection of two files: `Root.plist` and `Root.string`. Double-click on the `Root.plist` file in Xcode, and open the **Settings Property List** editor, where you can define the **Preference** items for your **Settings Page**.

Modify the `Root.plist` file to look as follows:

| Key                                | Type       | Value                 |
|------------------------------------|------------|-----------------------|
| ▼ Preference Items                 | Array      | (3 items)             |
| ▼ Item 0 (Group – WeatherPackt     | Diction... | (2 items)             |
| Title                              | String     | WeatherPackt Settings |
| Type                               | String     | Group                 |
| ▼ Item 1 (Text Field – Temperature | Diction... | (8 items)             |
| Autocapitalization Style           | String     | None                  |
| Autocorrection Style               | String     | No Autocorrection     |
| Default Value                      | String     |                       |
| Text Field Is Secure               | Boolean    | NO                    |
| Identifier                         | String     | name_preference       |
| Keyboard Type                      | String     | Alphabet              |
| Title                              | String     | Temperature Settings  |
| Type                               | String     | Text Field            |
| ▼ Item 2 (Toggle Switch – Celsius) | Diction... | (4 items)             |
| Default Value                      | Boolean    | YES                   |
| Identifier                         | String     | enabled_preference    |
| Title                              | String     | Celsius               |
| Type                               | String     | Toggle Switch         |
| Strings Filename                   | String     | Root                  |

We use the **Toggle Switch** with **Identifier** as `enabled_preference`, to allow the users to switch the Celsius display on/off. This identifier is also used in our application to fetch the current value of its state, the code for which is as follows:

```
// Read the Settings

NSUserDefaults *settings = [NSUserDefaults standardUserDefaults];
celsiusValue= [settings valueForKey:@"enabled_preference"];

// End of Read Settings
```

When we run the application and open the main iOS device's **Settings** screen, we should see the following option for WeatherPackt:



And its associated **Settings**:



The full code for the `WeatherPackt` app can be found on the book's website: project titled `WeatherPackt`.

### Have a go hero – adding the Alerts page to WeatherPackt

We showed you how to build a Live Weather and Forecast page. The app framework also supports a third view for displaying Weather Alerts. We have made provisions for both the iPhone and iPad View in the application. Using your knowledge of what you have learnt so far, complete the third page. Feel free to share the code; who knows your page could end up in the final app and on iTunes !!

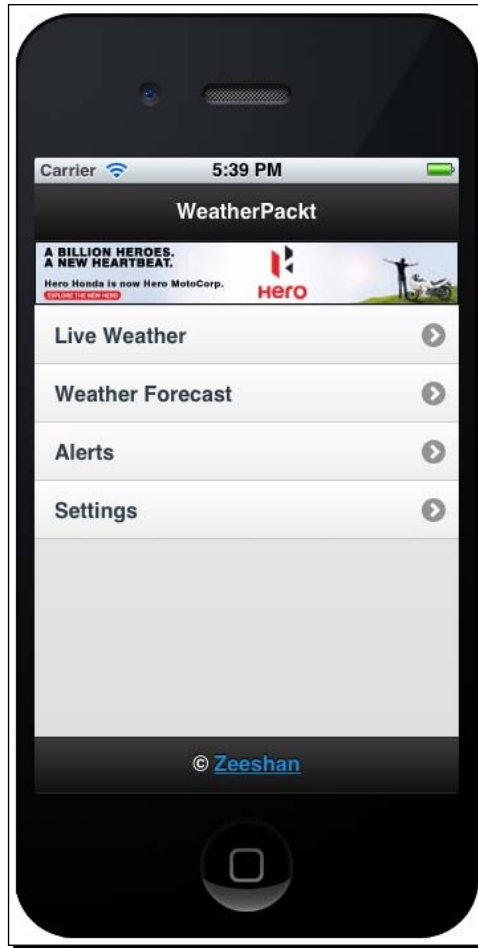
Here is how the `WeatherPackt` app should look, when you run it in the iOS simulator:





## Bonus: building WeatherPackt with PhoneGap

As a bonus, also find the WeatherPackt app done with PhoneGap on the book's website: project titled WeatherPackt - PhoneGap. Here is how it looks:



## Bonus: text-to-speech

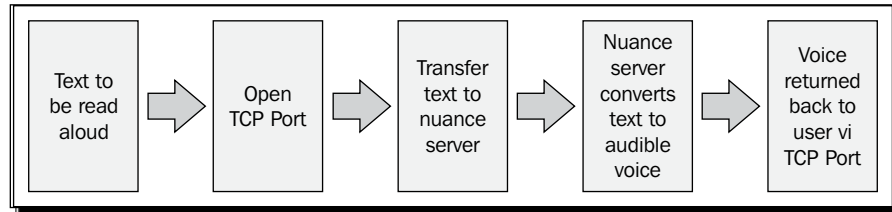
In our WeatherPackt app, we add the **Nuance Mobile SDK** to enable text-to-speech conversion within our app. You need to register with Nuance (<http://dragonmobile.nuancemobiledeveloper.com>) to get a development and production key to be used in your application. Here is how the text-to-speech function will look in our app.



Follow the easy to use documentation at: [http://dragonmobile.nuancemobiledeveloper.com/public/Help/DragonMobileSDKReference\\_iOS/Introduction.html](http://dragonmobile.nuancemobiledeveloper.com/public/Help/DragonMobileSDKReference_iOS/Introduction.html), to start using the Nuance SDK. Here is how we tied the microphone UIButton to a simple IBAction that has only two lines of code:

```
- (IBAction)speakText:(id)sender {
 NSString *stringToSpeak = [[NSString alloc] initWithFormat:
 @"Weather Today is %@",weatherConditions];
 [vocalizer speakString:stringToSpeak];
}
```

The conversion from text-to-speech from the user end to the Nuance server, and back, is depicted as follows:



The `SpeechKit` framework from Nuance contains not only text-to-speech functions but speech-to-text functions as well!

## Pop quiz – Weather Alert

1. What does the following code do?

```
NSArray *directoryPath= NSSearchPathForDirectoriesInDomains
(NSDocumentDirectory, NSUserDomainMask, TRUE);
```

- a. Searches for user directory in an array
  - b. Searches for the current user's Document directory and returns an array
2. What is the XML parsing flow using `NSXMLParser` class and delegates?
- a. `didStartElement -> didEndElement -> foundCharacters`
  - b. `didStartElement->foundCharacters->didEndElement`
  - c. `didStartElement->foundCharacters->didEndElement->parserDidEndDocument`

## Summary

In this chapter, we learned how to store location data into a SQLite database for offline support. We also looked at GeoNames and WeatherBug API to build our WeatherPackt application. We also showed a PhoneGap Application for WeatherPackt, so that we can make HTML developers happy. The PacktPub website has an excellent cookbook for building web applications with JavaScript, titled *iPhone JavaScript Cookbook*. Get it from <http://www.packtpub.com/iphone-javascript-cookbook/book>.

Specifically, we covered:

- ◆ SQLite.
- ◆ GeoNames API.
- ◆ WeatherBug API – Live Weather and Forecast.
- ◆ Creating a WeatherApp from scratch with settings.
- ◆ Creating a WeatherApp with PhoneGap.

In the next chapter, we will look at building an Events application with the Eventful API.



# 6

## Events App—PacktEvents

*An events app is a good entertainment companion for your iOS device. By using the events application, a user can browse for nearby gigs, learn about his/her favorite artists, and find events happening at his/her favorite venue.*

*Eventful.com is the leading events and entertainment service that provides real-time events information to millions of users. Eventful's unique killer feature is Demand it!—a service that empowers fans to get their favorite artists/performers to come to their town. We will use Eventful's rich and extensive API to build our events app – henceforth known as **PacktEvents**.*

In this chapter, we will discuss the following topics:

- ◆ PacktEvents – An overview of PacktEvents and a definition of the underlying architecture
- ◆ Storing and retrieving events with SQLite
- ◆ Plotting events on a map
- ◆ Using EventKit API to add events to the iOS calendar
- ◆ Filtering events display by categories
- ◆ Using Twitter integration in iOS 5 to tweet an event
- ◆ PacktEvents – Building the app
- ◆ Bonus: Using Layar Augmented Reality player in PacktEvents.

So let's get on with it...

## PacktEvents: Overview and architecture

We looked at the `Eventful.com` API in *Chapter 3* with a simple application titled *Hello Location – Eventful*.

Now we will build a complete app – **PacktEvents**, similar to **WeatherPackt**, which will be based on the Tabbed Application Xcode project template. However, we will extend it to three tabs: one for **Events**, another for **Venues**, and a third one for **Artists**. Throughout this chapter, we will show you the bits and pieces of the PacktEvents app. The readers are encouraged to put together the examples and build the PacktEvents app themselves. However, the full source code will be available from the book's page at [packtpub.com](http://packtpub.com)

## Architecture of PacktEvents

PacktEvents will be an Offline-Online app, with the app behavior controlled via the **Settings** page. Why Offline-Online and not one of the two? The idea evolved with the recent enhancements in Cloud Computing, 3G, and now 4G Network services, and real-time data availability. It is quite cheap and easy to fetch the nearby events happening from a web service these days. Storing events information (which tends to be very dynamic – consider an artist breaking his leg before the gig, thereby canceling the event at the last hour) offline does not make much sense, due to data integrity issues. However, it makes some sense to store the user's recently-browsed events/venues/artists in the app, so the user can start from where he/she left off (this could be due to a phone call interruption or some other event that causes our app to go to the background).

We will store up to 99 (Applesque number) events, venues, and artists offline on our PacktEvents app and still be real-time by using smart algorithms – if the user changes his location, the cache is cleared.

On the application side, we will use the new iOS 5 Twitter integration to tweet from any page on the app. The tweets could be for the following:

- ◆ Attending an event
- ◆ Liking an artist
- ◆ Tweeting about a venue

We will also use **Nuance Speech Mobile SDK** for iOS to implement Text-to-Speech and/or speech recognition in PacktEvents, so if you want to search for events by "Lady Gaga", you don't have to type it in the search box. Just press the record button and the Nuance Dragon will convert your speech into text and hit the Eventful API for "Lady Gaga" Events. Cool huh!

User-generated content can be handled by our app, but the data will be stored back at Eventful. We will use SQLite for our offline data storage.

For the design, we will use some free icons and background images for our app. We will use the Eventful logo wherever applicable. As Eventful has been generous enough to give us the API access to use in this book, it's time to return the love. For the app icon, we have used the free mobile-icon-set icons provided by [WebIconSet.com](http://WebIconSet.com).

## Storing and Retrieving Events with SQLite

For PacktEvents, we will create a SQLite database called *PacktEvents* that will have five tables: events, venues, artists, events category, and user's location information, respectively. We model our database using the SQLite Manager add-on for Firefox. Download the plugin from <https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/>.

Our Events table's Create SQL statement looks like the following code snippet:

```
CREATE TABLE "events" ("id" VARCHAR PRIMARY KEY NOT NULL UNIQUE ,
 "title" VARCHAR NOT NULL UNIQUE , "description" TEXT, "start_time"
 DATETIME, "end_time" DATETIME, "venue_id" VARCHAR, "latitude"
 DOUBLE, "longitude" DOUBLE, "price" VARCHAR, "images" VARCHAR,
 "category_id" INTEGER)
```

Now, we create the Venues table as follows:

```
CREATE TABLE IF NOT EXISTS "venues" ("id" VARCHAR PRIMARY KEY NOT
 NULL UNIQUE , "title" VARCHAR NOT NULL UNIQUE , "description"
 VARCHAR UNIQUE , "type" VARCHAR, "address" TEXT, "city" VARCHAR,
 "zip" VARCHAR, "country" VARCHAR, "latitude" DOUBLE, "longitude"
 DOUBLE, "images" VARCHAR)
```

Time for the Artist table; we create it as follows:

```
CREATE TABLE IF NOT EXISTS "artists" ("id" VARCHAR PRIMARY KEY NOT
 NULL , "name" VARCHAR NOT NULL UNIQUE , "short_bio" TEXT,
 "long_bio" TEXT, "event_count" INTEGER, "images" VARCHAR)
```

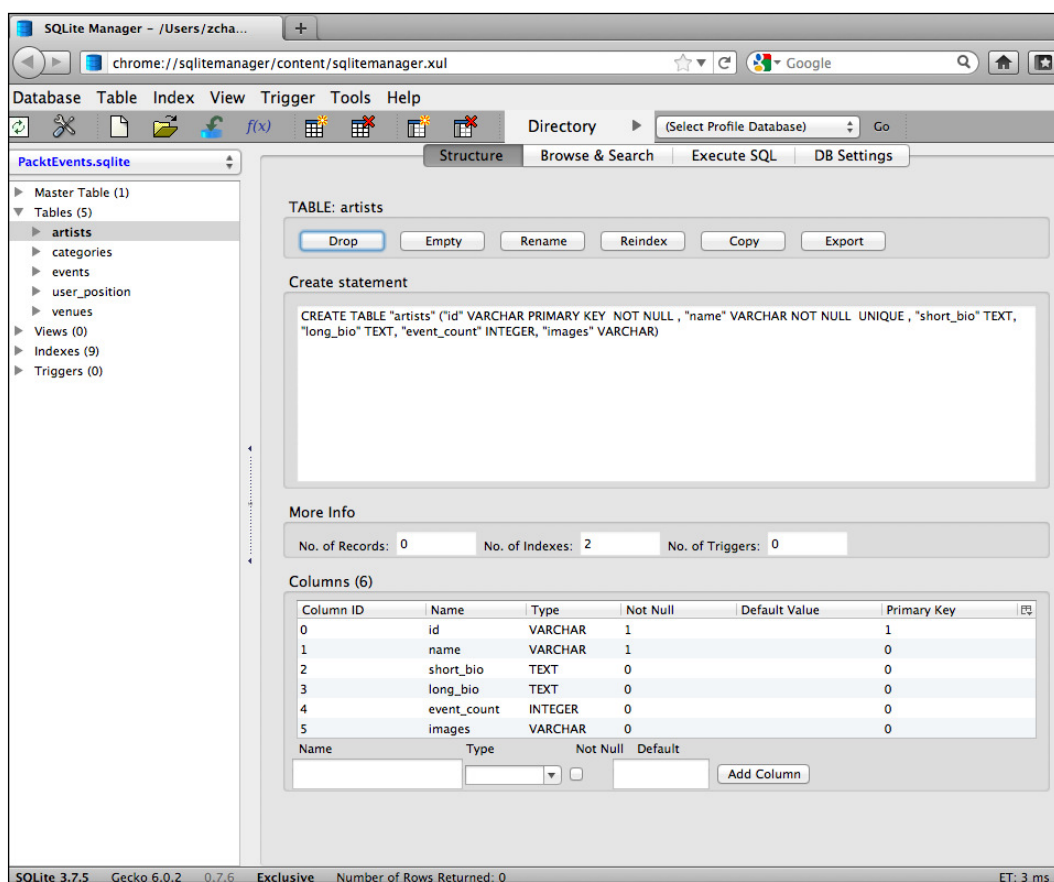
The next table we create is the categories table, which is used just to look up the category ID against a category name and vice versa, as follows:

```
CREATE TABLE IF NOT EXISTS "categories" ("id" VARCHAR PRIMARY KEY
 NOT NULL , "name" VARCHAR NOT NULL UNIQUE)
```



The last table we will create is the User Position table that we have used before to keep track of the user's position, as shown in the following code snippet:

```
CREATE TABLE IF NOT EXISTS "user_position" ("position_id" INTEGER
PRIMARY KEY,"latitude" DOUBLE, "longitude" DOUBLE, "city"
VARCHAR,"country" VARCHAR)
```



## Time for action – storing and retrieving events with SQLite

Now that we have defined how we will store the data received from Eventful API in SQLite, let's create a new series for this chapter, titled `Hello Events`. Note that this time we will use the JSON API from Eventful and the new JSON framework in iOS 5.

1. Create a new `Single View` application from within Xcode and name it **Hello Events-SQLite**.

2. Add the **Core Location** and SQLite libraries to your project. When creating the project, do not forget to name the class prefix as **Hello Events** as well.
3. Our sample reading and writing from/to an SQLite database example will work as follows:
  - ❑ Create a database on the device or open it, if already existing
  - ❑ Read Eventful JSON API
  - ❑ Write events into the **events** table
  - ❑ Read from the events table and render it on a **UITableView**
4. Open the `Hello_EventsViewController.h` file and declare variables for the Core Location Manager, a location object of the type `CLLocation` (for storing the user's latest position), a database object of the type `sqlite3`, and an `UITableView`, as well as a `NSJSONSerialization` object for storing JSON data, and one more variable for a `NSURLConnection` object for making the API Request to Eventful.
5. We also declare function calls for initializing the SQLite database (SQL create statements) and reading from the SQLite database. While reading from the local database is done with the `readEventsFromLocal`, the writing to the SQLite database is done via the Location Manager's `didUpdateToLocation` method, which eventually triggers the `NSURLConnection`'s `didReceiveData` and `connectionDidFinishLoading` methods via our `readEventFulApi` method. The JSON parsing and storing of events in the SQLite database is done in the `connectionDidFinishLoading` method.
6. In our `Hello_EventsViewController.m`, we start with the `viewDidLoad` method and initialize our Core Location objects as before. Then we define three variables, one for storing our JSON data, the second for storing the events data in an array, and the third one for storing our SQLite filename.
 

```
jsonContent = [[NSMutableData alloc] init];
events = [[NSMutableArray alloc] init];
sqliteFileName = [self getDatabaseFullPath];
[self initializeDatabase];
```
7. Note these variables are declared along with the `UITableView` object in the `Hello_EventsViewController.h` header file as follows:

```
UITableView *myTableView;
NSMutableData *jsonContent;
NSMutableArray *events;
NSString *sqliteFileName;
```

- 8.** We then call the `initializeDatabase` method that creates/opens our database and respective tables. However, before we do that, we must declare five variables in our code, which will hold the respective database table names. The following statements are added to the start of our `Hello_EventsViewController.m` file:

```
NSString *eventsTableName = @"events";
NSString *venuesTableName = @"venues";
NSString *artistsTableName = @"artists";
NSString *usersTableName = @"user_position";
NSString *categoriesTableName = @"categories";
```

- 9.** Do not forget to add a `UITableView` to your XIB file, connect the datasource and delegate to the File's Owner placeholder, and connect the outlet `myTableView` to the `UITableView`. Furthermore, implement the `UITableViewDataSource` delegate in your class declaration.

- 10.** We also used some local variables in the implementation of our class – `Hello_EventsViewController.m`. These variables are required to parse JSON and report a SQLite error as follows:

```
char *sqliteError;
NSMutableString *idText,*titleText,
 *descriptionText,*startTimeText,
 *endTimeText, *venueIdText,
 *latitudeText,*longitudeText,*priceText,
 *imagesText, *categoryText;
```

- 11.** While some other variables are used for the SQLite database insertions:

```
NSString *insertStatement,*selectStatement;
NSString *events_sql ,*venues_sql ,*artists_sql,*user_sql
,*category_sql;
```

- 12.** The `initializeDatabase` method is now defined as follows:

```
-(NSString *) initializeDatabase
{
 events_sql = [NSString stringWithFormat:@"CREATE TABLE IF
 NOT EXISTS '%@" ('id' VARCHAR PRIMARY KEY NOT NULL UNIQUE ,
 'title' VARCHAR NOT NULL UNIQUE , 'description' TEXT,
 'start_time' DATETIME, 'end_time' DATETIME, 'venue_id'
 VARCHAR, 'latitude' DOUBLE, 'longitude' DOUBLE, 'price'
 VARCHAR, 'images' VARCHAR, 'category_id'
 INTEGER) ",eventsTableName];

 venues_sql = [NSString stringWithFormat:@"CREATE TABLE IF
 NOT EXISTS '%@" ('id' VARCHAR PRIMARY KEY NOT NULL UNIQUE ,
 'title' VARCHAR NOT NULL UNIQUE , 'description' VARCHAR
```

---

```

 UNIQUE , 'type' VARCHAR, 'address' TEXT, 'city' VARCHAR, 'zip'
 VARCHAR, 'country' VARCHAR, 'latitude' DOUBLE, 'longitude'
 DOUBLE, 'images' VARCHAR)",venuesTableName];

artists_sql = [NSString stringWithFormat:@"CREATE TABLE IF
 NOT EXISTS '%@" ('id' VARCHAR PRIMARY KEY NOT NULL , 'name'
 VARCHAR NOT NULL UNIQUE , 'short_bio' TEXT, 'long_bio' TEXT,
 'event_count' INTEGER, 'images' VARCHAR)",artistsTableName];

user_sql = [NSString stringWithFormat:@"CREATE TABLE IF
 NOT EXISTS '%@" ('position_id' INTEGER PRIMARY KEY, 'latitude'
 DOUBLE, 'longitude' DOUBLE, 'city' VARCHAR, 'country'
 VARCHAR)",usersTableName];

category_sql = [NSString stringWithFormat:@"CREATE TABLE '%@"
 ('id' VARCHAR PRIMARY KEY NOT NULL , 'name' VARCHAR NOT NULL
 UNIQUE)",categoriesTableName];

if(sqlite3_open([sqliteFileName UTF8String],
 &database)==SQLITE_OK)
{
 if(sqlite3_exec(database, [events_sql UTF8String], NULL, NULL,
 &sqliteError)==SQLITE_OK)
 {
 NSLog(@"event table created");
 }
}

if(sqlite3_exec(database, [venues_sql UTF8String], NULL, NULL,
 &sqliteError)==SQLITE_OK)
{
 NSLog(@"venue table created");
}

if(sqlite3_exec(database, [artists_sql UTF8String], NULL, NULL,
 &sqliteError)==SQLITE_OK)
{
 NSLog(@"artist table created");
}

if(sqlite3_exec(database, [user_sql UTF8String], NULL, NULL,
 &sqliteError)==SQLITE_OK)
{
 NSLog(@"user table created");
}

```

---

```
 if(sqlite3_exec(database, [category_sql UTF8String], NULL, NULL,
 &sqliteError)==SQLITE_OK)
 {
 NSLog(@"category table created");
 }

 return @"Successfully Created Database";
 }
 else
 {
 return @"Failed creating database";
 }
}
```

- 13.** Now that the database is ready, we move onto calling the Eventful API via the `didUpdateToLocation` delegate method. After we got the updated latitude and longitude within this method, we invoke the `readEventFulApi` method as follows:

```
[self readEventFulApi];
```

- 14.** The `readEventFulApi` method is pretty straightforward; we construct an `NSURLRequest` with the Eventful JSON API URL, pass it to an `NSURLConnection`, and initiate the request as follows:

```
-(void) readEventFulApi
{
 // Call EventFul API Now

 NSString *appKey = @"xxxxxxxxxxxxxxxxxxxx"; // Get your own key
 from api.eventful.com

 NSString *url = [NSString stringWithFormat:
 @"http://api.eventful.com/json/events/search?location=%@,%@
 &app_key=%@&within=10", newLatitude,newLongitude,appKey];

 NSURL *urlToRequest = [[NSURL
 alloc]initWithString:url];
 NSURLRequest *request = [NSURLRequest
 requestWithURL:urlToRequest];

 urlConnection = [[NSURLConnection alloc]
 initWithRequest:request
 delegate:self startImmediately:YES];
}
```

- 15.** As the request gets connected and we start receiving the data through the `NSURLConnection`'s delegate method – `didReceiveData`, where we keep appending the data until the URL connection has completely received the data. The `didReceiveData` might be called a number of times (based on your iOS device's network connection). So it is a good practice to append the response in one variable and use the variable when the `NSURLConnection`'s `connectionDidFinishLoading` method is called, signaling the end of received data:

```
- (void)connection: (NSURLConnection *)connection
didReceiveData: (NSData *)data
{
 [jsonContent appendData:data];
}
```

- 16.** When the URL connection has completely received the data, the `connectionDidFinishLoading` method is triggered. It is here where the crux of our JSON parsing occurs. We initialize an `NSDictionary` object from the JSON data received by using the `NSJSONSerialization` class as follows:

```
dictionary= [NSJSONSerialization JSONObjectWithData:jsonContent
options:NSJSONReadingAllowFragments error:&jsonError];
```

- 17.** We then convert this dictionary into an array, so we can parse the JSON data:

```
items = [NSArray arrayWithObject:[dictionary
objectForKey:@"events"]];
```

- 18.** Once we are sure that enough events data has been retrieved from the Eventful JSON API by checking if the received events count is at least five, we purge the events table from the database, so that the new ten events are inserted in the same by using the following code snippet:

```
NSUInteger count=[[items
objectAtIndex:0]objectForKey:@"event"] count];

if(count >= 5)
{
 if(sqlite3_exec(database, [@"Delete from events" UTF8String],
 NULL, NULL, &sqliteError)==SQLITE_OK){
 NSLog(@"Events Purged");
 }
}
```

- 19.** We then proceed to extract the individual event attributes such as ID, Title, Description, Start, and End time from the JSON data array by using the following code:

```
idText = [[[items objectAtIndex:0]objectForKey:@"event"]
objectAtIndex:i] objectForKey:@"id"];
titleText = [[[items objectAtIndex:0]objectForKey:@"event"]
objectAtIndex:i] objectForKey:@"title"];

NSString *title = [NSString alloc] initWithFormat:titleText];
title = [title stringByReplacingOccurrencesOfString:@"\" \"
 withString:@"' '"];
```

- 20.** The title for events sometimes has double quotes in it ("), so we replace that with the escape double quotes, or else our SQL Insert statements will break.

- 21.** Similarly, we retrieve all the other attributes on an event (subject to our database design) and prepare an SQL Insert statement, shown as follows:

```
// Insert 10 nearby events in SQLite table events

if(sqlite3_open([sqliteFileName UTF8String],
&database)==SQLITE_OK)
{
insertStatement = [NSString alloc] initWithFormat:
@"INSERT OR REPLACE INTO
'%'('%', '%%', '%%', '%%', '%%', '%%', '%%', '%%', '%%', '%%', '%%', '%%')
VALUES('%%', \"%%\", \"%%\", '%%', '%%', '%%', '%%', '%%', '%%', '%%', '%%', '%%',
'%')", eventsTableName, @"id", @"title", @"description",
@"start_time", @"end_time", @"venue_id",
@"latitude", @"longitude",
@"price", @"images", @"category_id", idText, title, description,
startTimeText, endTimeText, venueIdText, latitudeText,
longitudeText, priceText, imagesText, categoryText];

if(sqlite3_exec(database, [insertStatement UTF8String], NULL,
NULL, &sqliteError)==SQLITE_OK)
{
NSLog(@"Events Inserted %@", title);
}
else
{
NSLog(@"Error :%@", insertStatement);
}
}
// End of Insert 10 nearby events
```

**22.** Once all the events are inserted, we call the `readEventsFromLocal` method to read the inserted values from the database and pass them onto the `UITableView` for display.

**23.** The `readEventsFromLocal` method is defined as follows:

```
-(void)readEventsFromLocal
{
 if(sqlite3_open([sqliteFileName UTF8String],
 &database)==SQLITE_OK)
 {
 selectStatement = [[NSString alloc] initWithFormat:@"SELECT
* from %@ order by id desc",eventsTableName];

 sqlite3_stmt *sqlStatement;

 if(sqlite3_prepare_v2(database, [selectStatement UTF8String],
 -1,
 &sqlStatement, NULL)==SQLITE_OK)
 {
 while(sqlite3_step(sqlStatement)==SQLITE_ROW)
 {
 NSString *idDataText = [NSString stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 0)];

 NSString *titleDataText = [NSString stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 1)];

 NSString *descriptionDataText = [NSString
 stringWithUTF8String:(char *)sqlite3_column_text
 (sqlStatement, 2)];

 NSString *startTimeDataText=[NSString
 stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 3)];

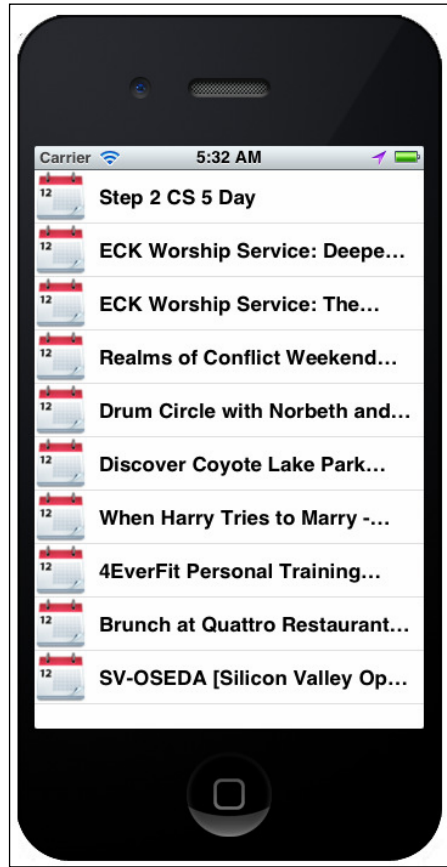
 NSString *endTimeDataText [NSString stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 4)];

 if(![events containsObject:titleDataText]) //Check for
 Duplicates
 {
 [events addObject:titleDataText];
 }
 }
 } //end of while
 } // End of SQLite prepared statement
} // End of if of sqlite3 open

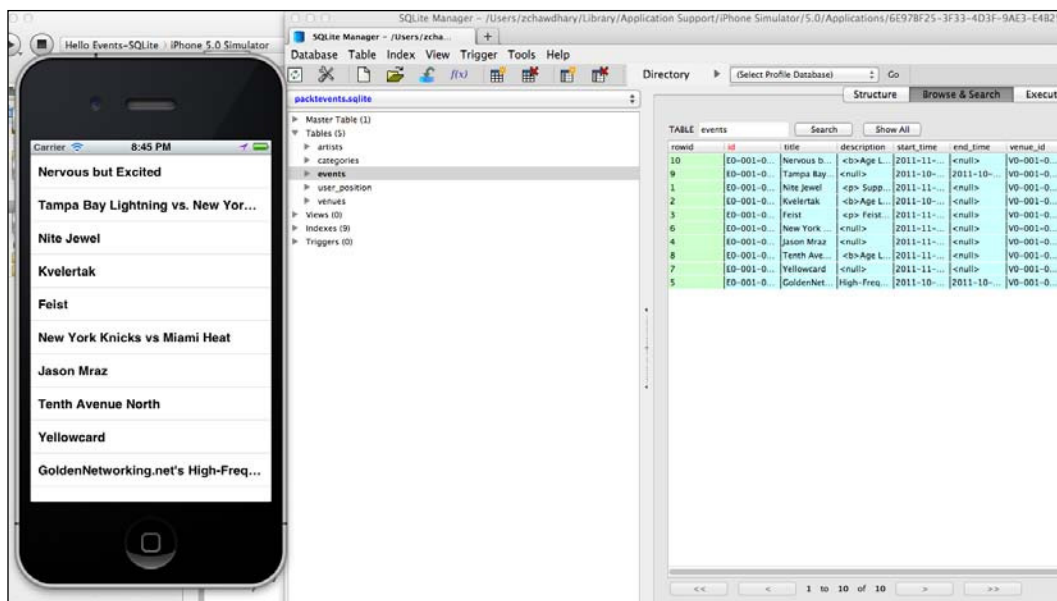
[myTableView reloadData]; // reload the UITableView // display
}
```



- 24.** You can find the complete code on the book's website, titled *Hello Events-SQLite*.
- 25.** Running the application produces the following result:



- 26.** It is a good idea to check the database values and compare them with the actual results; you can do so with the SQLite Manager add-on in Firefox.
- 27.** The SQLite database is also created on the simulator, in your `/Users/{USER_NAME}/Library/Application Support/iPhone Simulator/5.0/Applications/{Application ID}/Documents/packtevents.sqlite`, as seen in the following screenshot:



## What just happened?

The new JSON API in iOS 5 SDK, `NSJSONSerialization`, does the heavy work here. We use it to convert JSON into native Foundation objects; namely Dictionary and then eventually into arrays. We parse the array and retrieve the required attributes for an event by using `objectAtIndex` and `objectForKey` on the array.

```
startTimeText = [[[items objectAtIndex:0] objectForKey:@"event"]
 objectAtIndex:i] objectForKey:@"start_time"];
```

Once the parsing is done, we stored the events in a database and retrieved it to display on a `UITableView`.

## Plotting events on a map

In *Chapter 4*, we looked at the MapKit framework and understood how to add maps and markers to our application. We will now create a new example that uses the Hello Events example we created already in this chapter to also show all the events on the map via annotations for each event. We will do this by creating a new Tabbed Application in Xcode.

## Time for action – plotting events on a map

1. Open Xcode and start a new project by selecting the Tabbed Application template. Name it **Hello Events:Maps**.
2. We use the last example, **Hello Events**, and create the first tab as the `HelloEvents:SQLite` application we saw before. We add the tab icon as a calendar, and on the second tab, we add an image for the map view. These images are free to use, as mentioned earlier:

```
// Hello_EventsFirstViewController.m

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:
(NSBundle *)nibBundleOrNil
{
 self = [super initWithNibName:nibNameOrNil
 bundle:nibBundleOrNil];
 if (self) {
 self.title = NSLocalizedString(@"Events", @"Events");
 self.tabBarItem.image = [UIImage imageNamed:
 @"Mobile-Icons/02_calendar_48.png"];
 }
 return self;
}

// Hello_EventsSecondViewController.m

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:
(NSBundle *)nibBundleOrNil
{
 self = [super initWithNibName:nibNameOrNil
 bundle:nibBundleOrNil];
 if (self) {
 self.title = NSLocalizedString(@"Maps", @"Maps");
 self.tabBarItem.image = [UIImage imageNamed:
 @"Mobile-Icons/04_maps_48.png"];
 }
 return self;
}
```

3. As we already have the events in the database on the device now, our **Maps** tab and controller will just read it from the database and plot it on the maps.
4. The `Hello_EventsSecondViewController.h` and `Hello_EventsSecondViewController.m` files control the **Maps** tab. We reuse the code and add only the required objects and properties here; we need the variables for the `mapView` and the SQLite functionality only. Our `Hello_EventsSecondViewController.h` looks like the following code snippet:
 

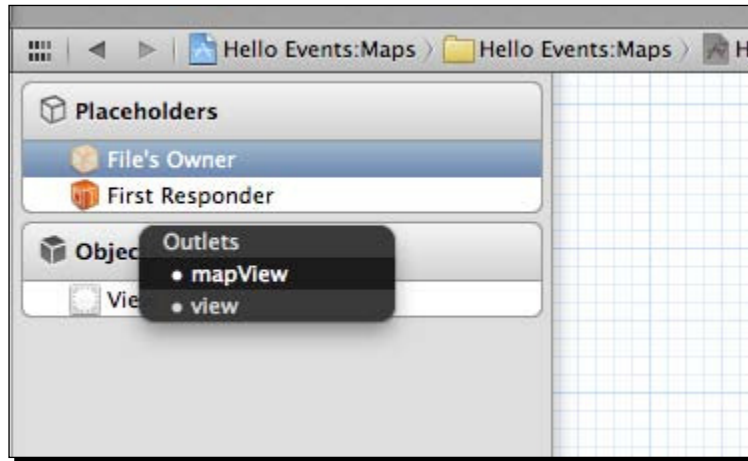
```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>
#import <sqlite3.h>
#import <MapKit/MapKit.h>

@interface Hello_EventsSecondViewController : UIViewController
{
 sqlite3 *database;
 NSString *sqliteFileName;
 MKMapView *map;
 UIView *mapView;
 MKPointAnnotation *annotation;
}

@property (retain, nonatomic) MKMapView *map;
@property (strong, nonatomic) IBOutlet UIView *mapView;
@property (retain, nonatomic) MKPointAnnotation *annotation;

- (NSString *) getDatabaseFullPath;
- (void) readEventsFromLocal;
```
5. The code is pretty straightforward; it is almost a mash up of our *Maps Examples* in *Chapter 4* and the *Hello Events* example.

6. In the `viewDidLoad` method in `Hello_EventsSecondViewController.h`, we create the `Map` object and define the `UIView` for it. We also connect the same from the Interface Builder (press down the `Ctrl` key and drag the mouse from File's Owner to the View object and select the **mapView** outlet). We then call the `readEventsFromLocal` method, as we are assuming that our application has already stored the same from Tab 1 – The **Events** tab.



7. In our `readEventsFromLocal` method, we have made some changes to create the Annotations (Markers) from the Events Data stored in the SQLite database using the latitude/longitude columns in the events table:

```
CLLocationCoordinate2D coord =
{
 .latitude = eventLatitude ,
 .longitude= eventLongitude
};
```

8. Eventually we create the annotations with the preceding `coord` variable:

```
annotation.coordinate = coord;
```

9. The complete code for the `readEventsFromLocal` method is now as follows:

```
-(void)readEventsFromLocal
{

 if(sqlite3_open([sqliteFileName UTF8String], &database)
 ==SQLITE_OK)
 {
 NSString *eventsTableName = @"events";
```

---

```

NSString *selectStatement = [[NSString alloc] initWithFormat:
 @"SELECT * from %@ order by id desc",eventsTableName];

sqlite3_stmt *sqlStatement;
if(sqlite3_prepare_v2(database, [selectStatement UTF8String],
 -1, &sqlStatement, NULL)==SQLITE_OK)
{
 while(sqlite3_step(sqlStatement)==SQLITE_ROW)
 {
 NSString *idDataText = [NSString stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 0)];
 NSString *titleDataText = [NSString stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 1)];

 NSString *descriptionDataText= [NSString
 stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 2)];

 NSString *startTimeDataText= [NSString
 stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 3)];

 NSString *endTimeDataText = [NSString stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 4)];

 double eventLatitude =
 sqlite3_column_double(sqlStatement, 6);
 double eventLongitude =
 sqlite3_column_double(sqlStatement, 7);

 //Annotations Started
 CLLocationCoordinate2D coord = {
 .latitude=eventLatitude,
 .longitude=eventLongitude };

 region.center = coord;
 region.span.latitudeDelta = 0.1;
 region.span.longitudeDelta = 0.1;
 [map setRegion:region animated:TRUE];

 annotation = [[MKPointAnnotation alloc] init];
 annotation.title = titleDataText;
 annotation.coordinate = coord;
 [map addAnnotation:annotation];
 }
}

```

```
 //Annotations Ended

 } // end of while loop
 } // end of if sqlite3 prepared statement
} // end of if of sqlite3 open
}
```

- 10.** Run the app in the iOS 5 Simulator with the location set to San Francisco. You should see the following output:



## What just happened?

We created a simple maps display for showing location-based events retrieved from `eventful.com`. The `MKPointAnnotation` object has been used in a loop to cycle through the events list and to add annotations to the map.

Find the example code for this example on the book's website, in the project titled *Hello Events/Maps*.

## Have a go hero – add more dynamics to the map

Push yourself to the challenge of adding a callout to the Annotation, so when you click on the Annotation, it shows you more information about that event. Head to [http://developer.apple.com/library/ios/documentation/MapKit/Reference/MKAnnotationView\\_Class/Reference/Reference.html#//apple\\_ref/occ/instp/MKAnnotationView/leftCalloutAccessoryView](http://developer.apple.com/library/ios/documentation/MapKit/Reference/MKAnnotationView_Class/Reference/Reference.html#//apple_ref/occ/instp/MKAnnotationView/leftCalloutAccessoryView) for some quick tips!

## Filtering Events display by Event Categories

Now that we have learned to show events on the maps, we move forward to filtering events by categories. Customizations that suit the end user's need make an app more useful. So there may be users who are interested only in concert events, while another group of users would prefer sports events. We resolve this by changing the home screen to displays a list of categories, and then, based on the selection and the user's location, we show nearby events.

Eventful has a rich set of categories for covering all the breadth of events happening around the globe. The following list shows the depth of event categories supported by the Eventful API:

| Category ID        | Category Name            |
|--------------------|--------------------------|
| music              | Concerts & Tour Dates    |
| conference         | Conferences & Tradeshows |
| learning_education | Education                |
| family_fun_kids    | Kids & Family            |
| festivals_parades  | Festivals                |
| movies_film        | Film                     |
| food               | Food & Wine              |
| fundraisers        | Fundraising & charity    |
| art                | Art Galleries & Exhibits |



| Category ID           | Category Name           |
|-----------------------|-------------------------|
| support               | Health & Wellness       |
| books                 | Library & Books         |
| attractions           | Museums & Attractions   |
| community             | Neighborhood            |
| business              | Business & Networking   |
| singles_social        | Nightlife & Singles     |
| schools_alumni        | University & Alumni     |
| clubs_association     | Organizations & Meetups |
| outdoors_recreation   | Outdoors & Recreation   |
| performing_arts       | Performing Arts         |
| animals               | Pets                    |
| politics_activisim    | Politics & Activism     |
| sales                 | Sales & Retail          |
| science               | Science                 |
| religion_spirituality | Religion & Spirituality |
| sports                | Sports                  |
| technology            | Technology              |



**Note:** Most of the API call works by the Category ID. The category Name is for display purpose; avoid it wherever possible. The category ID is preferred as it is all lowercase, clean, and has no special character-based keyword.

## Time for action – filtering Events by categories

We use the last example and extend it to first show the list of categories, and based on user selection, we call the right Eventful API URL. The project is titled *Hello Events-Filtering*.

1. Open `Hello_EventsFirstViewController.h` and declare some more functions as well as modifying the `readEventFulApi` method so that it starts accepting the Category ID as a parameter.

```
- (NSString *) getDatabaseFullPath;
- (NSString *) initializeDatabase;
- (void) readEventFulApi:categoryId;
```

```

- (void) readEventsFromLocal;
- (NSString *) returnCategoryIdForName:categoryName;

- (void) readCategoriesFromApi;
- (void) readCategoriesFromLocal;

```

2. The methods in bold are new additions. The `returnCategoryIdForName:categoryName` method is used to return the category ID, which is provided by the category Name (refer to the Category ID to Category Name mapping table discussed a little earlier).
3. The `readCategoriesFromApi` and `readCategoriesFromLocal` methods are used to parse the category data from the Eventful API, store it in the local SQLite database, and finally read it on demand.
4. In the `Hello_EventsFirstViewController.m` file, we define a new variable of the type `NSMutableArray` that will hold the categories data in an array and use it to display the category list on the `UITableView`:

```
NSMutableArray *categories;
```

5. We also define an `NSString` variable for the categories database `TableName`:

```
NSString *categoryTableName = @"categories";
```

6. In our `viewDidLoad` method, we create the categories array and call the `readCategoriesFromApi` method to start reading the category information from Eventful and store it in our local database.

```

categories = [[NSMutableArray alloc] init];
sqliteFileName = [self getDatabaseFullPath];

[self initializeDatabase];
[self readCategoriesFromApi];

```

7. In the `readCategoriesFromApi` method, we set the `inCategories` flag to `TRUE`; this is used to differentiate the current state of the app - between processing categories information and processing the events information. We switch between the two using the `inCategories` and `inEvents` flags. We then call the Eventful API URL for the Category and continue to process the JSON received:

```

NSString *url = [NSString stringWithFormat:
 @"http://api.evdb.com/json/categories/list?&app_key=%@", appKey];

```

- 8.** Once the API is called and the JSON is received, the `connectionDidFinishLoading` is called. We modify this by checking the `inCategories` and `inEvents` flag to perform processing accordingly. As we have the `inCategories` flag set to `TRUE` by default, we first process the categories information and store it in the database as follows:

```
// Start of Categories Parsing
if(inCategories)
{
 items = [NSArray arrayWithObject:
 [dictionary objectForKey:@"category"]];
 NSUInteger count = [[items objectAtIndex:0] count];

 if(count >= 5)
 {
 if(sqlite3_exec(database, [@"Delete from categories"
 UTF8String], NULL, NULL, &sqliteError)==SQLITE_OK)
 {
 NSLog(@"Categories Purged");
 categories = [[NSMutableArray alloc] init];
 }
 }

 for(NSInteger i=0;i<count-1;i++)
 {
 categoryIdText= [[NSMutableString alloc] init];
 categoryIdName= [[NSMutableString alloc] init];

 categoryIdText=[[[items objectAtIndex:0]
 objectAtIndex:i]objectForKey:@"id"];
 categoryIdName=[[[items objectAtIndex:0]
 objectAtIndex:i]objectForKey:@"name"];

 if(sqlite3_open([sqliteFileName UTF8String],
 &database)==SQLITE_OK)
 {
 insertStatement=[[NSString alloc] initWithFormat:
 @"INSERT OR REPLACE INTO '%@' ('%@','%@')
 VALUES (\\"%@\\",\\"%@\\")",categoryTableName,@"id",
 @"name",categoryIdText,categoryIdName];

 if(sqlite3_exec(database, [insertStatement UTF8String],
 NULL, NULL, &sqliteError)==SQLITE_OK)
 {
 NSLog(@"Categories Inserted");
 }
 else
 {

```

```

 NSLog(@"Error :%@",insertStatement);
 }
} // end of for loop
} // End of Categories Parsing

```

9. Once the categories have been parsed and inserted in the database, we call the `readCategoriesFromLocal` method, which reads these newly inserted values and stores the category in the categories array, defined earlier in the code, and this category array is then passed onto the `UITableView` (via the `cellForRowAtIndexPath` and the `inCategories` flag).
10. The `readCategoriesFromLocal` method is straightforward. By now, you should be comfortable in understanding the usual SQLite table read process:

```

- (void) readCategoriesFromLocal
{
 if (sqlite3_open([sqliteFileName UTF8String],
 &database) == SQLITE_OK)
 {
 selectStatement = [[NSString alloc] initWithFormat:
 @"SELECT * FROM %@", categoryTableName];

 sqlite3_stmt *sqlStatement;

 if (sqlite3_prepare_v2(database, [selectStatement UTF8String],
 -1, &sqlStatement, NULL) == SQLITE_OK)
 {
 while (sqlite3_step(sqlStatement) == SQLITE_ROW)
 {
 NSString *categoryIdDataText = [NSString stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 0)];

 NSString *categoryNameDataText = [NSString
 stringWithUTF8String:
 (char *)sqlite3_column_text(sqlStatement, 1)];

 categoryNameDataText = [categoryNameDataText
 stringByReplacingOccurrencesOfString:@"&"
 withString:@"&"];

 if (![categories containsObject:categoryNameDataText])
 {
 [categories addObject:categoryNameDataText];
 }
 }
 }
 }
}

```

```
 } // end of if of sqlite3 open

 [myTableView reloadData];

}
```

- 11.** In the `UITableView`'s `cellForRowAtIndexPath` method, we assign different images for the cells based on the information displayed. If events are being displayed, we show the same calendar icon as before. However, if categories are being displayed, we show a new icon this time. This is again based on the `inCategories` and `inEvents` flags; this is shown as follows:

```
if(inEvents)
{
 UIImage *newImage = [UIImage imageNamed:
 @"Mobile-Icons/02_calendar_48.png"];
 cell.imageView.image = newImage;
 cellContent = [events objectAtIndex:indexPath.row];
}

if(inCategories==TRUE)
{
 UIImage *newImage = [UIImage imageNamed:
 @"Mobile-Icons/08_settings_48.png"];
 cell.imageView.image = newImage;
 cellContent = [categories objectAtIndex:indexPath.row];
}
```

- 12.** To give the Categories List some interactivity, we give it an accessory. This will show more information when we click on the > sign. This much information is nothing but our events being called based on the Category selected. So we define the Cell's accessory type in the `cellForRowAtIndexPath` method of the `UITableView` delegate as follows:

```
cell.accessoryType=UITableViewCellAccessoryDetailDisclosureButton;
```

- 13.** Now when we select any category from the `UITableView`, the `didSelectRowAtIndexPath` method is called. So now we add the events API call here, specifying the category selected (note that we will send the category ID, so we get the same from the `returnCategoryIdForName:categoryName` method), as follows:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
 NSString *temp;
 if(inCategories)
 {
```

```

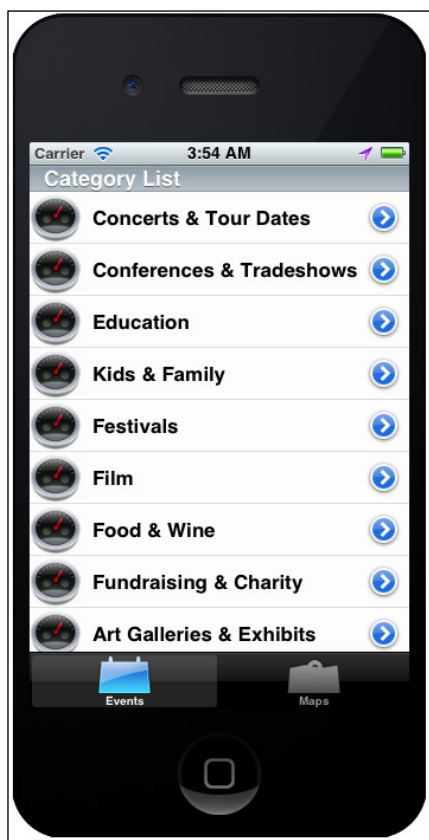
 temp = [tableView
 cellForRowAtIndexPath:indexPath].textLabel.text;
 NSString *apiCatId = [self returnCategoryIdForName:temp];

 [self readEventFulApi:apiCatId];
 }
 // now get a list of events for the particular category selected
 // via the uitableview
 [tableView deselectRowAtIndexPath:indexPath animated:YES];
}

```

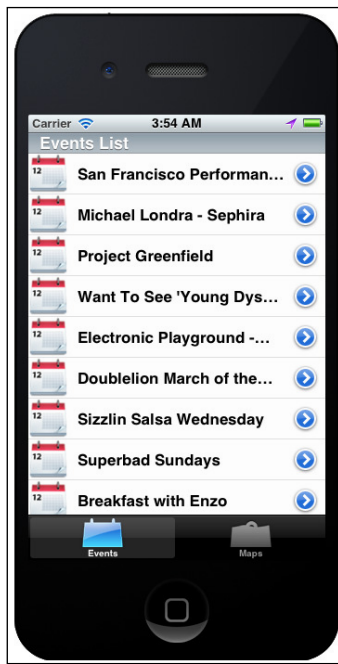
**14.** Once we get the selected category, Name, we convert it into an appropriate category ID using the `returnCategoryIdForName` method. We then call the `readEventFulApi: categoryId` method to start fetching events from the Eventful API based on the category ID supplied and then the JSON is parsed; events stored in the database are then read and displayed on the UITableView as before.

**15.** Run the application. You should see the start screen as follows:



- 16.** Now when you click on any of these categories, you will see that `tableView` is filled with events related to that category. We have also changed the `tableView`'s header by using the flags again:

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section{
 if(inCategories)
 {
 return @"Category List";
 }
 return @"Events List";
}
```



- 17.** We can also use the `tableView:accessoryButtonTappedForRowWithIndexPath` method of the `UITableView` class to define the actions for the accessory added in our `tableView` ( the blue arrow > ).

### ***What just happened?***

Using the `UITableView` smartly, we are able to switch between Categories display and Events display. We used the concept of flags to monitor the state of the application and perform actions accordingly.

We learnt how to use the Eventful Search API by passing category data.

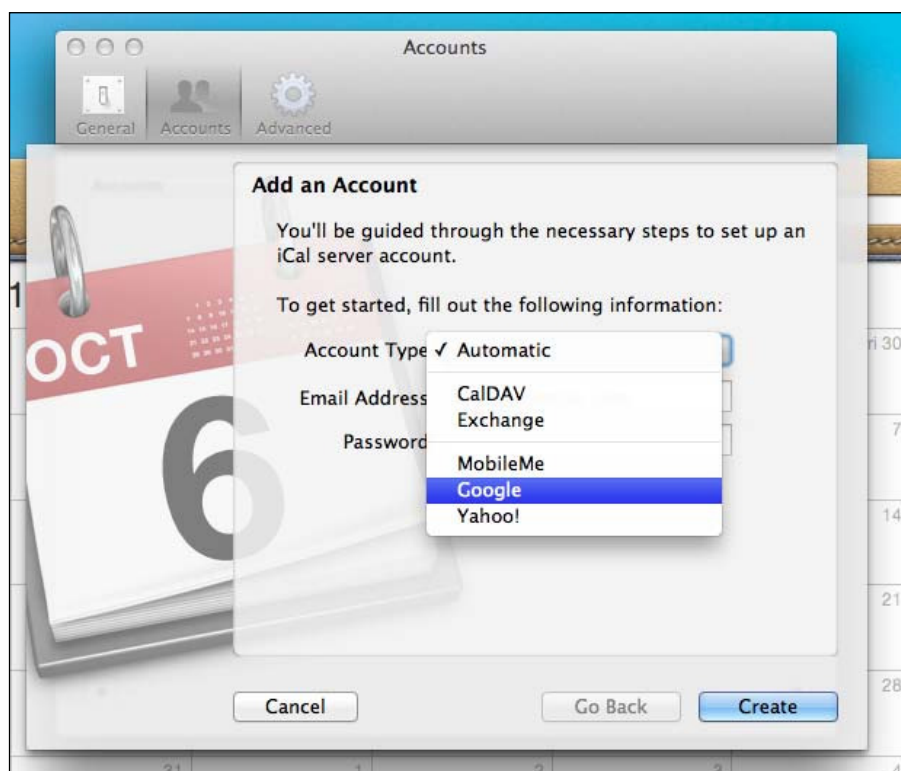
## Using the Event Kit framework to add events to your iPhone calendar

The Event Kit framework allows us to access the user's calendar and events information. Event Kit has two components. They are as follows:

- ◆ The Event Kit framework
- ◆ The Event Kit UI framework

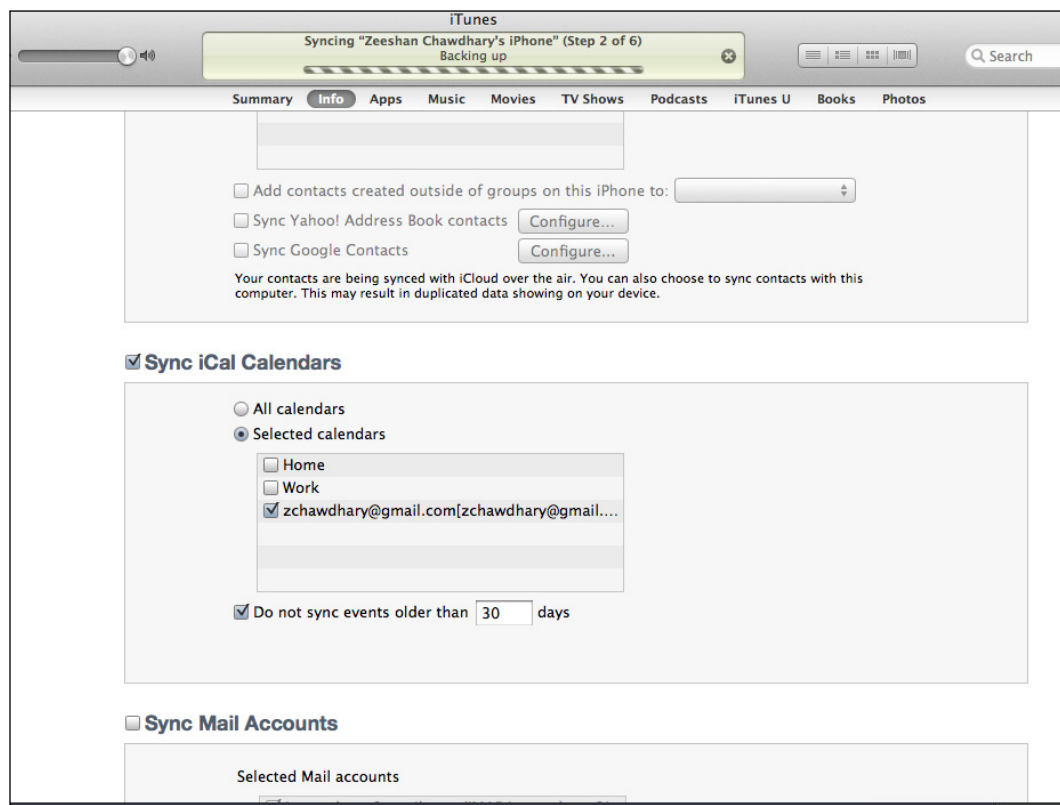
While the **Event Kit framework** allows us to programmatically access, create, delete, and update the events on the iOS Device, the **Event Kit UI framework** provides easy-to-use UI view controller classes that directly work with the iOS Calendar app, so kind of like a GUI-driven calendar manipulation.

We will look at the Event Kit UI framework and understand how to add events to our iOS Device. It is a good idea now to add a Calendar account on your iPhone; you can do it from your phone or from your Mac. On your Mac, open the iCal app and add your Google account to it from the **iCal | Preferences** menu option:





You can then sync this new account from within iTunes by selecting the **Info** tab in iTunes and navigating to the Sync iCal Calendars.



Once you are done syncing, the new calendar should be visible under the **Calendars** section in the iCal app. Similarly, you can have a number of Calendars.

The **EKEventStore** is the main class in the Event Kit Framework that contains references to the various Calendars available on the device, via the **EKCalendar** class. Each EKCalendar can then have events (**EKEvent** class object) attached to it.

## Time for action – adding events to your iPhone calendar

We take the Hello Events (SQLite example) and extend it to add Events onto our iOS calendar using EventKit and EventKit UI framework.

1. Add **EventKit** and **EventKit UI** framework to your project.
2. In your `Hello_EventsViewController.h`, define the variables for the EventStore, Event Calendar, and Event object as follows:
 

```
EKEventStore *eventStore;
EKCalendar *eventCalendar;
EKEvent *event;
NSMutableArray *eventList;
EKEventEditViewController *eventController;
```
3. We also define an array to hold our events data (of the type `EKEvent`) in an array, so we can parse through the events array and add it to the calendar on user input.
4. The `EKEventEditViewController` object allows us to use the core iOS Calendar UI and actions to create a new event or edit an existing event.
5. In our `Hello_EventsViewController.m` file, we define an extra global variable to hold the current Index of the event. This is needed in `EKEventEditViewController` to point to the current event being added.
 

```
NSInteger currentIndex;
```
6. In the `viewDidLoad` method, we initialize the Event Kit variables as follows:
 

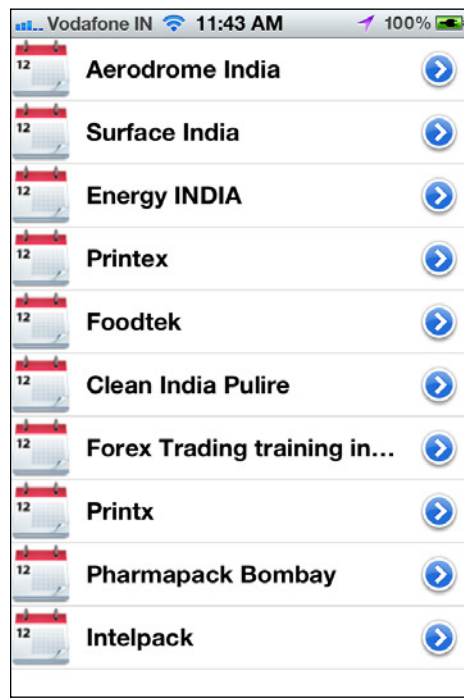
```
eventStore = [[EKEventStore alloc] init];
eventCalendar = [eventStore defaultCalendarForNewEvents];
eventList = [[NSMutableArray alloc] init];
```
7. We modify the `UITableView`'s `cellForRowAtIndexPath` method to add a new accessory button to it (similar to the earlier example), so that we can perform an action on the click of that button; in this case, we will fire the Calendar's add event UI via `EKEventEditViewController` as a modal pop-up:

```
cell.accessoryType=UITableViewCell
AccessoryDetailDisclosureButton;
```

- 8.** The callback for this accessory is where all the magic happens. As the user clicks a cell, we capture the row number, find the event at that row, and eventually pass it on to the EventKit UI via the `EKEventEditViewController` as follows:

```
-(void)tableView:tableView accessoryButtonTappedForRowWithIndexPath: (NSIndexPath *)indexPath
{
 currentIndex = indexPath.row;
 eventController = [[EKEventEditViewController alloc]
 init];
 eventController.eventStore= eventStore;
 eventController.event= [eventList objectAtIndex:currentIndex];
 eventController.editViewDelegate = self;
 [self presentViewController:eventController animated:YES];
}
```

- 9.** Running the app (also bundled on the book's website, titled *Hello Events-EventKit*) produces the following results:



- 10.** On clicking the blue arrow `>`, the application shows the same **Calendar | Add Event** pop-up as you would get from the iCal app.



- 11.** After clicking on the **Done** button at the top-right-hand corner, you should see the event added to your calendar, as follows:



- 12.** You can also add Alerts, Notes, or Invitees to the event.

## ***What just happened?***

Using the EventKit and EventKit UI Framework, we quickly added a calendar and events to our application. We used the default `EKEventEditViewController` controller of the Event Kit UI framework, which provides the default Add Events UI and functionality to our app.

## **Using the Twitter framework**

We are finally at the stage of looking at the most exciting new feature in iOS 5 – Twitter Integration in the iOS framework and how we can easily use the same in our app.

The Twitter framework in iOS 5 is pretty small and concise. It has just two main classes, and they are as follows:

```
TWRequest.
TWTweetComposeViewController.
```

The `TWRequest` is synonymous with the Twitter HTTP API, where in you can make GET, POST, and DELETE API calls. These are operations that you can perform on behalf of the user. A Twitter request is made up of the API URL (identifying the actual action to perform), parameters, and the HTTP method (GET/POST). The `initWithURL:parameters:requestMethod:` method of the Twitter framework in iOS 5 handles the Twitter request.

For our application, we are concerned with the `TWTweetComposeViewController` class, as this class provides an easy to use Modal view controller object that makes Tweeting from within the app a breeze.

## **Time for action – adding Twitter capabilities to your iPhone app**

We take the `Hello Events - SQLite` example and extend it to add tweets for each Event from within our application by using the iOS 5 Twitter framework; `TWTweetComposeViewController` to be specific.

1. Add the Twitter Framework to your project. Next, open the `Hello_EventsViewController.h` file and import the Twitter Library in your code by using the following:
2. In our `UITableView`'s `cellForRowAtIndexPath` method, add the `accessoryType` button, similar to what we did in the last example:

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

cell.accessoryType =
UITableViewCellAccessoryDetailDisclosureButton;
```

3. In the **Accessory** button tap method, we call the Twitter modal box and pass the event title as the tweet content. The `TWTweetComposeViewControllerResult` is the result object returned from the Twitter Modal Box, which returns either of the two values:

- ❑ `TWTweetComposeViewControllerResultCancelled` - If the user canceled the Twitter Modal Box
- ❑ `TWTweetComposeViewControllerResultDone` - If the user successfully continued using the Twitter Modal Box to send out a tweet

```

- (void) tableView:tableView
 accessoryButtonTappedForRowWithIndexPath:
 (NSIndexPath *) indexPath
{
 currentIndex = indexPath.row;
 NSString *eventTitle = [events objectAtIndex:currentIndex];

 if ([TWTweetComposeViewController canSendTweet])
 {
 TWTweetComposeViewController *tweetViewController =
 [[TWTweetComposeViewController alloc] init];
 [tweetViewController setInitialText:[NSString
 alloc] initWithFormat:@"I am attending this event - %@
 #eventful",eventTitle]];

 [tweetViewController setCompletionHandler:^(
 TWTweetComposeViewControllerResult result)
 {
 NSString *tweetOutput;

 switch (result) {
 case TWTweetComposeViewControllerResultCancelled:
 tweetOutput = @"The user cancelled the tweet. ";
 break;
 case TWTweetComposeViewControllerResultDone:
 tweetOutput = @"You sent a tweet successfully";
 break;
 default:
 break;
 }

 [self dismissModalViewControllerAnimated:YES];
 }];
 [self presentModalViewController:tweetViewController
 animated:YES];
 } // end of if canSendTweet
}

```

4. Before we send the tweet, we need to check if the device can send tweets. This is checked against the Twitter account setup within the iOS 5 device. The `canSendTweet` method is used to check if the user has set up his/her account or not:  

```
[TWTweetComposeViewController canSendTweet])
```
5. We retrieve the current cell and event title from the events array used for the `UITableView` rendering, as follows:  

```
currentIndex = indexPath.row;
NSString *eventTitle = [events objectAtIndex:currentIndex];
```
6. We pass this event title information to the Twitter object as the initial text to be tweeted, adding the hashtag `#eventful`.
7. The Twitter Modal Box is presented via the `[self presentViewController r:tweetViewController animated:YES]` code. The `setCompletionHandler` handles the result of the Tweet operation. In our example, we keep it simple and complete the action by dismissing the Twitter `ModalViewController` after the action is performed.
8. You can find the code on the book's website in a project titled *Hello Events- Twitter*.
9. On running the example, you should get the following screen:



## ***What just happened?***

Using the Twitter Framework on iOS 5 is a breeze. It is easy to set up and use in our apps. We used the `TWTweetComposeViewController` to compose a Modal Tweet Box in our app and send tweets for the event, tagged with a hashtag, and with the initial text as the title of the event.

With the Twitter framework, not only can you send tweets, but you can also geotag them, add images, and add URLs (Twitter framework uploads the images and shortens the URLs on its own). Moreover, multiple Twitter accounts are also supported!

## **Bonus: using the Layar Player API in your app: Augmented Reality**

**Layar** (<http://www.layar.com>) is a popular Augmented Reality EcoSystem for iOS, Android, Symbian, and BlackBerry platforms. We say ecosystem, because not only does it provide a standalone **Augmented Reality (AR)** app for the popular mobile platforms, but it also supports embedding AR into a general iPhone or Android app, through its various APIs and SDKs. For iOS devices, Layar has the **Layar Player**, available through <http://www.layar.com/player/> that provides developers with pre-built libraries for iOS.

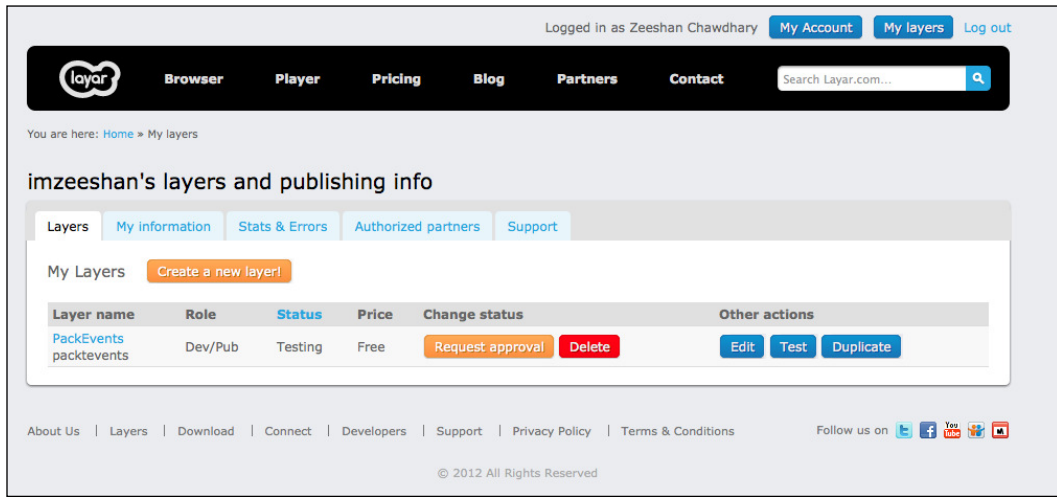
Now any iOS app can be made AR-Aware. Developers have to sign up (at <http://www.layar.com/development/>) and create a **layer** (a set of standard geo-content representation format – readable by the layar player) for their content. The Layar Player then embeds this layer into your iOS app. We move onto a quick example of integrating the Layar player into our *Hello Events* example.

### **Time for action – adding Augmented Reality to your iPhone app**

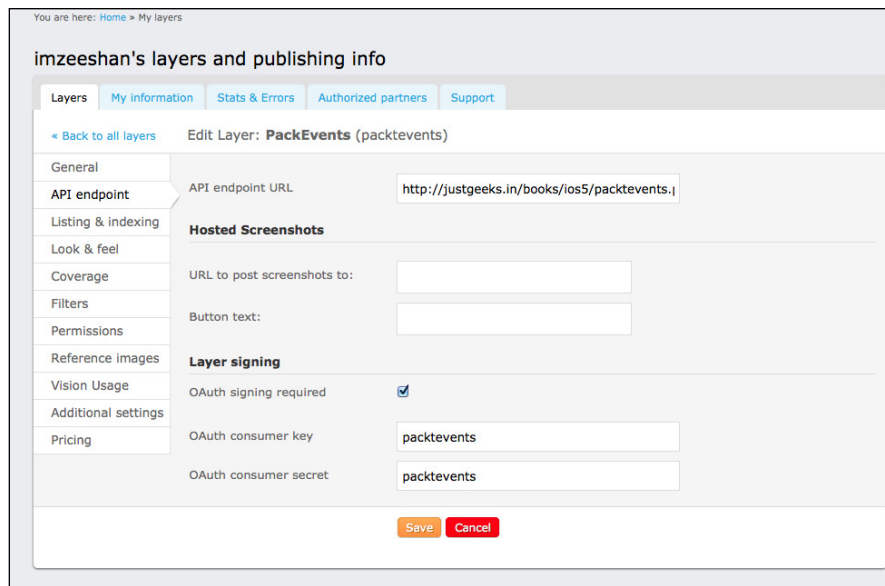
1. Download the Layar Player SDK from <http://www.layar.com/player/> and set up your existing app by importing the required libraries into your Xcode project. For this example, let's name it **Hello Events-Augmented Reality**. Follow the documentation available with the SDK to configure your project; read more about this at <http://layar.pbworks.com/w/page/35051901/Layar-Player-on-iPhone>. The sample code will be available on the book's site, so you can use the project template to play around with.



2. Build a layer at <http://www.layar.com/publishing/> by signing up and creating a new Layer. In our case, we name this Layer **PacktEvents**.



3. Configure the layer's setting. The following are of prime importance:
  - ❑ API endpoint URL
  - ❑ OAuth Consumer key
  - ❑ OAuth Consumer secret



4. We point the API endpoint URL to the PHP script on our webserver that powers our layer. The PHP script hits the `Eventful.com` API, based on the device's location and converts the response into a Layaar-recognized JSON format. The format for which can be found at <http://layar.pbworks.com/w/page/25427491/Tutorials> on creating a layer
5. Our PHP script looks like the following code snippet:

```
<?php

$layerName = $_REQUEST['layerName'];
$lat = $_REQUEST['lat'];
$lon = $_REQUEST['lon'];
$radius = $_REQUEST['radius'];

$eventfulUrl = "http://api.eventful.com/rest/events/
 search?location=$lat,$lon&app
 _key=xxxxxxxxxxxx&within=10";

$file = file_get_contents($eventfulUrl);
$eventsXML = simplexml_load_file($eventfulUrl);
$poi = array();
$hotspots = array();
$i = 0;

foreach ($eventsXML->events->event as $event)
{
 $poi['id'] = trim($event['id']);
 $poi['text']['title'] = trim($event->title);
 $poi['anchor']['geolocation']['lat'] =
 changetoFloat($event->latitude);
 $poi['anchor']['geolocation']['lon'] =
 changetoFloat($event->longitude);
 $poi['text']['description'] =
 htmlspecialchars($event->description);
 $hotspots[$i] = $poi;
 $i++;
}

$response = array();
$response['layer'] = 'packtevents';
$response['hotspots'] = $hotspots;
```

```
if (!$response['hotspots']) {
 $response['errorCode'] = 20;
 $response['errorString'] = 'No POI found. Please adjust the
 range.';
}
else {
 $response['errorCode'] = 0;
 $response['errorString'] = 'ok';
}

$jsonresponse = json_encode($response);
header('Content-type: application/json; charset=utf-8');
echo $jsonresponse;

function changetoFloat($string) {
 if (strlen(trim($string)) != 0)
 return (float)$string;
 return NULL;
}

?>
```

6. Where xxxxxxxxxxxx is our API Key from Eventful.com. The PHP script is included in the code download for this example.
7. Our main screen for this example will be the regular UITableView page with an accessory button, as seen in previous examples. On the accessory button click, we will load the Laya Player and initialize our layer we created in step 2. Note that the layer name, consumer key, and secret values must match the ones we used while creating the layer.

```
-(void)tableView:tableView accessoryButtonTappedForRowWithIndexPath:
(NSIndexPath *)indexPath
{
 currentIndex = indexPath.row;
 NSString *eventTitle = [events objectAtIndex:currentIndex];

 NSString *layerName = @"packtevents";
 NSString *consumerKey = @"packtevents";
 NSString *consumerSecret = @"packtevents";
 NSArray *oauthKeys = [NSArray
 arrayWithObjects:LPConsumerKeyParameterKey,
 LPConsumerSecretParameterKey, nil];
 NSArray *oauthValues = [NSArray arrayWithObjects:consumerKey,
 consumerSecret, nil];
```

```

NSDictionary *oauthParameters = [NSDictionary
 dictionaryWithObjects:oauthValues forKeys:oauthKeys];

NSArray *layerKeys = [NSArray arrayWithObject:@"radius"];
NSArray *layerValues = [NSArray arrayWithObject:@"50000"];
NSDictionary *layerFilters = [NSDictionary
 dictionaryWithObjects:layerValues forKeys:layerKeys];

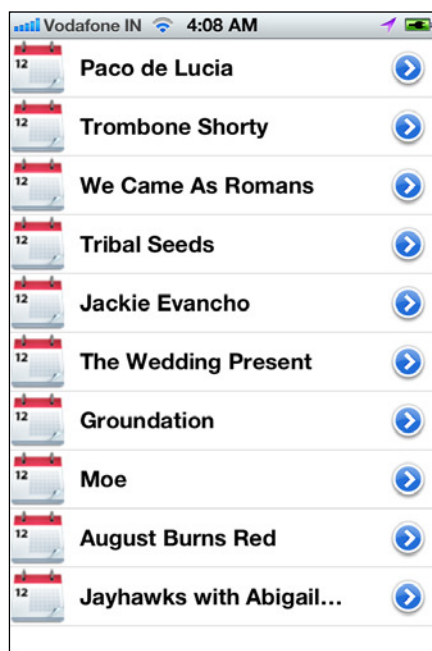
LPAugmentedRealityViewController *augmentedRealityViewController
 = [[LPAugmentedRealityViewController alloc] init];

augmentedRealityViewController.delegate = self;

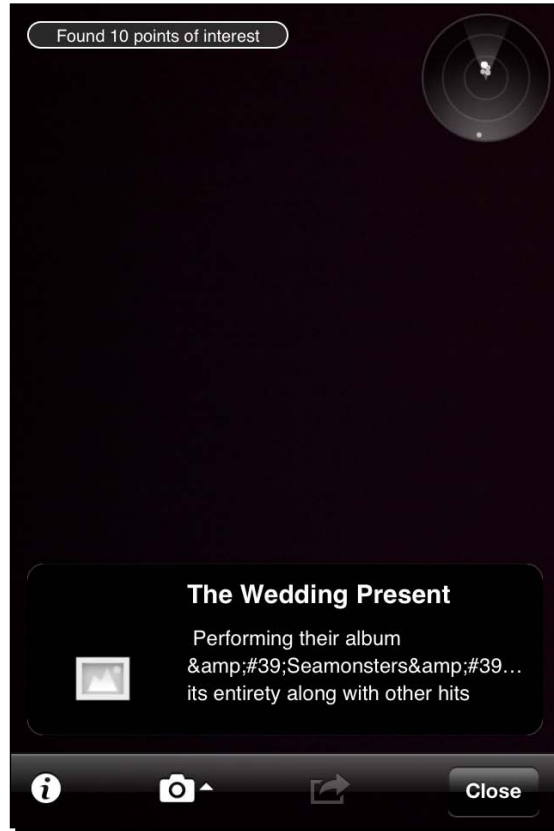
[self presentViewController:augmentedRealityViewController
 animated:YES];
[augmentedRealityViewController loadLayerWithName:layerName
 oauthParameters:oauthParameters layerFilters:layerFilters
 options:LPMMapViewDisabled | LPListViewDisabled];
}

```

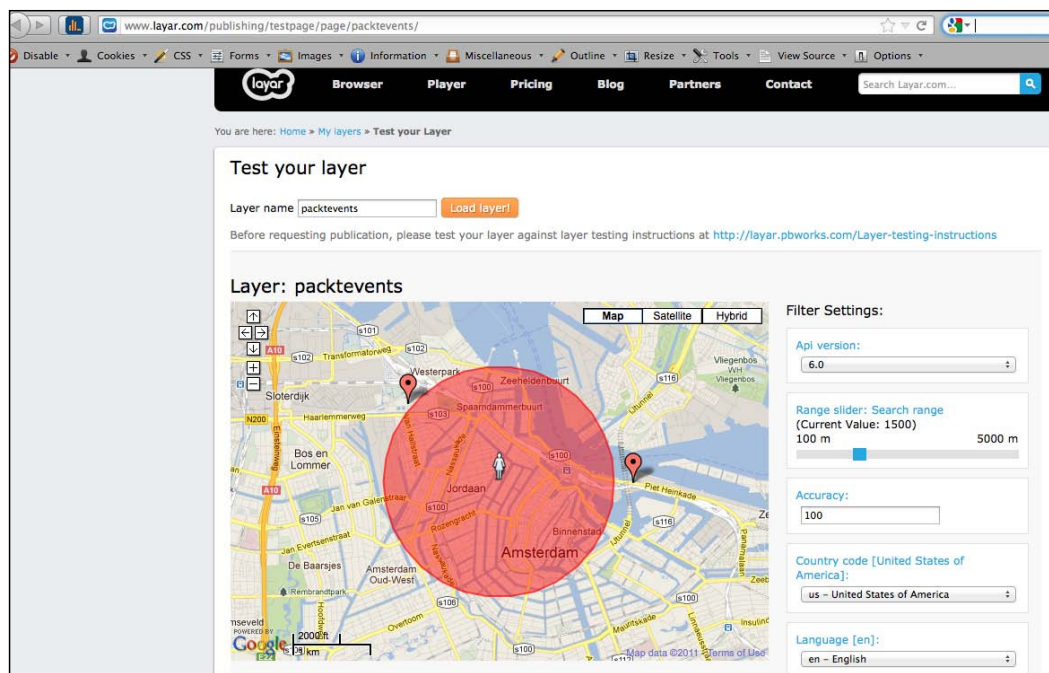
8. The rest of the code is the same as any of the examples we saw before. Run the app on your iPhone. Trying to run the example on the iOS simulator will fail, as we are using the Camera, OpenGL, and related classes here (within the Layar Player libraries). You should get the following screenshot:



9. Clicking on the blue arrow should load the Augmented Reality view, as shown in the following screenshot:



10. You can also test your layers online, via the Layar Publishing Portal.



## What just happened?

By Using the Layar Player SDK for iOS, we saw how easy it was to integrate the Augmented Reality feature in your iOS app. We also saw how to build a layer with a PHP script that parses the Eventful API for events and converts it into a Layar-recognized format. Layar also offers Layar Vision, which allows for Image Recognition capabilities. Another cool feature is the ability to load 3D Models within the Augmented Reality display. By now, you should know where to go for more details!

## PacktEvents: building the app

In *Chapter 5* – we saw how to build the components on the WeatherPackt application. We also combined the components to build the app. In this chapter, we saw all the modules for our PacktEvents app. Now it is time for you, the reader, to build the app as an exercise.

However, do not fear if you still cannot fill in the blocks; the code for PacktEvents will be put up on the book's website along with the full source code and explanation, along with the voice recognition module - courtesy Nuance Mobile SDK.

## Pop quiz – have a blast with events

1. What are the different API components available to add events to the iOS device's calendar?
2. How can you determine whether your iOS 5 capable iOS device can send tweets using the new Twitter Framework in iOS 5?

## Summary

In this chapter, we learned how to consume the Eventful API (courtesy [eventful.com](http://eventful.com)) and store events locally in the best way using SQLite. We also looked at filtering the events by category.

We looked at how the EventKit framework can be used to add events to our calendar. The new Twitter Framework in iOS 5 was also explored.

Specifically, we covered:

- ◆ Eventful API – deep analysis
- ◆ iOS EventKit framework
- ◆ Twitter framework
- ◆ Using the Laya Player (Bonus)
- ◆ Building the PacktEvents app

We learned to build two real life iPhone applications in our last two chapters: a weather app and an events app. Now it is time to learn some advanced iOS concepts such as Core Motion and Notifications. So let's move on to it.

# 7

## Advanced Topics

*Local and push notifications, augmented reality, and Geofencing are some exciting features that lure the users back to your applications. Smart use of these features can lead to extended app usage, thereby generating more revenues for the developer.*

*Smart push techniques combined with Geofencing, reduce the user's efforts to open applications and search for content. Instead, the application pushes notifications and messages to the user, triggering the application to launch from background.*

*Background apps are another way to let your app work in the background, and fire an event in case the user matches certain application logic, thereby bringing the user back to the app.*

In this chapter, we will deal with the advanced topics for iOS 5:

- Using directions with location
- Motion manager
- Running apps in the background, along with background location
- Push notifications and local notifications

So let's get on with it...



## Using directions with location

So far, we have only used location values (latitude and longitude) from the iOS device, which lets us know the user's position. iOS devices can also report the direction of the user's phone (very helpful for navigation apps). The **Core Location** framework supports two methods of determining direction, using magnetometer and the device GPS.

### Direction using heading

As discussed before, the direction in which an iOS device is pointing to is reported by the device magnetometer. This information is known as **heading**. The device GPS hardware reports the direction in which an iOS device is moving. This information is known as **course**.

### Getting your app ready for direction

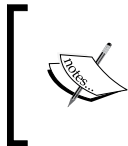
Before we can use direction information in our app, we need to include the `UIRequiredDeviceCapabilities` key in our `Info.plist` file. Depending on what we need to use, heading or course, in our app, the accompanying strings should be added in the `UIRequiredDeviceCapabilitiesKey`.

### Understanding heading using magnetometer

The magnetometer in the iOS device measures nearby magnetic activity. This helps in determining the device orientation. The heading values returned are relative to true north or magnetic north; however, magnetic north should be used for most applications, since the magnetic north keeps shifting each year (due to the movement of the earth's crust). In simple words, true north is a theoretical concept, while magnetic north is more practical-oriented. To receive the heading information in your app, you need to do the following:

- Create a Core Location Manager Object
- Use the `headingAvailable` method to check for heading availability on the device
- Call the `startUpdatingHeading` method.

The heading values are returned as a `CLHeading` class object. The `CLHeading` object contains both, the true and magnetic north values. So, in case you need to switch, you can use the values accordingly.



*HowStuffWorks* has a good article on true north and magnetic north. Read more about it at <http://adventure.howstuffworks.com/outdoor-activities/hiking/compass-or-gps2.htm>.

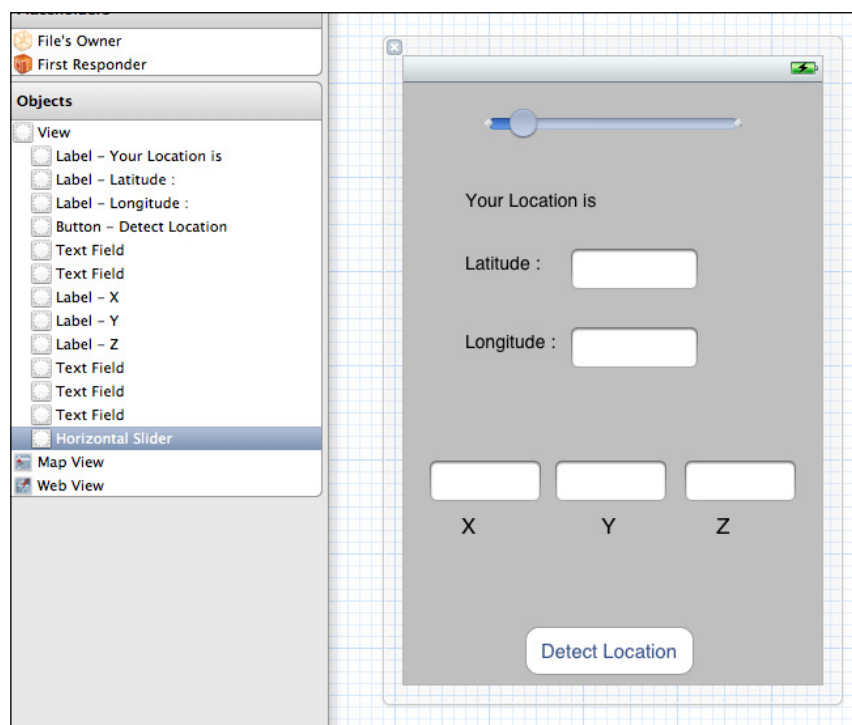
## Time for action – using heading for direction in your app

We revisit the Hello Location - Location Updates example from *Chapter 3, Using Location in your iOS Apps – Core Location*, and add the heading component to our application. Here is what we will achieve: Our application will check for heading information, and as we move the device left/right, we will adjust the on-screen slider to reflect the change in the direction.

1. We begin with defining a UISlider object in our app in the Hello\_LocationViewController.h file, and expose it as a property.  

```
@property (strong, nonatomic) IBOutlet UISlider *XSlider;
```
2. We also declare three additional properties of type UITextField that will hold the X, Y, and Z values retrieved from the magnetometer. Of this, our interest lies in the X value. We will use this to move the slider, as we move right /left.  

```
@property (strong, nonatomic) IBOutlet UITextField *XVALUE;
@property (strong, nonatomic) IBOutlet UITextField *YVALUE;
@property (strong, nonatomic) IBOutlet UITextField *ZVALUE;
```
3. We modify the Hello\_LocationViewController.xib file to look as follows:



4. Do not forget to update your Info.plist file with the following:

```
<key>UIRequiredDeviceCapabilities</key>
<array>
 <string>magnetometer</string>
</array>
```

|                                        |             |                                    |
|----------------------------------------|-------------|------------------------------------|
| ▼ Required background modes            | Array       | (1 item)                           |
| Item 0                                 | String      | App registers for location updates |
| ▼ Required device capabilities         | ⊕ ⊖ Array   | (1 item)                           |
| Item 0                                 | String      | magnetometer                       |
| ▼ Supported interface orientations     | Array       | (0 items)                          |
| ▼ Supported interface orientations (i) | ⬆ ⊕ ⊖ Array | ⬆ (0 items)                        |

5. Now, coming to the main action in `Hello_LocationViewController.m`, we check if heading services are available for the device or not, by using the `headingAvailable` method, and starting the heading updates by using `startUpdatingHeading`.
- ```
// Start heading Updates.

if ([CLLocationManagerheadingAvailable]) {
    locMgr.headingFilter = 5;
    [locMgrstartUpdatingHeading];
}

// End Heading Updates
```
6. Play around with `headingFilter` variable to define how much sensitivity you need in the app. The value passed here signifies how much change in degrees is required to initiate a heading change event.
7. Similar to the `didUpdateToLocation` method, Core Location Manager also exposes the `didUpdateHeading` method, which is fired when a heading change event occurs.
8. We detect the X, Y, and Z values (signifying the horizontal, vertical, and depth deviations) from the new heading information, and pass on the X values to the UISlider, to update its value based on how much deviation has occurred in the X-axis (left or right side deviation).

```
- (void) locationManager: (CLLocationManager *) manager
didUpdateHeading: (CLHeading *) newHeading {

    if (newHeading.headingAccuracy < 0)
    {
```

```

        return;
    }

    XVALUE.text = [NSStringalloc] initWithFormat:@"%f",newHeading.x];
    YVALUE.text = [NSStringalloc] initWithFormat:@"%f",newHeading.y];
    ZVALUE.text=[ NSStringalloc] initWithFormat:@"%f",newHeading.z];

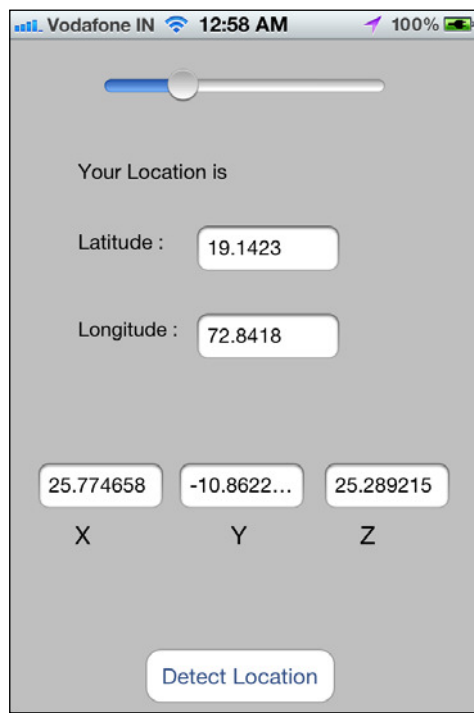
    float value = [XVALUE.text floatValue];
    XSlider.value = value;

    NSLog(@"Slide Value - %f",value);

}

```

- 9.** That's it we are now ready to run the app. Make sure that you have a configured iOS device to test this demo, since it won't be possible to simulate the heading in the iOS simulator easily. You can achieve that with the **Instruments** tool, albeit with some coding. Here is how it works on an actual iPhone:



- 10.** Don't forget to connect the `UITextField`s for X, Y, and Z to the `XVALUE`, `YVALUE`, and `ZVALUE` outlets, respectively.

What just happened?

We used the heading information available from the location manager to detect the heading information, specifically the x-axis deviation, and used the real-time information from the device to control a `Slider` object in our main UI.

Note that even if you do not allow your app to use the location settings, the heading information will still be provided by the Core Location Manager. Find the code for this example on the book's website: project titled *Hello Location-Location Updates with Heading*

We will look at more controls for device handling with Core Motion, which includes parameters , such as gravity and user acceleration.

Direction using course

The device's course information is returned in the `CLLocation` object, which we use to get the user's location. Whenever the location is updated, Core Location also updates the course and the speed values, as and when they become available. Remember that the course information need not necessarily specify the direction of the device; it could also signify the direction in which the device is moving. So, navigational apps rely on the course values.

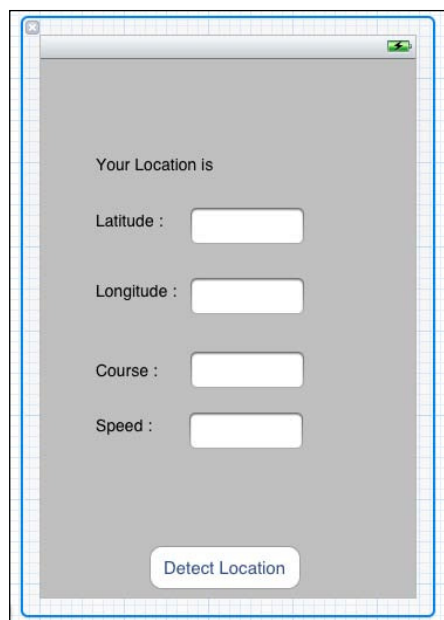
Time for action – using course for direction in your app

We modify the `Hello Location - Location Updates` example, to add course and speed values in the `Hello Location` app.

1. Open the `Hello_LocationViewController.h` file, and add two outlets, one for the speed data and another for the course data.

```
@property (retain, nonatomic) IBOutlet UITextField *courseText;  
@property (retain, nonatomic) IBOutlet UITextField *speedText;
```

2. We just need the speed and course information, so we add two labels and two `UITextFields` in our XIB file as shown in the following screenshot. Also, we connect the same to the outlets created previously.



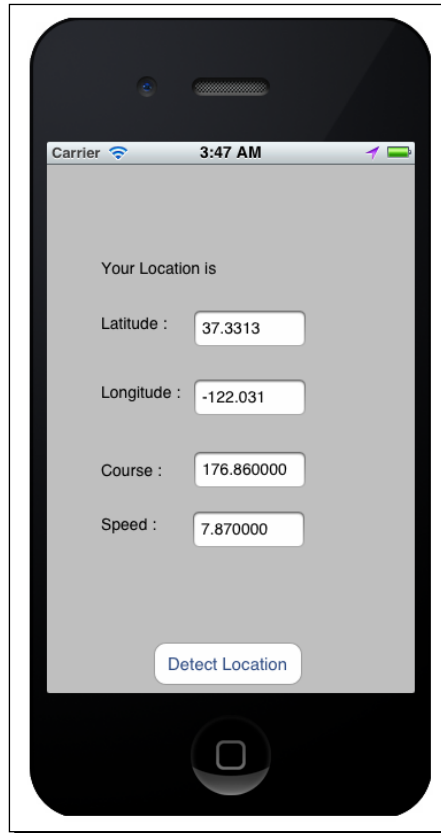
3. In our `ViewDidLoad` method, we change the location manager's accuracy to `kCLLocationAccuracyBestForNavigation`.
4. Next, in the `didUpdateToLocation` and `locationDetect`, we fetch the values of speed and course from the `CLLocation` object.

```
courseText.text = [[NSString alloc]
    initWithFormat:@"%f",newLocation.course];
speedText.text = [[NSString alloc]
    initWithFormat:@"%f",newLocation.speed];
```

The value for course could be as follows:

- ❑ 0 for North
- ❑ 90 degrees for East
- ❑ 180 degrees for South
- ❑ 270 degrees for West

5. Running the app in the simulator produces the following output:



A negative value, usually -1, would imply that the course or speed value is not available for your device.

Find the code example on the book's websites: projected titled Hello Location-Location Updates with Course.

What just happened?

We successfully created a simple app that uses the course and speed values from the core location object to signify the direction of your iPhone, or direction in which your iPhone is travelling, along with its speed.

Note that it is a good idea to include GPS and location-services keys in the `UIRequiredDeviceCapabilities` property in your `Info.plist` file.

Core Motion: Motion Manager

Core Motion primarily handles the accelerometer and gyroscope management for your application through the **Core Motion Manager** framework. Core Motion also runs in its own thread. The benefits of running in its own thread implies that your application does not have to wait for the Core Motion Manager to send information, and can continue running as and when the Core Motion Manager sends values your application thread can consume, providing a better user experience; as we all know – no one likes waiting for information on their devices.

Starting with iOS 5, the Core Motion framework also includes the raw magnetometer data, which was not available to users of iOS version 4.x. It also provides an option to run in the background and access the attitude data. Depending upon which iPhone sensor you are interested in, the Core Motion Manager returns the appropriate Core Motion object.

- `CMAccelerometerData`: For the accelerometer data
- `CMGyroData`: For the gyroscope data
- `CMMagnetometerData`: For the magnetometer data
- `CMDeviceMotion`: For the north referenced attitude data

Let's define each of the sensors to clearly understand their requirements and data they return.

- ◆ **Accelerometer**: The accelerometer is used to measure gravity and user acceleration on the iPhone. The `CMAccelerometerData`, returned by the Core Motion Manager includes the *x*, *y*, and *z* axes acceleration values (in gravitational force uni.). It is the most common sensor that can be found in almost all iOS devices from the first iPhone to the latest iPhone/iPad versions.
- ◆ **Gyroscope**: The gyroscope measures the rate at which the device is rotating with respect to the earth's rotation. So you not only get the *x*, *y*, and *z* values, but also the rotation happening in each of these three axes. Hence, accelerometer + gyroscope give you a six-axes motion control system, which is mostly useful in gaming apps.
- ◆ **Magnetometer**: The magnetometer measures the orientation of the device by using the nearby magnetic fields. As discussed before, these are simple *x*, *y*, and *z* values.
- ◆ **Device Motion**: The `CMDeviceMotion` contains the attitude, gravity, rotation rate, user acceleration, and the magnetic fields information within itself. This is derived by combining the accelerometer and the gyroscope.

Attitude of the device is basically the device's orientation in 3D space. It is a good time to get familiar with concepts, such as roll, yaw, and pitch.

How to use Core Motion

To use Core Motion in your app, you need to do the following:

- Start the Core Motion Manager.
- Define the update interval.
- Stop the Core Motion Manager.

Core Motion supports both `Push` and `Pull` methods of retrieving the sensor data. By specifying the interval, the associated block handle and an operation queue with the start method, the device pushes the sensor data at the specified interval through the block handler.

Periodically asking the Core Motion Manager for sensor data through the `start` method, and accessing the respective motion sensor property, we can pull the sensor data based on our application logic.

We will look at the pull-based approach of retrieving the sensor data from the Core Motion Manager. Let's look at the methods, classes, and properties required for each of the sensors.

	Start method	Object returned	Associated property
Accelerometer	<code>startAccelerometerUpdates</code>	<code>CMAccelerometerData</code>	<code>accelerometerData</code>
Gyroscope	<code>startGyroUpdates</code>	<code>CMGyroData</code>	<code>gyroData</code>
Magnetometer	<code>startMagnetometerUpdates</code>	<code>CMMagnetometerData</code>	<code>magnetometerData</code>
Device Motion	<code>startDeviceMotionUpdates</code>	<code>CMDeviceMotion</code>	<code>deviceMotion</code>

Before we start using these sensors, we need to detect whether these sensors are present on the intended hardware or not. Each sensor has its associated property to determine its availability, and determine whether it is active or not.

The accelerometer has the `accelerometerAvailable` and `accelerometerActive` properties to check for device compatibility. The gyroscope has `gyroAvailable` and `gyroActive` properties. The magnetometer has the `magnetometerAvailable` and `magnetometerActive` properties. Finally, the device motion has the `deviceMotionAvailable` and `deviceMotionActive` properties.

When our application is done processing the sensor data, it is time to call the respective stop methods of the sensors. They are as follows:

- `stopAccelerometerUpdates`
- `stopGyroUpdates`
- `stopMagnetometerUpdates`
- `stopDeviceMotionUpdates`

Time for action – using MotionManager: accelerometer

We have seen how to use the magnetometer data from the Core Location Manager object before. Now we will use the Core Motion Manager to access the device's accelerometer data. For the purpose of this example, we will create a new project named `Hello Motion: Accelerometer`. We will also implement both the `push` and `pull` methods of getting data from the motion manager.

1. Create a new project titled `Hello Motion: Accelerometer`, and add the Core Motion Framework from the **Targets | Hello Motion: Accelerometer | Build Phases** option in Xcode.
2. In our `ViewController.h` file, we import the `<CoreMotion/CoreMotion.h>` header file. We then declare the Motion Manager object, as well as a queue of type `NSOperationQueue`. We also declare three `UITextField` variables that will hold the X, Y, and Z acceleration values. We expose them as outlet properties for the XIB file connection. Another outlet that we create is to connect the `UIButton`.
3. Next, we declare two `IBActions`, one for getting the accelerometer data through the `pull` method (`getAccelerometerData`), and another for stopping the accelerometer updates on the device (`stopAccelerometer`).

Our `ViewController.h` file should now look as follows:

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

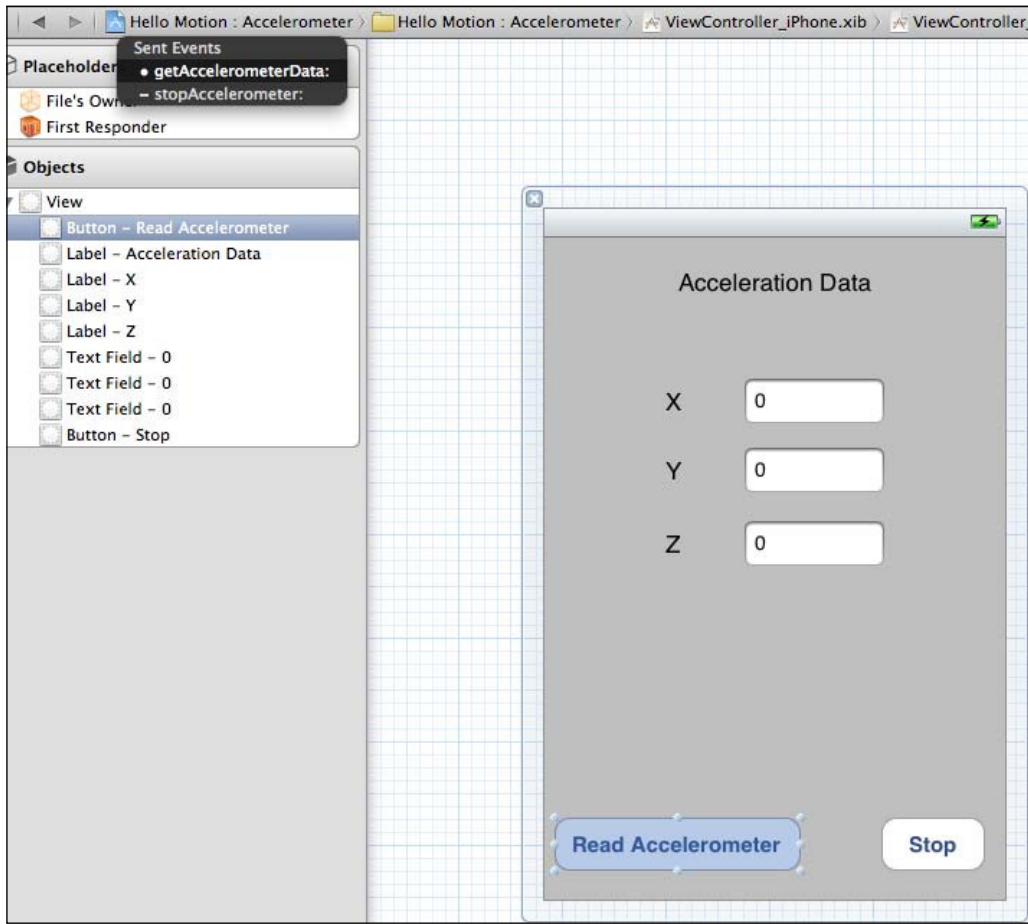
@interface ViewController : UIViewController
{
    CMMotionManager *coreMotionManager;
    NSOperationQueue *coreMotionQueue;
    UITextField *accelerationX;
    UITextField *accelerationY;
    UITextField *accelerationZ;
}

@property (strong, nonatomic) IBOutletUIButton
    *showAccelerometerData;
@property (strong, nonatomic) IBOutletUITextField *accelerationX;
@property (strong, nonatomic) IBOutletUITextField *accelerationY;
@property (strong, nonatomic) IBOutletUITextField *accelerationZ;

- (IBAction)getAccelerometerData:(id) sender;
- (IBAction)stopAccelerometer:(id) sender;

@end
```

4. Open our iOS device XIB file, and add some UI controls to display the labels and text for the x, y, and z acceleration values, as well as two buttons for reading the values on demand (Push), and stopping the accelerometer updates on the device. Your UI should look as follows:



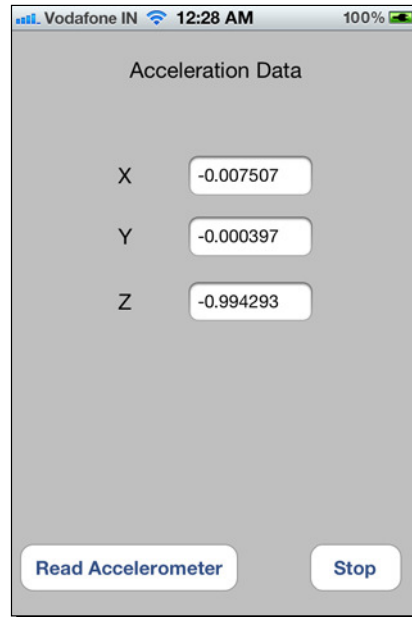
5. Open your `ViewController.m` file. In the `viewDidLoad` method, we initialize the Core Motion object, and set its update interval to 1 second. We also declare the block of code that will execute every time the device triggers an accelerometer update.

6. The `startAccelerometerUpdatesToQueue` method is used to push the accelerometer updates on an operation queue. As the device accelerometer becomes active, we use the acceleration values and assign it to the respective text fields.

```
coreMotionManager = [[CMMotionManager alloc] init];
coreMotionQueue = [[NSOperationQueue alloc] init];
if ([coreMotionManager isAccelerometerAvailable])
{
    coreMotionManager.accelerometerUpdateInterval = 1.0;
    [coreMotionManager startAccelerometerUpdatesToQueue:
    coreMotionQueue withHandler:^(CMAccelerometerData
    *newAccelerometerData, NSError *error)
    {
        if ([coreMotionManager isAccelerometerActive])
        {
            accelerationX.text = [NSString stringWithFormat:@"%f",
            newAccelerometerData.acceleration.x];
            accelerationY.text = [NSString stringWithFormat:@"%f",
            newAccelerometerData.acceleration.y];
            accelerationZ.text = [NSString stringWithFormat:@"%f",
            newAccelerometerData.acceleration.z];
        }
    }];
}
```

7. Now, for the pull-based approach of accessing the accelerometer data, we define the `getAccelerometerData` method, and use the Motion Manager's `startAccelerometerUpdates` method, if the accelerometer is not active. If it is active, we fetch the values from **Core Motion Manager | accelerometer Data Object | acceleration property**.
8. To stop the acceleration updates, we call the `stopAccelerometerUpdates` method under the `stopAccelerometer` function that is triggered when we hit the **Stop** button on the UI.

9. Running the application produces results as shown in the following screenshot. Note that this example will only run on the device. Tilt the device and wait for the device to provide you with the update x, y, and z acceleration values. If you hit the **Stop** button, and hit the **Read Accelerometer** button again, it will only be a pull-based call thereafter, since we started the push-based process in the `viewDidLoad` method, which will not be called until the application is reloaded again.



Find the code for this example on the book's website: project titled Hello Motion: Accelerator.

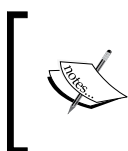
What just happened?

By using the `startAccelerometerUpdatesToQueue` and `startAccelerometerUpdates` methods of updating the device accelerometer values, we used the Core Motion framework to understand how the acceleration values can be obtained through the push and pull mechanism.

We used the `NSOperationQueue` to regulate the execution of the acceleration update process. We also looked at the concept of blocks. Find more information about these at http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/NSOperationQueue_class/Reference/Reference.html and <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html>.

Core Motion conclusion

Core Motion is a vast subject to explore. We looked at the accelerometer and the magnetometer, and so far we had been using the raw data received from these sensors. You can use the raw data, but unless you are a physics student, some of these data and terms will be Greek to you. Thankfully, with iOS 5, Apple has made it simpler for developers to use these sensors using the **Device Motion Data** (also known as **Process Device Motion Data**), which reads raw motion data from the accelerometer and the Gyroscope, and generates refined processed data for attitude, unbiased rotation rate, the direction of gravity, and the user acceleration on the device. In simple words, the calibration and removal of error bias is taken care of by the sensor fusion algorithms in Device Motion.



If you have used `UIAccelerometer` in your iOS app before, with iOS 5 the following changes need to be done in your code. From `UIAccelerometer`, transition to `CMMotionManager`, and from `UIAcceleration`, transition to `CMAccelerometer`.

Background app execution

In our earlier chapters, we looked at enabling background location through the `UIBackgroundModes` key in the `Info.plist` file of our application. Let's look deeper into how background processing for our applications works. Background execution of the code is possible through the implementation of multitasking in iOS.

Why is background code execution needed? Let's take up a scenario. You are a frequent visitor to restaurants or bars, and you are interested in getting the best deals of such venues nearby. Now, if you were using a regular application that showed you nearby places having discount, for say a Pizza meal, you would have to fire the app, hit the search button, and then locate the closest venue. What if the app does it all for you, so that as you move from your home location to say four blocks away, the application automatically calculates your latest position, and based on your preferences, it can show you an alert for nearby deals at restaurants and bars.

The most common use case for background app execution is **Background Location**, so your applications can keep a track of your position, even when running in the background. With iOS 4 and higher versions, applications are no longer terminated when the iPhone's **Home** button is pressed. Instead, the applications are shifted to a background suspended state, where they are either removed from the memory, or based on the application settings, they can continue running in the background.

What apps can run in the background?

Applications that use any of the following, can continue to run in the background:

- The application needs a quantifying amount of time to perform some critical task
- The application supports services that need the application to run in the background
- The application uses local notifications to show user alerts at pre-determined times

As we saw before, we need to specify what background services we need in our app, by specifying the same in the `UIBackgroundModes` key in the `Info.plist` file (or any `.plist` file for that matter). The values for `UIBackgroundModes` can be `audio`, `location`, or `VoIP`.

With the introduction of iOS 5, Core Motion is now added as a supported background mode.

Background location

iOS supports the following background location tasks:

- Standard location service
- Significant location changes
- Continuous location updates

We have discussed these services before in *Chapter 3, Using Location in your iOS Apps – Core Location*. To use these services in the background, we just need to add location as a value in the `Info.plist` file's `UIBackgroundModes` key.

Besides background processing, iOS also includes options to put the application into a suspended mode, when the user presses the **Home** button the iPhone. When you double-tap the **Home** button and re open the app, the app resumes normal operation. This is done internally by iOS, by utilizing the memory efficiently. For most applications, you will not need to change this behavior, but in case of memory-consuming apps or mission-critical applications, you might need to understand how to effectively manage your application code around the various iOS application states. Let's look a bit deeper into the various states of an iOS application.

Understanding the iOS application life cycle

The default behavior of iOS applications is intended to be *Fast Launch, Short Use*. A typical user will pull out his/her iPhone, use an application, such as weather, local search, e-mail, or messaging, use it for a few minutes, and put it back in his/her pocket. Let's see the various stages in the application as the user carries out various tasks on his iPhone.

The entry point of every iOS application is the main function, such as any C program. Looking back at our WeatherPackt application, our `main.m` file contains the following code:

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

#import "AppDelegate.h"

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

The `UIApplicationMain` function is the core of the main method, taking four parameters. The first two being run-time arguments, and the third parameter is the name of the application principal class, usually `nil` for the third parameter as it is the principal class itself, the fourth being the application's delegate class responsible for the overall integration of our code with the system, in our case it is the `AppDelegate` class. The `UIApplicationMain` function also loads the main XIB file through the `UIApplicationDelegate` object. Open the `AppDelegate.h` file from our WeatherPackt application, and you will find that it implements the `UIApplicationDelegate – UIApplicationDelegate`.

The various states of an application can be as follows:

State	Description
Not running	The application has not been started or has been terminated by the system. This could happen due to the system's automatic graceful termination of the application based on memory usage.
Inactive	The application is running in the foreground, but not receiving events, may be because of an incoming phone call, or an SMS, or because the device has auto-locked after being idle for a few minutes. Another reason for the inactive state to be enabled could be when the device transitions from one state to another.
Active	Normal running state of the application, responding to user inputs, and updating display.
Background	The application is in background (iOS 4 and above only).
Suspended	The application is suspended and no background code is being run.

The application delegate contains the following methods to manage the transition to/from these states.

- `application:didFinishLaunchingWithOptions:`
- `applicationDidBecomeActive:`
- `applicationWillResignActive:`
- `applicationDidEnterBackground:`
- `applicationWillEnterForeground:`
- `applicationWillTerminate:`

Launching the application calls the `application:didFinishLaunchingWithOptions` method, from our `WeatherPackt` example. You will see that the various tab views' XIB files are associated with the respective View controllers, and control is passed on to the main `tabBarController`. You will also find the code template for all the other states ready within in the `AppDelegate.m` file when you created the new project.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindowalloc] initWithFrame:[UIScreen mainScreen]
        bounds]];
    // Override point for customization after application launch.
    UIViewController *viewController1, *viewController2,
        *viewController3;

    if ([[UIDevicecurrentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPhone)
    {
        viewController1 = [[FirstViewControlleralloc]
            initWithNibName:@"FirstViewController_iPhone"bundle:nil];

        viewController2 = [[SecondViewControlleralloc]
            initWithNibName:@"SecondViewController_iPhone"bundle:nil];

        viewController3 = [[ThirdViewControlleralloc]
            initWithNibName:@"ThirdViewController_iPhone"bundle:nil];
    }

    else
    {
        viewController1 = [[FirstViewControlleralloc]
            initWithNibName:@"FirstViewController_iPad"bundle:nil];
    }
}
```

```

viewController2 = [[SecondViewControlleralloc]
    initWithNibName:@"SecondViewController_iPad"bundle:nil];

viewController3 = [[ThirdViewControlleralloc]
    initWithNibName:@"ThirdViewController_iPad"bundle:nil];
}

self.tabBarController = [[UITabBarControlleralloc] init];
self.tabBarController.viewControllers =
    [NSArray arrayWithObjects:viewController1,
        viewController2,viewController3, nil];

self.window.rootViewController = self.tabBarController;
[self.windowmakeKeyAndVisible];
return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    /*Sent when the application is about to move from active to
    inactive state. This can occur for certain types of temporary
    interruptions (such as an incoming phone call or SMS message), or
    when the user quits the application, and it begins the transition
    to the background state.

    Use this method to pause ongoing tasks, disable timers, and
    throttle down OpenGL ES frame rates. Games should use this method
    to pause the game.
    */
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    /*Use this method to release shared resources, save user data,
    invalidate timers, and store enough application state information
    to restore your application to its current state in case it is
    terminated later.

    If your application supports background execution, then this
    method is called instead of applicationWillTerminate: when the
    user quits.
    */
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{

```

```
    /*Called as part of the transition from the background to the
       inactive state; here you can undo many of the changes made on
       entering the background.
    */
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
    /*Restart any tasks that were paused (or not yet started) while the
       application was inactive. If the application was previously in
       the background, optionally refresh the user interface.
    */
}

- (void)applicationWillTerminate:(UIApplication *)application
{
    /*Called when the application is about to terminate.
       Save data if appropriate.
       See also applicationWillEnterBackground:.
    */
}
```

As we discussed before, for the most part of our application development process, we will most likely not be using these methods; however, in specialized cases, if the need arises, we can extend the default behavior of our app – loading, pausing, resuming, and exiting, by using these `UIApplication` methods, based on our business logic. Apple has an extensive documentation on this subject, which you can refer to at the following URL: <http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/CoreApplication/CoreApplication.html>

Push notifications - overview

Push notifications are an easy means of notifying the user of a specific event. It could be as simple as an alert for your favorite stock price (if it exceeds a min/max price value in a trading session) to a bit complex as a geo alert. If you enter a specific region or a city, remember our region monitoring example, which is a simple example of push notifications.

Push notifications originate from a server that processes the user's iOS device behavior or state to send notifications to the device. Such notifications can be stored if the device is offline. However, the general behavior is to store and forward the notification.

Local notifications

With iOS 4 and higher, we also have the ability to fire local notifications from within our application. An example of local notification would be setting a local notification for an event happening at a future date.

The `UILocalNotification` class is needed by our app to schedule a local notification. It has the following three properties:

1. **Scheduled time:** The date and time for iOS to deliver the notification
2. **Notification type:** The notification type could be a simple alert, application icon badge (for example, the number of e-mails you have unread in your mail app), or playing a sound
3. **Custom data:** Location notifications could also include custom `NSDictionary` data



Each application can only have a maximum 64 local notifications scheduled. Anything greater than that will be discarded.

Time for action – using local notifications

We create a simple example of an app that uses local notifications. We also use the **application badge** (the number that gets appended to your `Email` app on the iPhone, signifying the unread e-mail messages or similar number on the `Message` App). We control the application badge with the new iOS 5 control `UIStepper` that is basically a UI control for increasing or decreasing a value. In our case, it is the application badge number.

1. We begin with creating a new single View application named `Local Notifications`. In the `ViewController.h` file, we declare a variable of type `UILocalNotification` named `localNotification` that will be responsible for controlling the local notifications in our app. We create a `UIButton` variable named `stopNotifications`, which will be used to stop the notifications from happening.
2. We now create an object of class `UIStepper` named `badgeStepper`, which is used to increase/ decrease the value of our application badge. Another variable `badgeText` of type `UITextField`, is used to render the value received from the `UIStepper` object.

- 3.** Next, we define an `IBAction` - `stepperChanged` that will be called when the value of `UIStepper` changes. We then create the `IBOutlet` for the variables required to connect in our NIB file.

```
@interface ViewController : UIViewController
{
    UILocalNotification *localNotification;
    IBOutletUIButton    *stopNotifications;
    UIStepper           *badgeStepper;
    UITextField         *badgeText;
}

- (IBAction)stepperChanged:(id)sender;

@property (retain) IBOutlet UILocalNotification *localNotification;
@property (retain) IBOutlet UIStepper *badgeStepper;
@property (retain) IBOutlet UITextField *badgeText;
```

- 4.** In our `ViewController.m`, we synthesize the property variables, and in our `viewDidLoad` method, we proceed to initialize the `localNotification` object as follows:

```
localNotification      = [[UILocalNotification alloc] init];
NSDate *currentDate    = [NSDate date];
localNotification.fireDate=[currentDate
    dateByAddingTimeInterval:60];
localNotification.timeZone=[NSTimeZone defaultTimeZone];
localNotification.alertAction=@"Open the App";
localNotification.alertBody=@"Daily Reminder - Minute by Minute ";
localNotification.repeatInterval = NSMinuteCalendarUnit;
localNotification.applicationIconBadgeNumber=10;

[[UIApplication sharedApplication]
    scheduleLocalNotification:localNotification];
```

- 5.** The `fireDate` property of `UILocalNotification` is very important. If it is not defined, the notifications won't be fired. We define it as an event happening at 60 seconds from the current date. So, the notifications will start 60 seconds from the current date. Or one minute after the application is loaded.

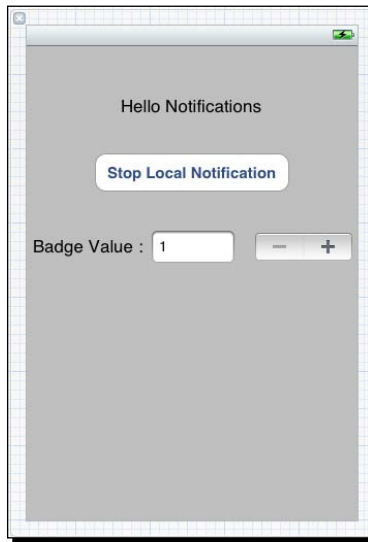
6. The `repeatInterval` property defines the interval at which the notification will be rescheduled. We define it as 1 minute through the `NSMinuteCalendarUnit` constant. Next, we set the application badge to 10, by default.
7. To schedule the notification, we call the `application:scheduleLocalNotification` method of `UIApplication`. The `sharedApplication` method returns the singleton application instance, which is the current application instance. We pass the `localNotification` object declared before to the `scheduleLocalNotification` method. If we need to run the local notification immediately, we can call the `presentLocalNotificationNow` method.
8. To stop the notifications, we define the `stopNotifications` action as follows:


```
- (IBAction)stopNotifications: (id) sender
{
    [UIApplicationsharedApplication]
        cancelLocalNotification:localNotification];
    [UIApplicationsharedApplication].applicationIconBadgeNumber=
        (NSInteger)badgeStepper.value;
}
```
9. The `cancelLocalNotification` method of `UIApplication` is called to stop the local notification. To cancel all the notifications, we can use the `cancelAllLocalNotifications` method. We also update the application badge with the current value from the `UIStepper` object `badgeStepper`.
10. The `stepperChanged` `IBAction` is used to fetch the latest value from the `UIStepper` object, and is passed to the `badgeText` `UITextField`:


```
- (IBAction)stepperChanged: (id) sender
{
    badgeText.text= [ [NSStringalloc] initWithFormat:@"%%.f",
        badgeStepper.value];

    [UIApplicationsharedApplication].
        applicationIconBadgeNumber= (NSInteger) badgeStepper.value;
}
```

- 11.** We construct our UI in the NIB as shown in the following screenshot, and connect the `UIStepper` to `stepperChanged` `IBAction`, **Stop Local Notification** button to the `stopNotifications` `IBAction`. Similarly, we connect the respective outlets for `badgeText` and `badgeStepper`.



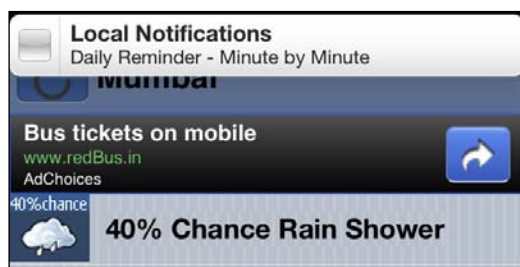
- 12.** Running the example on an iPhone with iOS 5 produces the following output:



- 13.** Change the value of the `UIStepper` to 5, and observe the application badge.



- 14.** Lastly, observe how the alerts show up when you are running some different applications on your iPhone.



What just happened?

By using the local notification, combined with the new notification center in iOS 5, we are able to engage the user at specific intervals of time, thereby increasing the user's visibility of our application and hence, increased business case for our iPhone app.

We also showed you how to create application badges for your app that can quickly notify the user of the tasks pending with respect to our application. In the last chapter, *Building a Social Governance App*, we can use this to signify the number of new social issues (after our last use of the application) pending in our city.

Have a go hero – add local notifications to WeatherPackt

Another use of local notifications could be to display weather alerts to the user at specific intervals. In the case of our `WeatherPackt` application, we could show a local notification every morning at six o'clock, by retrieving the forecast data.

Go ahead, make the change, and share the updated code with us. We would love to include your code in the main app.

Pop quiz – the rocket science

1. The Core Motion manager runs in its own thread.
 - a. True
 - b. False
2. With iOS 5, Core Motion can run in the background mode using the `UIBackgroundModes` Key
 - a. Yes
 - b. No
3. What is the maximum number of local notifications supported by an application in iOS 5?
 - a. 8
 - b. 16
 - c. 32
 - d. 64

Summary

In this chapter, we learned some advanced topics for iOS 5 including

- Using directions with magnetometer and GPS
- Using the Motion Manager
- Understanding iOS application life cycle and background apps
- Push messages

We will use some of these new learning in the apps that we will build in the forthcoming chapters.

8

Local Search—PacktLocal

Local Search is the darling app on most mobile phones/smartphones these days. Everyone loves to search for nearby pizza outlets or the nearest movie theatres, restaurants to eat, nightclubs to spend a good evening/night, and so on. Apps, such as foursquare, Gowalla, and Yelp, allow users to find such local content using their smartphones.

foursquare goes further with a **gamification** strategy that benefits both the end users and the venues owners (for example, a restaurant owner), by offering loyalty programs, discounts, and user badges to flaunt, depending upon how many times a user has checked-in to a venue.

In this chapter, we look at how the foursquare extensive API sets and builds a local search app named PacktLocal.

In this chapter, we will deal with the following topics:

- ◆ Consuming the foursquare venue API
- ◆ Building a simple UI for local search
- ◆ Saving venue information on the device (caching with SQLite)
- ◆ Adding a geo-fencing support
- ◆ Building the app with UI and code

So let's get on with it....

Consuming the foursquare venue API

In *Chapter 3, Using Location in your iOS Apps – Core Location*, we looked at a simple location-based foursquare example (see the example named *Hello Location – Foursquare*) that fetched nearby venues, based on the device location. However, there are more venue endpoints in the foursquare API. An extensive list can be found at <https://developer.foursquare.com/docs/>. For the purpose of our discussion and our app – *PacktLocal* – we will focus on the venue API, and aspects related to it.

The foursquare venue API consists of the following API calls:

- ◆ Add a venue
- ◆ Get a list of venue categories
- ◆ Explore the recommended and popular venues
- ◆ Search for venues
- ◆ Get trending venues
- ◆ Get detailed venue information, including tips, photos, links, events, and the number of people at a current venue right now

Most of the API endpoints do not need authentication, except for the *Add venue*, which requires user authentication.

Venue categories

Let's begin exploring the API starting from the venue categories.

Time for action – consuming the foursquare venue API - categories

1. Open Xcode, and start a new project named *Hello foursquare*, using the *Tabbed Application* template.
2. Add a new header file (*.h* extension) to your project, by selecting the **File | New | New File | C and C++** option from Xcode file menu. Name the new file as *Configuration.h*. This file will hold the foursquare client ID and client secret, and any other foursquare configuration that we might need in future. Keep the configuration in one place, make the code more robust and easy-to-extend, as anyone can start using the code by replacing the configuration values. You can define the values in the *Configuration.h* file as follows:

```
#ifndef Hello_foursquare_Configuration_h
#define Hello_foursquare_Configuration_h
```

```
#define CLIENT_ID      @"XXXXXXXXXXXXXXXXXX"
#define CLIENT_SECRET @"YYYYYYYYYYYYYYYYYY"

#endif
```

Here, xxxx is your client ID, and yyyy is your client secret.

3. Next, add the Core Location, MapKit, Twitter, and SQLite framework to your project. This is similar to our previous projects and examples. We will discuss more on how these libraries are used, as we learn more over the course of this chapter. Rewind back to *Chapter 6, Events App - PacktEvents*, for a quick look at some of these topics, notably JSON parsing, Twitter, and SQLite.
4. In the `Hello_foursquareFirstViewController.h` and `Hello_foursquareFirstViewController.m` files, implement the Core Location delegate and the Core Location manager's methods. We also use the region monitoring through `CLRegions` for San Francisco, Mumbai, and New York.

```
CLLocationCoordinate2D regionCords =
    CLLocationCoordinate2DMake(37.33 , -122.03);

CLRegion *sanFranciscoBoundary = [[CLRegion alloc]
    initWithCenter:regionCords
    radius:5000
    identifier:@"San Francisco"];

regionCords = CLLocationCoordinate2DMake(40.71490, -74.00679);

CLRegion *newYorkBoundary = [[CLRegion alloc]
    initWithCenter:regionCords
    radius:5000
    identifier:@"New York"];

regionCords = CLLocationCoordinate2DMake(19.142472, 72.841198);

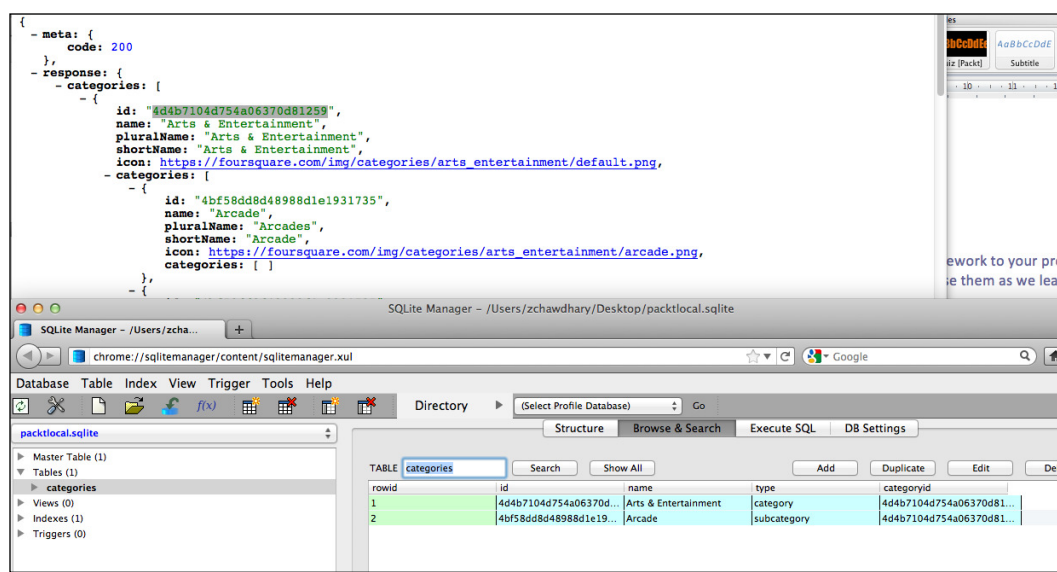
CLRegion *mumbaiBoundary = [[CLRegion alloc]
    initWithCenter:regionCords
    radius:5000
    identifier:@"Mumbai"];

[locationManager startUpdatingLocation];
[locationManager
    startMonitoringForRegion:sanFranciscoBoundary];
[locationManager startMonitoringForRegion:newYorkBoundary];
[locationManager startMonitoringForRegion:mumbaiBoundary];
```

5. After we are done with the location handling code, we call the `initializeDatabase` method that is used to create our initial tables. For now, we create the categories table using the following table structure:

```
CREATE TABLE IF NOT EXISTS 'categories' ('id' VARCHAR PRIMARY
KEY NOT NULL , 'name' VARCHAR, 'type' VARCHAR, 'categoryid'
VARCHAR, 'subCategoryId' VARCHAR, 'icon' VARCHAR)
```

6. To get an idea on how we are going to store the category hierarchy, please see the following screenshot, where we depict the response from the `foursquare` category API, and our local SQLite database storage for the first few category values:



7. After we are done with the database initialization, we call the `getfoursquareCategories` method that does a `NSURLConnection` call to the `foursquare` API, for retrieving categories information, as follows:

```
NSString *url = [NSString
stringWithFormat:@"https://api.foursquare.com/v2
/venues/categories?&client_id=%@&client_secret=%@",
CLIENT_ID, CLIENT_SECRET];
```

Here `CLIENT_ID` and `CLIENT_SECRET` are values that we defined in `Configuration.h`.

- 8.** Once the connection finishes loading the data through the `connectionDidFinishLoading` method, we parse the received JSON, and retrieve the category information by converting the JSON data first to an `NSDictionary`, and thereafter from an `NSDictionary` to `NSArray`, for easier parsing. We then generate our `INSERT` statements, and populate the SQLite database with the category information. Note that we also receive the category icon from foursquare, which we have inserted in our table.

```
- (void)connectionDidFinishLoading: (NSURLConnection *)connection
{
    NSError *jsonError;
    NSDictionary *dictionary;
    NSArray *items;

    dictionary= [NSJSONSerialization JSONObjectWithData:jsonContent
options:NSJSONReadingAllowFragments error:&jsonError];

    if([dictionary count]>0)
    {
        items          = [NSArray arrayWithObject:[dictionary
objectForKey:@"response"]objectForKey:@"categories"]];
        NSUInteger count = [[items objectAtIndex:0] count];
        for(NSInteger i=0;i<count;i++)
        {
            NSString *categoryId = [[[items objectAtIndex:0]
objectAtIndex:i]objectForKey:@"id"];

            NSString *categoryName = [[[items objectAtIndex:0]
objectAtIndex:i]objectForKey:@"name"];

            NSString *categoryIcon = [[[items objectAtIndex:0]
objectAtIndex:i]objectForKey:@"icon"];

            NSString *categoryType = @"category";

            NSString *subCategories = [[[items objectAtIndex:0]
objectAtIndex:i]objectForKey:@"categories"];

            NSString *icon = [[[items objectAtIndex:0]
objectAtIndex:i]objectForKey:@"icon"];
```

```
NSString *insertStatement = [[NSString alloc] initWithFormat:
@"INSERT OR REPLACE INTO '%@'('%@','%@','%@','%@','%@') VALUES('%@'
',\\\"%@\\\",\\\"%@\\\",'%@','%@')",categoriesTableName,@"id",@"name",@"ty
pe",@"categoryid",@"icon",categoryId,categoryName,categoryType,cat
egoryId,icon];

        if(sqlite3_open([sqliteFileName UTF8String],
            &database)==SQLITE_OK)
        {
if(sqlite3_exec(database, [insertStatement UTF8String],NULL,
NULL, &sqliteError)==SQLITE_OK)
        {
            NSLog(@"category table populated");
        }
        else
        {
            NSLog(@"%s",sqliteError);
        }
        }
    } // end of for loop
    [self showCategoriesFromLocal];
}
}
```

- 9.** Most categories in the foursquare category hierarchy also have more subcategories within them, for example the **Airport** category has the following sub categories: **airport food court**, **airport gates**, **airport lounges**, **airport terminals**, and so on. Feel free to use them as your application demands. For now, we have captured the same in the `subCategories` variable.
- 10.** We now define one more method that reads the values from the locally stored categories table. We name this function as `showCategoriesFromLocal`. A simple select statement is executed here, which retrieves the category info from the local SQLite database, and adds it to an NSArray variable that is used to render on a UITableView.

```
- (void) showCategoriesFromLocal
{
    categories = [[NSMutableArray alloc] init];
```

```

        if(sqlite3_open([sqliteFileName UTF8String],
            &database)==SQLITE_OK)
        {
            NSString *selectStatement =[[NSString alloc]initWithFormat:
                @"SELECT * from %@",categoriesTableName];
            sqlite3_stmt *sqlStatement;

            if(sqlite3_prepare_v2(database, [selectStatement UTF8String],
                -1, &sqlStatement, NULL)==SQLITE_OK)
            {
                while(sqlite3_step(sqlStatement)==SQLITE_ROW)
                {
                    NSString *titleDataText=[[NSString stringWithUTF8String:
                        (char *)sqlite3_column_text(sqlStatement, 1)];

                    if(![categories containsObject:titleDataText])
                    {
                        [categories addObject:titleDataText];
                    }
                }
            }
        }
    } // end of if of sqlite3 open

    [myTableView reloadData];
}

```

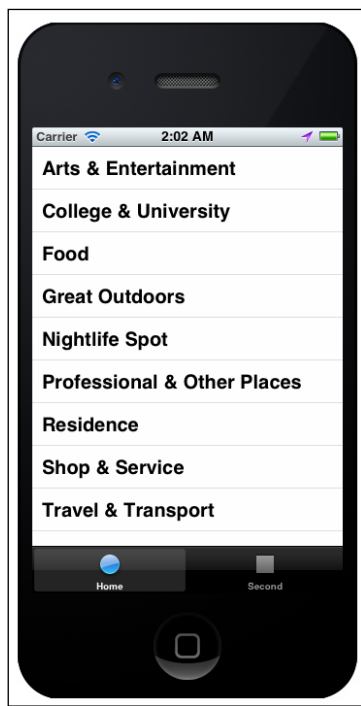
- 11.** Make sure you that use the right variable name in the `cellForRowAtIndexPath` delegate method of the `UITableView`. In this case, the `categories` variable is as follows:

```

NSString *cellContent=[categories objectAtIndex:indexPath.row];

```


- 12.** Running the application should produce the following result. Note we leave the UI design to the reader; we use storyboard here, which will be discussed in more detail in *Chapter 9, Location Aware News - PacktNews*. You could choose to use the default **Tabbed Application** template, without selecting the **Use Storyboard** checkbox option. Note that **storyboarding** only helps in easier user interface management. So, if you think you can handle the UI without storyboard, you can choose to not select it.



What just happened?

We consumed the `foursquare` venue category API using the JSON parsing API in iOS5, and created a local table for storing the category information. `foursquare` recommends purging the category information after a week. But, for most apps, this is highly unlikely, since the app is built around some of these popular categories. But, depending upon your requirement, use the API and caching wisely.

We used the storyboard for our app, for which we will have a full-length explanation coming in the next chapters. We also used a separate configuration file to keep the `foursquare` API keys and `Auth` configuration values separate from the core app. A similar approach could be used for region monitoring using `CLLocationRegion` and custom map annotations in separate files, to make the code flow better organized.

Recommended and popular venues

The recommended and popular venues API from foursquare are experimental API endpoints (API endpoints can be thought of as API URLs) that are added recently. So, be cautious to use them in your long-term app approach. This endpoint is different from the trending venues end point in the fact that the recommended and popular venue endpoint is more socially relevant, since it ranks the venues based on you and your friends, while the trending endpoint is more of an algorithmic count.

To learn more, and be updated on this API endpoints, keep an eye on <https://developer.foursquare.com/docs/venues/explore.html>.

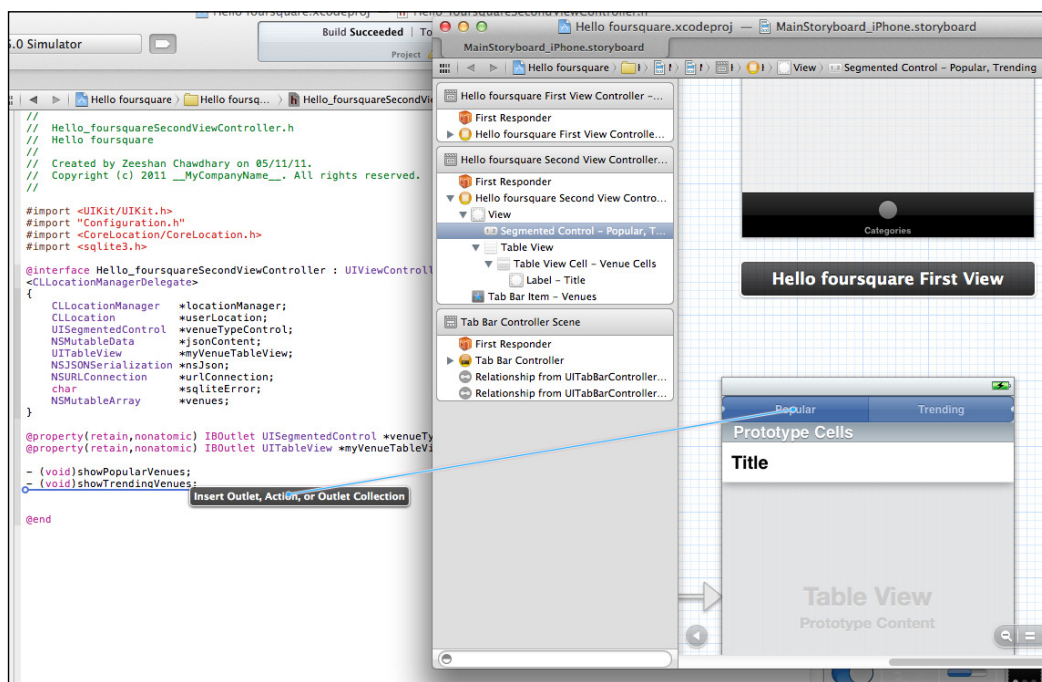
Time for action – recommended and popular venues

1. We continue with the same example app (Hello foursquare) as before. Open the project titled `Hello foursquare`, and open the iPhone storyboard file (`MainStoryboard_iPhone.storyboard`). In the second View controller View, make the UI look as the one shown in the next screenshot, by adding a `UISegmentedControl` and a `UITableView` instance to it. Make sure to type the identifier name for your **Prototype Cells** as `Venue Cells`.

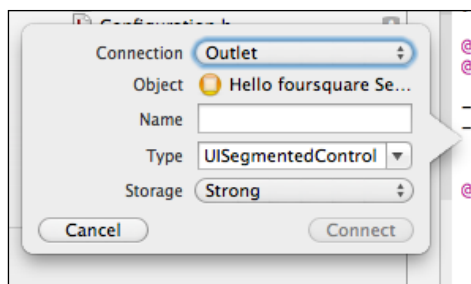


2. Rename the segmented controls to **Popular** and **Trending**, for showing popular venues and trending venues, respectively, according to the option selected.
3. In the `Hello_foursquareSecondViewController.h` file, expose the `UISegmentedControl` and `UITableView` as properties.


```
@property(retain, nonatomic) IBOutlet UISegmentedControl
*venueTypeControl;
@property(retain, nonatomic) IBOutlet UITableView
*myVenueTableView;
```
4. Create an IBAction by **Control+dragging** from the `UISegmentedControl` in Interface Builder, and dropping it on `Hello_foursquareSecondViewController.h`.



5. From the context menu that shows up, select **Action**, and name your action as `switchVenueType`:



6. Also declare two methods that we will use to fetch the popular and trending venues from the foursquare API:
 - (void) showPopularVenues;
 - (void) showTrendingVenues;
7. Include the Core Location and configuration header files in `Hello_foursquareSecondViewController.h`, besides including the necessary variables for the `NSURLConnection`, and a variable of type `NSMutableArray` that will hold the venues.
8. There are other ways to share variables between multiple views (from the first View controller to the second View controller). For example, sharing the `userLocation` variable from the `Hello_foursquareFirstViewController.h` file to the `Hello_foursquareSecondViewController.h` file, so that we can re-use the location attributes. For our learning, we recreate the same code; however, the readers are encouraged to use any other method they see fit. One way would be to store the user location in the SQLite database. Another way would be by storing the application settings through the `NSUserDefaults` class.
9. Your `Hello_foursquareSecondViewController.h` file should now look as follows:

```
#import <UIKit/UIKit.h>
#import "Configuration.h"
#import <CoreLocation/CoreLocation.h>

@interface Hello_foursquareSecondViewController : UIViewController
<CLLocationManagerDelegate>
{
    CLLocationManager    *locationManager;
    CLLocation           *userLocation;
    UISegmentedControl   *venueTypeControl;
    NSMutableData         *jsonContent;
    UITableView          *myVenueTableView;
}
```

```
    NSJSONSerialization *nsJson;
    NSURLConnection      *urlConnection;
    char                  *sqliteError;
    NSMutableArray        *venues;
}

@property(retain, nonatomic) IBOutlet UISegmentedControl
*venueTypeControl;
@property(retain, nonatomic) IBOutlet UITableView
*myVenueTableView;

- (void) showPopularVenues;
- (void) showTrendingVenues;
- (IBAction) switchVenueType: (id) sender;

@end
```

- 10.** In the class implementation for your second View controller (`Hello_foursquare SecondViewController.m`), we modify the `viewDidLoad` method to obtain the location from the location manager (initialized in the first View controller) as follows:

```
userLocation = [[CLLocation alloc] initWithLatitude:location
Manager.location.coordinate.latitude longitude:locationManager.
location.coordinate.longitude];
```

- 11.** Our `UISegmentedControl` is attached to a variable named `venueTypeControl` (by *Control*+dragging the mouse from the second View controller onto the `UISegmentedControl` in the Interface Builder, and selecting the outlet as `venueTypeControl`), and the default selected index is set to 0 – for popular venues.

```
venueTypeControl.selectedSegmentIndex=0;
```

- 12.** Next, we initialize the venues and the JSON variable, and call the `showPopularVenue` method, based on assumption taken in *point 10*.

```
venues = [[NSMutableArray alloc] init];
jsonContent = [[NSMutableData alloc] init];
[self showPopularVenues];
```

- 13.** `foursquare` now requires versioning information to be passed through some of the API URLs. This is to ensure that the client is up-to-date. The description for this can be found at the following URL: <https://developer.foursquare.com/docs/overview.html#versioning>

- 14.** We use the versioning parameter in the `showPopularVenue`, by using the `NSDate` and `NSDateFormatter` class instances. foursquare requires the versioning date format to be `YYYYMMDD`. Here is the code to achieve this:

```
NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
[dateFormatter setDateFormat:@"YYYYMMDD"];
```

```
NSDate *now = [NSDate date];
NSString *version = [NSString alloc]
initWithFormat:@"%@", [dateFormatter stringFromDate:now]];
```

- 15.** The code `[NSDate date];`, returns the current date, which is then formatted by the `dateFormatter`, and stored back in the `version` variable.

- 16.** The URL for the popular venue API call is then constructed, and the `NSURLConnection` request is sent.

```
NSString *url = [NSString stringWithFormat:@"https://api.
foursquare.com/v2/venues/explore?client_id=%@&client_
secret=%@&ll=%@,%@&v=%@", CLIENT_ID, CLIENT_SECRET, userLat, userLon, v
ersion];
```

- 17.** The JSON parsing then occurs through the `didReceiveData` and the `connectionDidFinishLoading` delegate methods. In the `connectionDidFinishLoading` method, we check for the type of the venue called for, by checking the value of the `UISegmentedControl` value, since the `foursquare api` for popular venues and trending venues returns different JSON payloads.

```
if (venueTypeControl.selectedSegmentIndex==0)
```

- 18.** We parse the JSON according to the type of venue called for, as well as the JSON payload received. Here is the full code for the `connectionDidFinishingLoading` method:

```
- (void)connectionDidFinishLoading: (NSURLConnection *)connection
{
    NSError *jsonError;
    NSDictionary *dictionary;
    NSArray *items;

    dictionary= [NSJSONSerialization JSONObjectWithData:jsonContent
options:NSJSONReadingAllowFragments error:&jsonError];

    if (venueTypeControl.selectedSegmentIndex==0)
    {
        if ([dictionary count]>0)
```

```
        {
            items = [NSArray arrayWithObject:
[[dictionary objectForKey:@"response"]
objectForKey:@"groups"]];

            items = [[[items objectAtIndex:0]
objectAtIndex:0]objectForKey:@"items"];

            NSUInteger count = [items count];
            for(NSInteger i=0;i<count;i++)
            {
                NSString *venueName = [[[items
objectAtIndex:i]objectForKey:@"venue"] objectForKey:@"name"];

                if(![venues containsObject:venueName])
                {
                    [venues addObject:venueName];
                }
            }

            // end of for loop
        }
    }
else
    {

        items = [NSArray arrayWithObject:[dictionary
objectForKey:@"response"]objectForKey:@"venues"]];
        NSUInteger count = [[items objectAtIndex:0] count];

        for(NSInteger i=0;i<count;i++)
        {
            NSString *venueName = [[[items
objectAtIndex:0]objectAtIndex:i]objectForKey:@"name"];

            if(![venues containsObject:venueName])
            {
                [venues addObject:venueName];
            }
        }
    }

    [myVenueTableView reloadData];
}
```

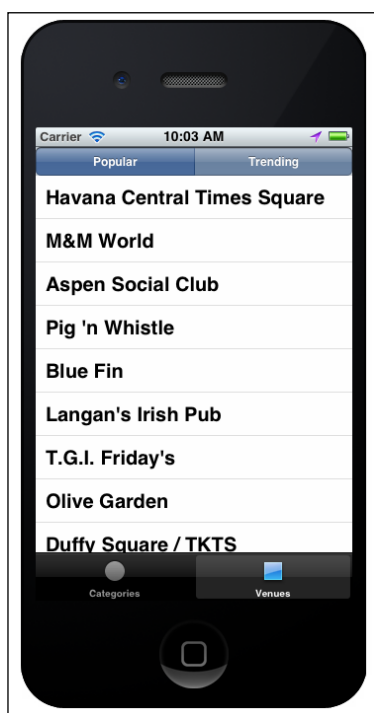
- 19.** The `showTrendingVenues` method is almost similar to the `showPopularVenues`, except for the change in the URL.

```
NSString *url = [NSString stringWithFormat:@"https://api.foursquare.com/v2/venues/trending?&client_id=%@&client_secret=%@&ll=%@,%@&v=%@", CLIENT_ID, CLIENT_SECRET, userLat, userLon, version];
```

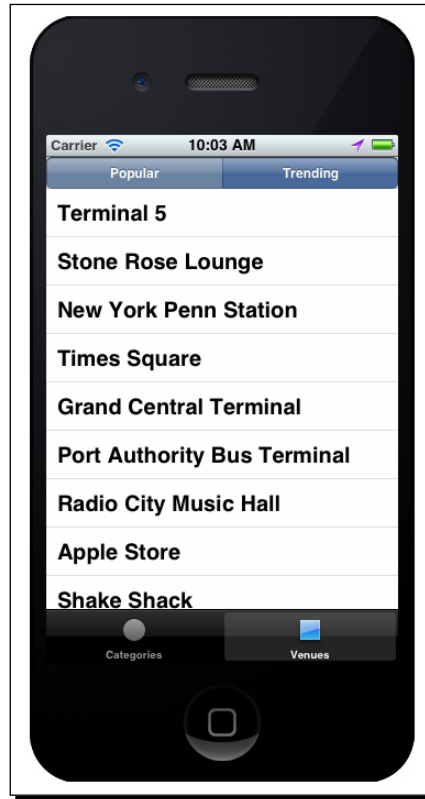
- 20.** Finally, the `switchVenueType` `IBAction` that is fired when we switch from the popular venue to the trending venue through the `UISegmentedControl` is defined as follows:

```
- (IBAction)switchVenueType:(id)sender {
    if (venueTypeControl.selectedSegmentIndex==0)
    {
        [self showPopularVenues];
    }else
    {
        [self showTrendingVenues];
    }
}
```

- 21.** Running the app now, produces the following results; notice the difference in the venues showing up in each of the segments; this one depicts the Popular venues:



While the screen shot below depicts the Trending venues:



What just happened?

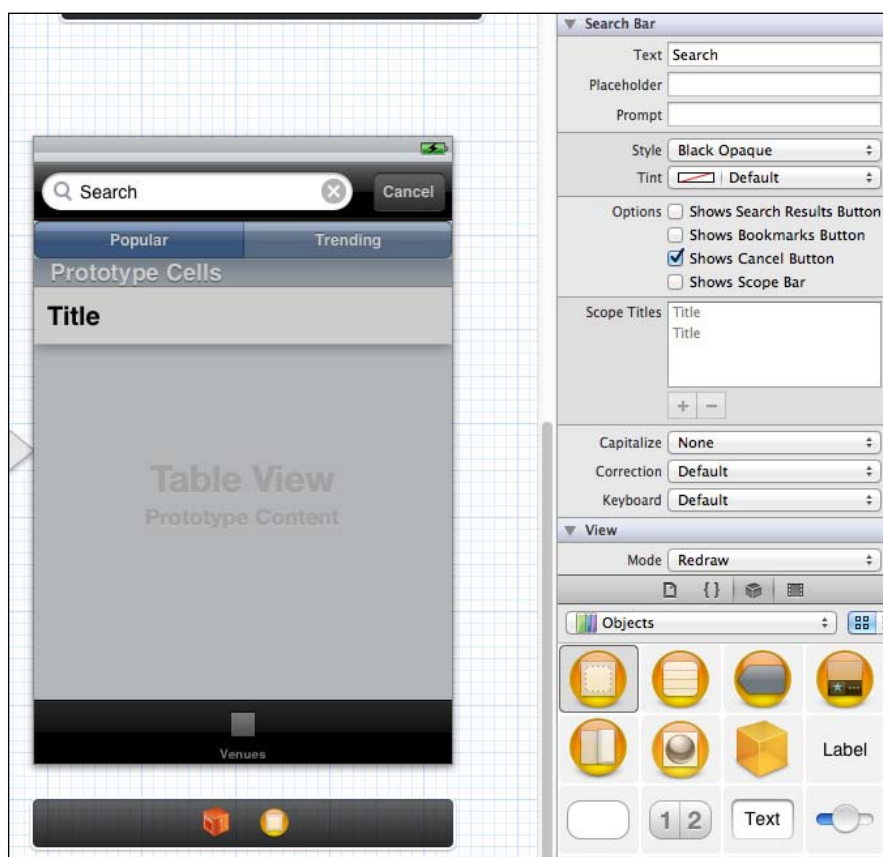
We looked at two popular `foursquare` venue identifier APIs – finding popular venues and finding the nearby trending venues. We introduced the `UISegmentedControl` as a UI display differentiation method, as well as learned how to use it to control the application logic, through the selective JSON parsing, based on the segment selected.

Search for venues

We now look at the `Search` API end point for venues. This would help us build our app, later in the chapter. `foursquare` recently added the `foursquare` venue mapping to its API. What it does is basically helps correlate `foursquare` venues with other venue providers, such as `yelp`, `wcities.com`, `tripadvisor`, `citysearch`, `menupages`, or any other popular HyperLocal venue information provider. Read more about it at <https://developer.foursquare.com/venues/mapping.html>. This mapping is also exposed in the search API - <https://developer.foursquare.com/docs/venues/search.html>.

Time for action – exploring the foursquare Search API

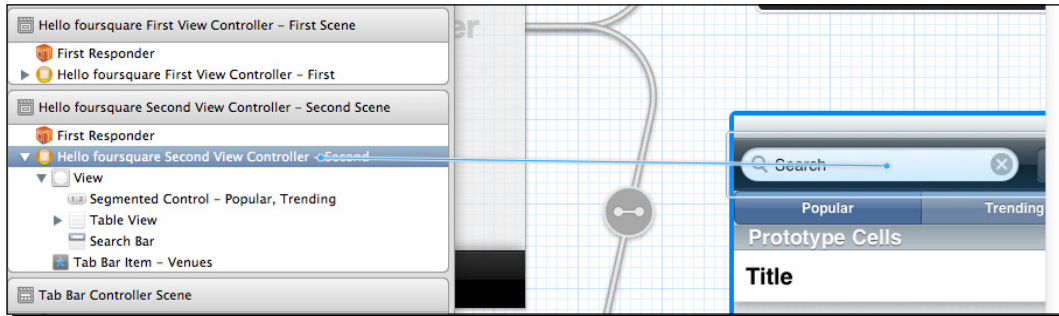
1. Continuing from where we left off from the Hello foursquare example, we add a `UISearchBar` to our second View controller in the Interface Builder.



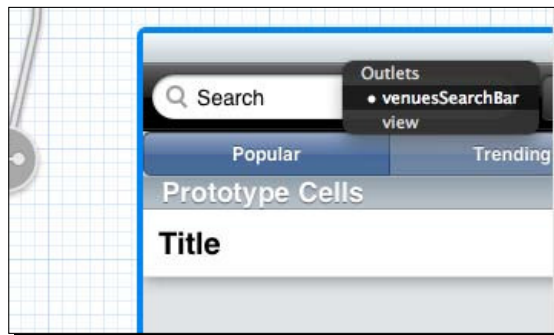
2. In the `Hello_foursquareSecondViewController.h` file, we define a variable of the type `UISearchBar` in the class declaration `UISearchBar *venuesSearchBar;`, and expose it as a property.

```
@property(retain, nonatomic) IBOutlet UISearchBar *venuesSearchBar;
```

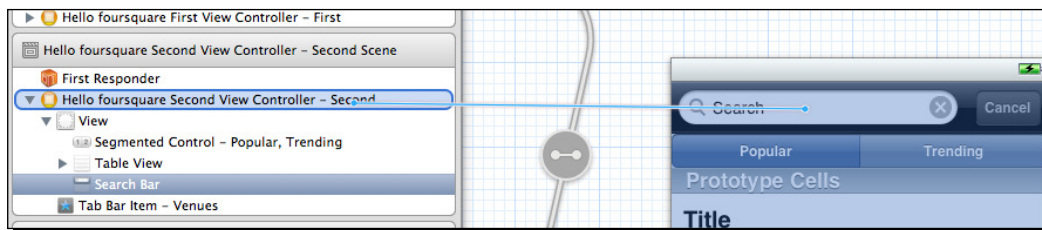
3. Connect the `venuesearchBar` outlet to the **Search** bar in the Interface builder by **Control+dragging** the mouse pointer from the second View controller to the **Search** bar.



Drop the mouse pointer on the **Search** bar, and select `venueSearchBar` as the outlet.



4. Connect the **Search** bar to the delegate (`UISearchBarDelegate`), by **Control+dragging** the mouse pointer from the **Search** bar to the second View controller delegate.



5. Add the `UISearchBarDelegate` delegate to the `Hello_foursquareSecondViewwController.h` class. We also declare a Boolean variable in **Search** to control the in-app search behavior, and JSON parsing accordingly.

6. The method that will search the foursquare API is declared as follows:

```
- (void) searchForVenues;
```

7. The `UISearchBarDelegate` protocol defines several methods for `UISearchBar`, including `searchBar:textDidChange`, `searchBarTextDidBeginEditing`, `searchBarTextDidEndEditing`, `searchBarCancelButtonClicked`, `searchBarSearchButtonClicked`, besides some more. Of our interest, are two of such methods, namely `searchBarCancelButtonClicked` and `searchBarSearchButtonClicked`, which we will implement.
8. In the `viewDidLoad` method of the `Hello_foursquareSecondViewController.m` file, we initialize the **Search** bar, and set the delegate to `self`. This is very important, as without the delegate property set, the **Search** bar will not trigger any actions.

```
venuesSearchBar = [[UISearchBar alloc] init];
venuesSearchBar.delegate=self;
inSearch         = false;
```

9. Now, on the **Search** bar's **Search** button-click, we hide the keyboard using the `resignFirstResponder` method of the `UIResponder` class, which is the superclass (from object-oriented paradigm) of `UIApplication` – our main application class. We then set the `inSearch` flag as `TRUE`, and call the `searchForVenues` method. The search term is captured in our `venuesSearchBar.text` property as shown in the following code:

```
- (void) searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
    [searchBar resignFirstResponder];
    inSearch = TRUE;
    venuesSearchBar.text=searchBar.text;
    [self searchForVenues];
}
```

10. The `searchForVenues` method is similar to the other methods that we used for fetching foursquare venues – trending and popular. The only change we need to do is change the calling URL to the following:

```
NSString *url = [NSString stringWithFormat:@"https://api.foursquare.com/v2/venues/search?&query=%@&client_id=%@&client_secret=%@&ll=%@,%@&v=%@", venuesSearchBar.text, CLIENT_ID, CLIENT_SECRET, userLat, userLon, version];
```

Here, we pass the **Search** bar text as a search parameter to the foursquare venue search API.

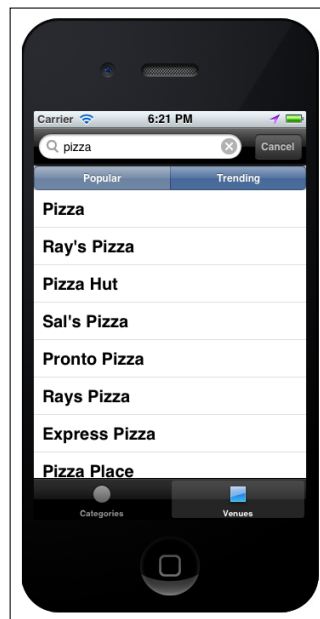
- 11.** Then, the `connectionDidFinishLoading` takes over; much of the code remains the same, except for the `inSearch` flag checking and conditional JSON parsing of the search response.

```
if (inSearch)
{
    items = [NSArray arrayWithObject:[dictionary
        objectForKey:@"response"]objectForKey:@"venues"];
    NSUInteger count = [[items objectAtIndex:0] count];
    for (NSUInteger i=0;i<count;i++)
    {
        NSString *venueName = [[[items
            objectAtIndex:0]objectAtIndex:i]objectForKey:@"name"];
        if (![venues containsObject:venueName])
        {
            [venues addObject:venueName];
        }
    }
}
```

- 12.** We then reset the `inSearch` flag to false, after our table view has been updated.

```
[myVenueTableView reloadData];
inSearch=FALSE;
```

- 13.** Run the application in the iOS simulator, and then select the **Venues** tab. You should see the following output for the keyword search for `pizza`:



What just happened?

Search is an integral part of any mobile app. With contextual location, search becomes even more powerful. `foursquare` understands these market and user expectations, and handles the venue search in an excellent manner, with its ever growing algorithms and metrics to show the best result to its users. We learnt how to use the powerful `foursquare` search API in our example.

We briefly looked at a couple of `foursquare` endpoints. There are more endpoints documented at https://developer.foursquare.com/docs/index_docs.html. We will try to incorporate as many API endpoints in our next voyage building the `PacktLocal` app!!

Find the code for this example and previous examples on the book's website: project titled `Hello foursquare`.

Building an UI for our local search app - PacktLocal

A good and successful app needs the right ingredients: a cool UI Design, application logic, great backend, and killer features to disrupt the market. We have often discussed the importance of a good design. For a programmer, the application logic, great backend, and killer features are aspects that can be taken care of. The design should be a non-programmer role, leaving it to the guys who know it the best. Throughout the book and various apps, we have tried to get the best design (legit, or open source, or creative commons images) that fits our app requirements, based on excellent communities, such as <http://www.dribbble.com>, <http://www.365psd.com>, and the holy grail of design on the web - <http://www.smashingmagazine.com>.

For `PacktLocal`, we will re-use the excellent app and design done by the guys at <http://zhphree.com/> for the Palm WebOS (now HP WebOS) version of the `foursquare` app. The complete source code is available at <https://github.com/foursquare/foursquare-palmpre>. Of interest to us is the excellent UI layouts and icons, which we will import in our Xcode project, and re-use wherever applicable. For the app icon, we will use a restaurant icon from <http://www.ioandecean.info/2011/05/restaurant-farfurie-tacamuri-free-psd/>, since the focus of our app is hotels and restaurants venues.

We will use the **Tabbed Application** template within Xcode to build PacktLocal with two tabs - one for showing nearby venues and the other for search. A `settings` bundle will be added in the app to allow for more customization.

Saving venue information on the device

Now that we have defined our UI, we move further to the implementation of PacktLocal in our Hello foursquare example. We saw how to fetch the popular and trending venues for PacktLocal. We will use the same logic, but enhance it by caching the top 30 venues in the local SQLite database.

We saw how to use SQLite in our WeatherPackt and PacktNews apps. For PacktLocal, we will need two tables - one for storing the venue categories, and the other for storing the 30 venues fetched from the foursquare API. Further venue information, such as check-ins, tips, and so on, will be handled on the fly due to the dynamic nature of the content. We use the same category table structure as we saw in the Hello foursquare example.

```
CREATE TABLE "categories" ("id" VARCHAR PRIMARY KEY NOT NULL , "name"
VARCHAR, "type" VARCHAR, "categoryid" VARCHAR, "subCategoryId" VARCHAR)
```

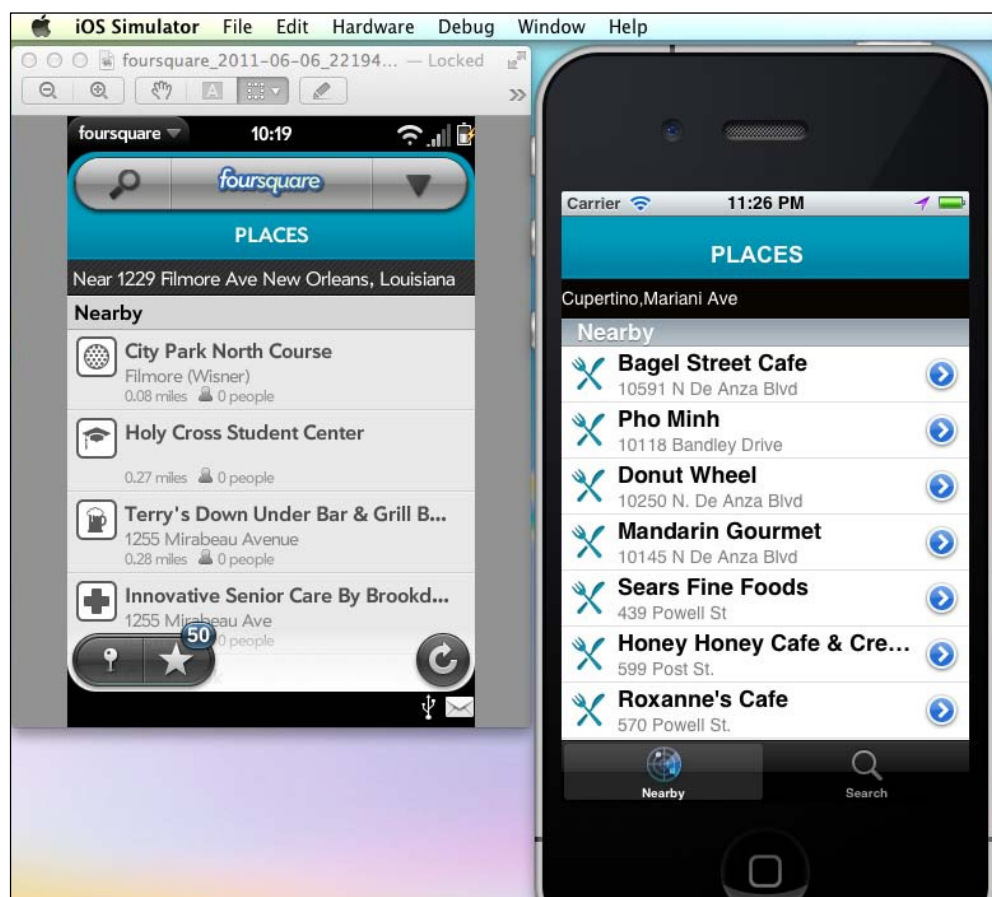
For the venues, we use the following structure (loosely-based on the response got from the venue search API):

```
CREATE TABLE "venues" ("id" VARCHAR PRIMARY KEY NOT NULL , "title"
VARCHAR NOT NULL , "address" TEXT, "city" VARCHAR, "zip" VARCHAR, "country"
VARCHAR, "latitude" DOUBLE, "longitude" DOUBLE, "images" VARCHAR, "category"
VARCHAR, "checkins" INTEGER, "userscount" INTEGER, "tipcount"
INTEGER, "phone" VARCHAR)
```

Building the app: PacktLocal

As mentioned before, our PacktLocal app is heavily influenced by the Palm pre-version of the foursquare app (source code available at <https://github.com/foursquare/foursquare-palmpre>).

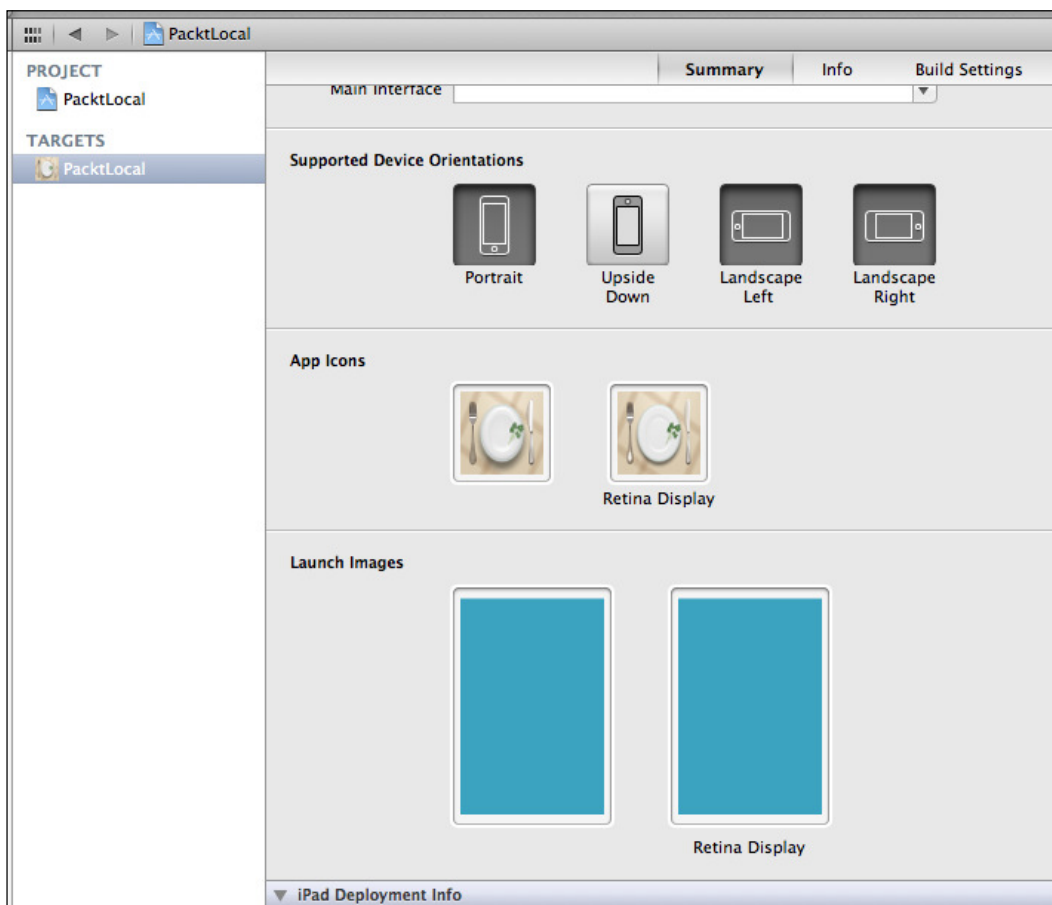
To give the reader an idea, here is a side-by-side comparison of the Palm version and the one we are going to build:



Time for action – building the app - PacktLocal

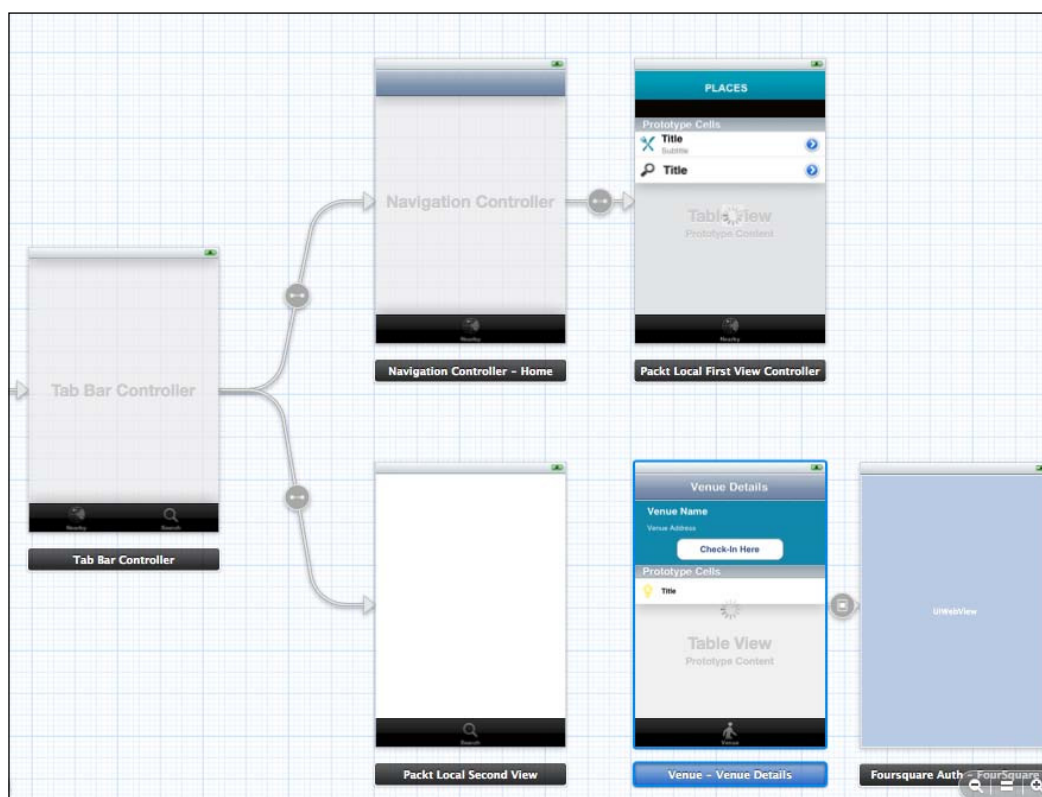
1. Create a new project using the **Tabbed Application** template. Name the application as **PacktLocal**.
2. Add the SQLite, MapKit, Twitter, and Core Location frameworks to your project.
3. Add a **Settings.bundle** to your project from the **Xcode | File | New File | Resource** option. We will use this to enable the offline/online support for our app. to enable or disable the SQLite venue cache, as well as to store the access token from foursquare authentication (more information on this topic can be found at <https://developer.foursquare.com/docs/oauth.html>).

4. Configure the **App Icons** and **Launch Images** as shown in the following screenshot. We use the icon from <http://www.ioandecean.info/2011/05/restaurant-farfurie-tacamuri-free-psd/>, and the splash image from the Palm Pre-source code (file named `4sq-login-scene.psd` – you need to strip the other layers, and just use the background image).

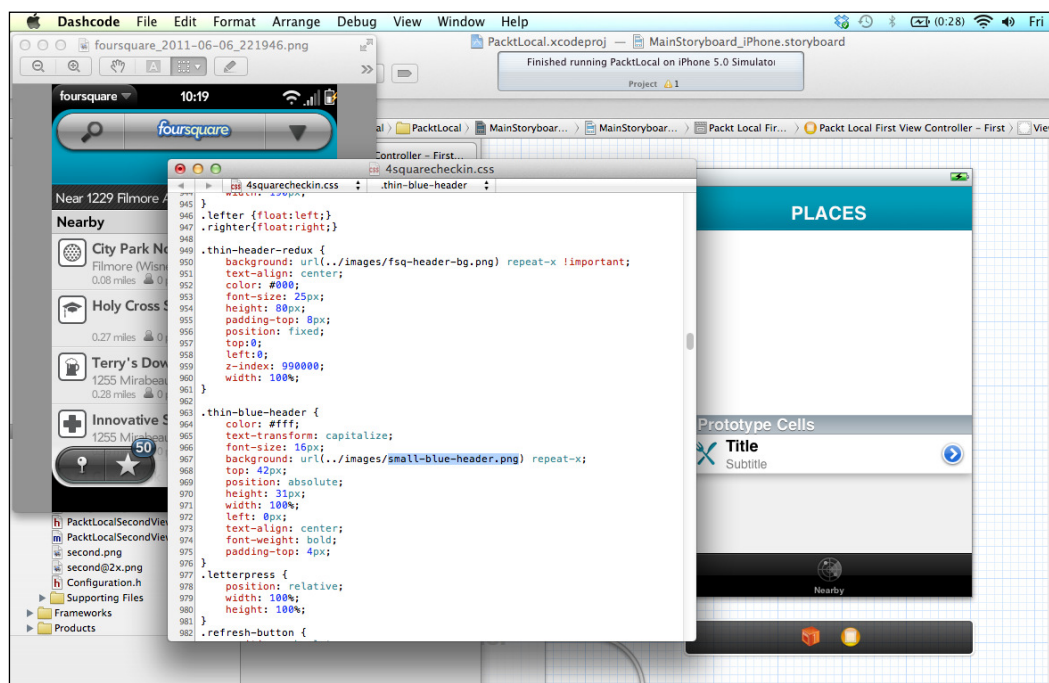


5. Add a new Objective-C class to your project from **File | New | New File | Cocoa Touch** option named **Venue**. This class will hold the individual venue information retrieved from *foursquare*, including venue id, venue name, venue address, venue city, venue check-in count, and other attributes.
6. Extend the `Venue` Class by subclassing it from the `UIViewController` class. We will come back to this class a bit later.

7. Add a new `UIViewController` class to your project by selecting **File | New | New File | Cocoa Touch | UIViewController** class option named `foursquareAuth`. We will need this to implement the `foursquare` user authentication that is needed for the venue check-in operation. We will learn more about this in the later sections.
8. Coming back to our main application code, `PacktLocalFirstViewController`. In the `PacktLocalFirstViewController.h` and `PacktLocalFirstViewController.m` files, add the code for the location manager, region monitoring, as we have done in our previous apps. As well as, initialize the SQLite database.
9. Our main `Storyboard` looks as follows:



- 10.** We style the **Home** screen by digging the Palm Pre-source code, inspecting the CSS, and identifying the images needed. Programming gurus will know that understanding someone else's code is a good way to test your programming skills, and learn from other's code.



- 11.** In our `PacktLocalFirstViewController.m` file, after the SQLite database is initialized, we fetch the list of venue categories from foursquare API, and store it in the categories table. Now, as soon as the location is updated through the `didUpdateToLocation` method, we call the `showNearbyVenues` method that hits the foursquare API, and fetches the nearby venues tagged by `food`. On success of the API call through the `connectionDidFinishLoading` method, we store the 30 venues in our `venues` table (after purging previous venues). This also makes our app work well with the `foursquare` terms and conditions.
- 12.** The `cellForRowAtIndexPath` method of the `UITableView` is something new here. When we are processing the venue information (by using the `inCategories` flag), we use the `Venue` class that we added to our project before, to hold the venue information.

- 13.** This Venue class is used to create an array of all the 30 venues, by reading the values stored in the venues table, and then creating objects of type Venue in the showVenuesFromLocal method.

```
- (void) showVenuesFromLocal
{
    categories = [[NSMutableArray alloc] init];
    if(sqlite3_open([sqliteFileName UTF8String],
        &database)==SQLITE_OK)
    {
        NSString *selectStatement = [[NSString alloc]
            initWithFormat:@"SELECT * from %@",venuesTableName];
        sqlite3_stmt *sqlStatement;

        if(sqlite3_prepare_v2(database, [selectStatement UTF8String],
            -1, &sqlStatement, NULL)==SQLITE_OK)
        {
            while(sqlite3_step(sqlStatement)==SQLITE_ROW)
            {
                myVenue = [[Venue alloc] init];
                myVenue.id= [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 0)];

                myVenue.name = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 1)];

                myVenue.address = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 4)];

                myVenue.city = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 5)];

                myVenue.country = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 7)];

                myVenue.zip = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 6)];

                myVenue.lat = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 8)];

                myVenue.lon = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 9)];
            }
        }
    }
}
```

```

myVenue.categoryId = [NSString stringWithUTF8String:
    (char *)sqlite3_column_text(sqlStatement, 11)];

myVenue.phone = [NSString stringWithUTF8String:
    (char *)sqlite3_column_text(sqlStatement, 15)];

myVenue.checkins = (int)[NSString stringWithUTF8String:
    (char *)sqlite3_column_text(sqlStatement, 12)];

myVenue.usersCount = (int)[NSString stringWithUTF8String:
    (char *)sqlite3_column_text(sqlStatement, 13)];

if (![venues containsObject:myVenue])
    {
        [venues addObject:myVenue];
    }
}

} // end of if of sqlite3 open
[loadingIcon stopAnimating];
[myVenueTableView reloadData];
}

```

- 14.** The `venues` array does not only hold the title of the venues, but the full venue hierarchy defined in the `Venue` class. The `cellForRowAtIndexPaths` uses the `Venue *myVenue;` variable to fetch the venue information, and passes it to the `UITableView` for display.
- 15.** To summarize the **Home** screen of the app, first initialize the database, then get the `foursquare` categories, render it (hence, the two prototype cells in the Interface Builder), then call the `foursquare` `nearby venues` method (through the location manager's `didUpdateToLocation` method), and display the nearby venues.

```
[self initializeDatabase];
[loadingIcon startAnimating];
[self getfoursquareCategories];
```

- 16.** Now, when the user selects a venue from the `UITableView`, we fire the following code to call the next view from the `Storyboard`, venue details.

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath
:(NSIndexPath *)indexPath
{
    venueDetailsView = [self.storyboard
    instantiateViewControllerWithIdentifier:@"Venue Details"];
}
```

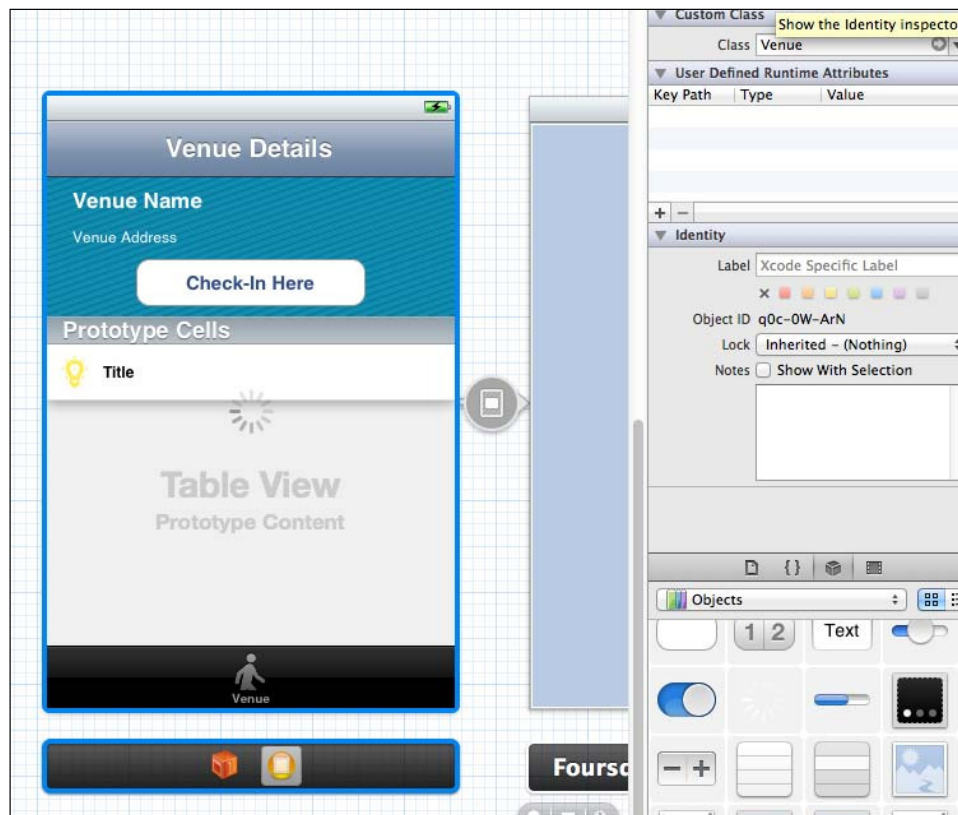
```

myVenue                                =    [venues
objectAtIndex:indexPath.row];
venueDetailsView.id                    =    myVenue.id;
venueDetailsView.name                  =    myVenue.name;
venueDetailsView.address               =    myVenue.address;
venueDetailsView.city                 =    myVenue.city;
venueDetailsView.country              =    myVenue.country;

[self.navigationController pushViewController:venueDetailsView
animated:YES];
[myVenueTableView deselectRowAtIndexPath:indexPath
animated:YES];
}

```

- 17.** Do not forget to change the class in **Interface Builder | Identity Inspector | Class** to **Venue**. This binds the `UIViewController` in Interface Builder to the `Venue` class that we added through Xcode.



18. Now, coming back to our venue class, it handles the check-in functionality, gets the individual venue information from foursquare venue API call, and renders the venue tips on the UITableView.

19. The venue information is retrieved through the following code in the viewDidLoad method:

```
NSString *url = [NSString stringWithFormat:
@"https://api.foursquare.com/v2/venues/%@?client_id=%@&client_
secret=%@&v=%@", id, CLIENT_ID, CLIENT_SECRET, version];

NSURL *urlToRequest = [[NSURL alloc] initWithString:url];
NSURLRequest *request = [NSURLRequest
                        requestWithURL:urlToRequest];
```

20. Note the id parameter, which is initialized in the code at *point 16*.

```
venueDetailsView.id = myVenue.id;
```

21. We parse the venue JSON payload through the connectionDidFinishLoading method, and render the tips on the UITableView.

22. The venue check-in is handled by the checkin IBAction, which is connected to the UIButton (with text Check-in Here) in Interface Builder.

```
- (IBAction)checkin:(id) sender
{
    inCheckin = TRUE;
    NSString *url = [NSString stringWithFormat:
@"https://api.foursquare.com/v2/checkins/add?
venueId=%@&client_id=%@&client_secret=%@&oauth_token=%@", id,
CLIENT_ID, CLIENT_SECRET, accessToken];

    NSURL *urlToRequest = [[NSURL
                            alloc] initWithString:url];
    NSMutableURLRequest *request = [NSMutableURLRequest
                                    requestWithURL:urlToRequest];
    [request setHTTPMethod:@"POST"];

    venueNameLabel.text = name;
    venueAddressLabel.text = address;

    urlConnection = [[NSURLConnection alloc]
                    initWithRequest:request
                    delegate:self startImmediately:YES];
}
```

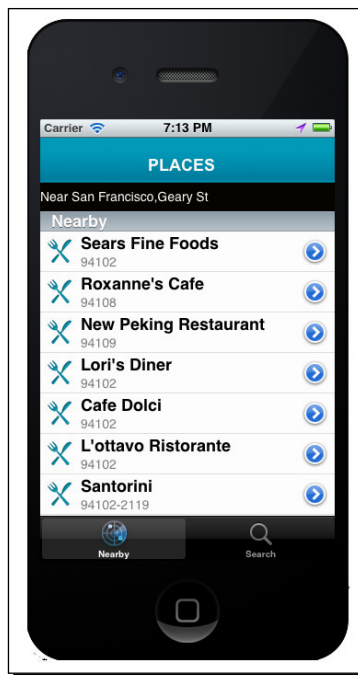
- 23.** Note that in the previous above, we need the `accessToken`, which is set only after a successful `foursquare` user authentication. This, as we mentioned before, is handled by the `foursquareAuth` class that we added before, which loads a `UIWebView`, and passes on the control to `foursquare`, to do the user authentication. On success of the user authentication, we retrieve the `accessToken`, and also store it in the app's **Settings** page, using the key `access_token`. Once the `access_token` is set in the app's **Settings** page, we can use it in any code within our app.

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

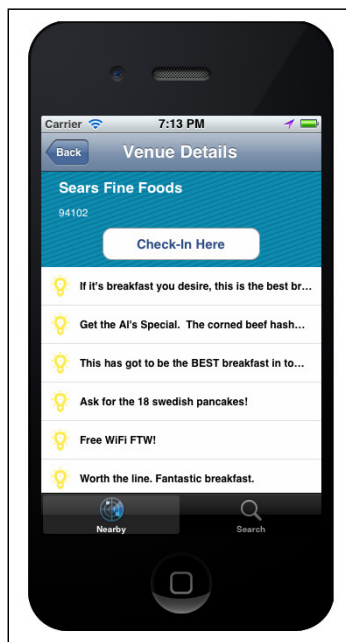
    NSString *authenticateURLString = [[NSString alloc]
initWithFormat:@"https://foursquare.com/oauth2/authenticate?
client_id=%@&response_type=token&redirect_uri=%@", CLIENT_ID,
REDIRECT_URL];

    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL
URLWithString:authenticateURLString]];
    [webView loadRequest:request];
}

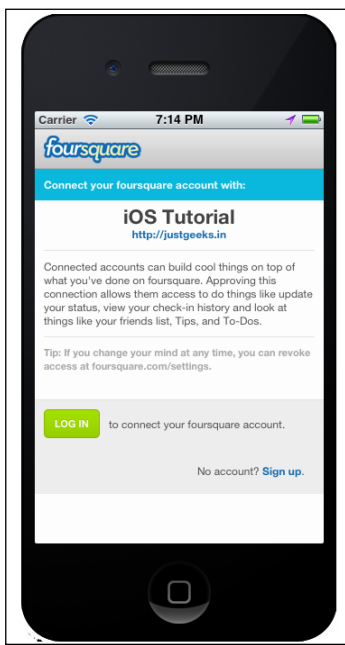
- (void)webViewDidFinishLoad:(UIWebView *)webView {
    NSString *URLString = [[self.webView.request URL]
absoluteString];
    if ([URLString rangeOfString:@"access_token="].location !=
        NSNotFound)
    {
        accessToken = [[URLString componentsSeparatedByString:@"="]
lastObject];
        NSUserDefaults *defaults = [NSUserDefaults
standardUserDefaults];
        [defaults setObject:accessToken forKey:@"access_token"];
        [defaults synchronize];
        [self dismissModalViewControllerAnimated:YES];
    }
}
```

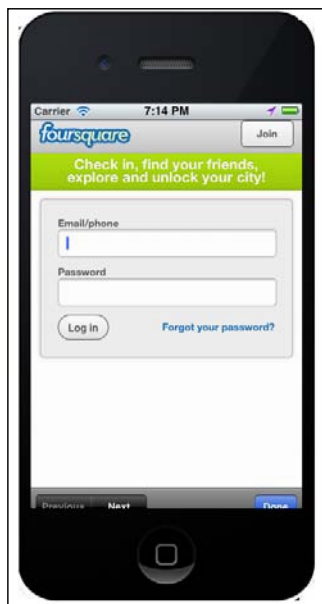
27. Click on the venue name to see the venue details and tips!



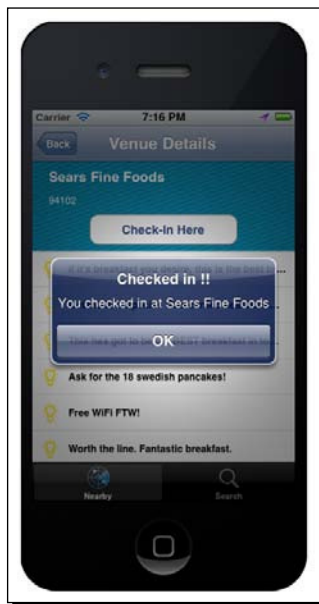
- 28.** Clicking on the **Check-in Here** button will lead you to the foursquare authentication URL as shown in the following screenshot:



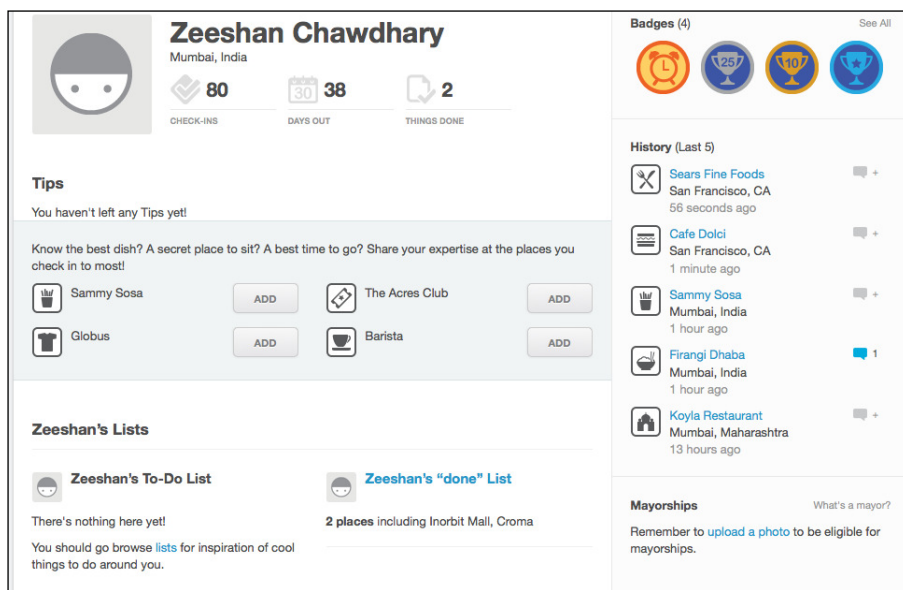
- 29.** Sign-in with your foursquare username and password, and let the check-ins begin.



- 30.** Once you have signed-in with your foursquare username and password, you should get the following screenshot, confirming your check-in at the venue:



- 31.** Log on to <http://www.foursquare.com> and visit your profile page to view your check-ins.



- 32.** Note the last screen from the author's profile page on <http://www.foursquare.com>; the check-ins are visible almost immediately.



Don't forget to check for NULL values while retrieving column values in SQLite. If the columns are null and haven't been thought of in the code, the app will crash. Use the following code to check each suspectable NULL column:

```
if (sqlite3_column_text (sqlStatement, 13) != NULL)
```

What just happened?

The navigation controller provides an easier mechanism for providing application flow logic, and also passing variables values from one View to another. We saw the usage of a `WebView` to perform third-party authentication over the regular browser, and retrieving the `Auth` token to be used in our app.

`foursquare` API provides a comprehensive API. We have used the popular ones to build `PacktLocal`, the source code for which can be found on the book's website: project titled `PacktLocal`. Feel free to extend the same, and add other features to the app. We have left the integration of the **Search** page to the readers.

Have a go hero - implement the add venue endpoint

We have showed you how to implement authentication as well as to consume the other endpoints. Use your learning to implement the add venue endpoint in the app. See <https://developer.foursquare.com/docs/venues/add.html>. Use the `Venue` class to build the new `Venue` object, and pass it over to the `foursquare` API.

Summary

In this chapter, we learned how to build a local search app – `PacktLocal`, by using the `foursquare` API. We also saw how to re-use the design from another open source project, and implement an almost similar UI in our app.

Specifically, we covered:

- ◆ Consuming the `foursquare` venue API
- ◆ Performing `foursquare` authentication
- ◆ Building the `PacktLocal` app

In the next chapter, we will look at building a news application with **AOL's Patch News API**.

9

Location Aware News—PacktNews

HyperLocal applications and websites, such as AOL's Patch provide precise and accurate news and information up to the neighborhood-level for a user. We will learn how to build a Hyper Local app - PacktNews using the Patch News API; however before you do that, you need to register for a key at <http://developers.patch.com/>.

In this chapter, we will deal with the following topics:

- ◆ Understanding the Patch News API – HyperLocal News
- ◆ Consuming the Patch News API
- ◆ Adding Geo-fencing support
- ◆ Building our app – PacktNews

So let's get on with it...

Understanding the Patch News API – HyperLocal News

AOL's **Patch.com** is a HyperLocal News portal that provides a comprehensive and trusted local content to the users. Patch has a huge editorial team (1000+ editors) that creates original content, which is published to a network of more than 850+ sites. Patch covers local content including news, events, business listing, photos, videos, and announcements. AOL merged **Outside.in** – a news aggregating service with AOL's Patch in March 2011, creating a unique combination of HyperLocal content, with **News** being the core product.

Patch.com provides a Developer API at <http://developers.patch.com/>, which we will use to build our own News app – PacktNews. Before we build the app, we will have a detailed look at the API calls provided by Patch.com. The Patch News API has the following four main components:

- ◆ Authentication
- ◆ Taxonomy (categories)
- ◆ Finding stories by location
- ◆ Finding locations by name

Authentication

Any app or website that intends to use Patch News API must obtain authentication. The authentication is a combination of your developer key, secret key, and the current TimeStamp, combined together and converted to a MD5 hash key. If you have coded in PHP before or still do, you can find the following script to generate the encrypted API URL for finding nearby stories around San Francisco.

```
<?php
$time    = time();
$key     = "xxxxxxxxxxxx";
$secret  = "xxxxxxxxxxxx";
$sig     = md5($key.$secret.$time);
$url     = "http://news-api.patch.com/v1.1/nearby/37.785368,-
          122.441654/stories?dev_key=$key&sig=$sig";
?>
```

Encryption methods for other languages, including Objective-C (which we will use in our app) can be found at <https://github.com/outsidein/api-examples>.

Taxonomy

Patch organizes the stories into three main taxonomy types:

Vertical: Topic of the content

Format: Medium from which the content was found

Author: Who wrote the content - individual or business organizations

These taxonomies are further classified as

Vertical

- ◆ News
- ◆ Lifestyle
- ◆ Education
- ◆ Business
- ◆ Science and technology
- ◆ Sports

Format

- ◆ Stories
- ◆ Reviews and ratings
- ◆ Event listings

Author

- ◆ Individuals
- ◆ Businesses and organizations
- ◆ Educational institutions
- ◆ Government
- ◆ Sharing and community sites
- ◆ Independent news media
- ◆ Mainstream media, such as cnn, nytimes

Finding stories by location

The Patch News API supports location-based search for stories, by using any of the following:

- ◆ State
- ◆ City
- ◆ Zip code
- ◆ Neighborhood
- ◆ Nearby
- ◆ Patch location `UUID` (Patch.com internal city/ state and neighborhood IDs), which can be retrieved by the `Find Locations by Name` method, described in the next section

Find location by names

Patch.com News API supports a location retrieval API call that accepts a text string, and returns the well-formatted location information, something similar to the Geonames API or reverse geocoding that we saw before.

A comprehensive documentation of all the available Patch News API methods and return values can be found at <http://developers.patch.com/docs/>.



The Patch News API supports **Cross-Origin Resource Sharing Requests (CORS)**. CORS is the new W3C proposed mechanism for cross-site HTTP requests. Read more about CORS at <http://www.w3.org/TR/cors/>.

Time for action – consuming the Patch News API

Having looked at the working of the Patch News API, let us fire some code to consume the News services by user's location. We will use the location manager for managing the user's location as before, SQLite for storing the taxonomy and news for offline usability, and UITableView for the display.

As in the case of the events app, we will create the database structure for the following:

- ◆ Storing the latest news entries
- ◆ Storing the category or the taxonomy in the case of Patch News API
- ◆ Storing the user's last known location

1. So, we create the news table in a database named packtnews.sqlite, as follows:

```
CREATE TABLE IF NOT EXISTS "news" ("uuid" VARCHAR UNIQUE ,  
"title" VARCHAR, "summary" TEXT, "story_url" VARCHAR, "feed_title"  
VARCHAR, "tags" TEXT, "source_verticals" TEXT, "source_formats"  
TEXT, "source_author_types" VARCHAR, "location_lat" DOUBLE,  
"location_lon" DOUBLE, "published_at" VARCHAR, "feed_url" VARCHAR)
```

2. Next, we create the taxonomy table. Since there is no API call to read and parse the taxonomy data, we will create the taxonomy table, and also prefill it with the values.

Taxonomy type	Category	Sub category
Vertical	news	national
		local
		crime
		politics-and-political-analysis

Taxonomy type	Category	Sub category
		opinion
	lifestyle	activism
		arts-and-entertainment
		crafts
		fashion
		food-and-restaurants
		nightlife
		shopping
		real-estate
		health
		travel
		recreation
		parenting-family-and-children
		personal
		religion
		community
	education	colleges-and-universities
		high-schools
		libraries
	business	finance
		marketing
		small-business
		advertising
		business-promotion
	science-and-technology	
	sports	
Format	stories	blog-posts
		news-articles
		press-releases
	reviews-and-ratings	
	event-listings	
Author type	mainstream-media	
	independent-new-media	
	sharing-and-community-sites	

Taxonomy type	Category	Sub category
	business-and-organizations	corporations
		small-businesses
		real-estate-agents-and-brokers
		non-profit-and-not-for-profit-organizations
		sports-teams
		religious-institutions
		political-parties
	individuals	general
		celebrities
	educational-institutions	colleges-and-universities
		high-schools
		libraries
	government	

The SQL for this taxonomy table is

```
CREATE TABLE "taxonomy" ("type" VARCHAR, "category" VARCHAR,
"subcategory" VARCHAR)
```

3. We populate the taxonomy table with the data from the taxonomy table as follows:

```
INSERT INTO "taxonomy" VALUES('vertical','news','national');
INSERT INTO "taxonomy" VALUES('vertical','news','local');
INSERT INTO "taxonomy" VALUES('vertical','news','world');
INSERT INTO "taxonomy" VALUES('vertical','news','crime');
INSERT INTO "taxonomy" VALUES('vertical','news','politics-and-
political-analysis');
INSERT INTO "taxonomy" VALUES('format','stories','blog-posts');
INSERT INTO "taxonomy" VALUES('format','stories','news-articles');
INSERT INTO "taxonomy" VALUES('format','stories','press-
releases');

INSERT INTO "taxonomy" VALUES('author','mainstream-media','');
INSERT INTO "taxonomy" VALUES('author','independent-new-
media','');
INSERT INTO "taxonomy" VALUES('author','sharing-and-community-
sites','');
```

4. Finally, we will create the user position table as we did in the `PacktEvents` app as follows:


```
CREATE TABLE IF NOT EXISTS "user_position" ("position_id"
INTEGER PRIMARY KEY, "latitude" DOUBLE, "longitude" DOUBLE, "city"
VARCHAR, "country" VARCHAR)
```
5. Create a new Single View Application project in Xcode and name it `Hello News`. Also, add the class prefix as `Hello News`, so that our View controllers are named appropriately.
6. Add the SQLite3 library to your project from the **Target-| Build Phases | Link Binary with Libraries** tab option in your Xcode project workflow. Also, add the Core Location framework.
7. The Patch API needs authentication (as discussed before), which is a combination of your developer key, secret key, and the current `TimeStamp`, combined together, and converted to a MD5 hash key. We have seen the PHP code before; however, for our application, we need the Objective-C code. Thankfully, AOL provides the libraries to do the same. Visit <http://developers.patch.com> to get the library, available for a couple of languages.
8. We need the `MD5.h` and `MD5.m` files. Drag these two files, and add it to your project in Xcode.
9. Our `Hello News` example will simply hit the Patch API with the user's location, get the news (in JSON format), and render it on a `UITableView`. This is similar to the examples that we have seen for `Last.fm` (`Hello Location - Last.fm` from *Chapter 3, Using Location in your iOS Apps – Core Location*) or `EventFul.com` (`Hello Location - Eventful` from *Chapter 3*) before. The only addition we now have is that we are storing the news offline as well.
10. In our `Hello_NewsViewController.h` file, we declare the necessary variables and functions as follows:

```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>
#import <sqlite3.h>

@interface Hello_NewsViewController : UIViewController
<CLLocationManagerDelegate>
{
    CLLocationManager    *locationManager;
    CLLocation           *userLocation;
    NSURLConnection     *urlConnection;
    sqlite3              *database;
    NSString             *sqliteFileName;
```

```
UITableView      *myTableView;
NSMutableDictionary *jsonContent;
NSJSONSerialization *nsJson;
}

@property(retain, nonatomic) IBOutlet UITableView *myTableView;

- (NSString *) getDatabaseFullPath;
- (NSString *) initializeDatabase;

- (void) readNews;
- (void) readNewsFromLocal;

@end
```

11. Our `Hello_NewsViewController.m` file is where most of the action is. We begin by including the `MD5.h` file, and declaring the variables for the `news`, the `user`, and the `taxonomy` tables that we defined earlier.

12. In our `viewDidLoad` method, we initialize a variable `news` that will hold the ten nearest News items' titles in an array. We then call the `initializeDatabase` method that creates the required database tables for `news`, `user`, and the `taxonomy` table. Then we can call the `readNews` method that reads the Patch News API, and stores the news values in the database through the `JSON` `connectionDidFinishLoading` method. Finally, we read the values in the database, and render it to a `UITableView`.

```
news            = [[NSMutableArray alloc] init];
sqliteFileName  = [self getDatabaseFullPath];
jsonContent     = [[NSMutableDictionary alloc] init];
[self initializeDatabase];
[self readNews];
[self readNewsFromLocal];
```

13. The `initializeDatabase` method is something new here. We have the `taxonomy` information from the Patch News API, so it does make sense to hit the API again and insert it in the database, rather we just insert it with the values we know. This helps in saving a precious HTTP call through a user's mobile device.

```
- (NSString *) initializeDatabase
{
    NSString *success = @"FALSE";
    NSString *taxonomy_insert_sql = @"
INSERT INTO taxonomy VALUES('vertical','news','national');

INSERT INTO taxonomy VALUES('format','stories','blog-posts');
```

```

INSERT INTO taxonomy VALUES('author','mainstream-media', NULL);

// Please see full source code on the books page at packtpub.com

news_sql    = @"CREATE TABLE IF NOT EXISTS 'news' ('uuid'
VARCHAR UNIQUE , 'title' VARCHAR, 'summary' TEXT, 'story_url'
VARCHAR, 'feed_title' VARCHAR, 'tags' TEXT, 'source_verticals'
TEXT, 'source_formats' TEXT, 'source_author_types' VARCHAR,
'location_lat' DOUBLE, 'location_lon' DOUBLE, 'published_at'
VARCHAR, 'feed_url' VARCHAR)";

taxonomy_sql    = @"CREATE TABLE 'taxonomy' ('type' VARCHAR,
'category' VARCHAR, 'subcategory' VARCHAR)";

user_sql        = @"CREATE TABLE IF NOT EXISTS 'user_position'
('position_id' INTEGER PRIMARY KEY, 'latitude' DOUBLE, 'longitude'
DOUBLE, 'city' VARCHAR, 'country' VARCHAR)";

if(sqlite3_open([sqliteFileName UTF8String],
&database)==SQLITE_OK)
{
    if(sqlite3_exec(database, [news_sql UTF8String], NULL, NULL,
&sqliteError)==SQLITE_OK)
    {
        //do something or echo
    }

    if(sqlite3_exec(database, [taxonomy_sql UTF8String], NULL,
NULL, &sqliteError)==SQLITE_OK)
    {
        //do something or echo
    }

    if(sqlite3_exec(database, [user_sql UTF8String], NULL, NULL,
&sqliteError)==SQLITE_OK)
    {
        NSLog(@"user table created");
    }

    if(sqlite3_open([sqliteFileName UTF8String],
&database)==SQLITE_OK)
    {
        if(sqlite3_exec(database, [@"Delete from taxonomy"
UTF8String], NULL, NULL, &sqliteError)==SQLITE_OK)

```

```
        {
            NSLog(@"taxonomy Purged");
        }

        if(sqlite3_exec(database, [taxonomy_insert_sql
            UTF8String], NULL, NULL, &sqliteError)==SQLITE_OK)
        {
            NSLog(@"taxonomy Inserted");
        }
    }
    success=@"TRUE";
}

return success;
}
```

- 14.** The `readNews` method generates the MD5 signature needed by the Patch API, using the `md5hex` method defined in the `MD5.h` file. This MD5 signature is then passed to the `NSURLConnection` object as follows:

```
-(void) readNews
{
    NSString *appKey    = @"xxxxxxx"; // Get your own key from
                                   developer.patch.com
    NSString *secret    = @"xxxxxxx";
    long time= (long) [[NSDate date]
        timeIntervalSince1970];
    NSString *signature=[MD5 md5hex:[NSString
        stringWithFormat:@"%@@@%", appKey, secret, time]];
    NSString *userLat=[[NSString alloc]
        initWithFormat:@"%g",userLocation.coordinate.latitude];
    NSString *userLon=[[NSString alloc]
        initWithFormat:@"%g",userLocation.coordinate.longitude];

    NSString *url    = [NSString stringWithFormat:@"http://news-api.
        patch.com/v1.1/nearby/%@,%@/stories?dev_key=%@&sig=%@&radius=5000&
        include-locations=true",userLat,userLon,appKey,signature];

    NSURL      *urlToRequest=[[NSURL alloc] initWithString:url];
    NSURLRequest *request=[NSURLRequest requestWithURL:urlToRequest];

    urlConnection =[[NSURLConnection alloc] initWithRequest:request
        delegate:self startImmediately:YES];
}
```

- 15.** Once the JSON data is received with the `connectionDidFinishLoading` method, we parse the individual news items and parent JSON tag as stories. The insert statement for a new story is as follows:

```
insertStatement = [[NSString alloc] initWithFormat:@"INSERT OR
REPLACE INTO
'%@'('%@','%@','%@','%@','%@','%@','%@','%@','%@','%@','%@','%@','%@','
'%@')
VALUES('%@','%@','%@','%@','%@','%@','%@','%@','%@','%@','%@','%@','
'%@','%@')",newsTableName,@"uuid",@"title",@"summary",
@"story_url",@"feed_title",@"tags",@"source_verticals",
@"source_formats",@"source_author_types",@"location_lat",
@"location_lon",@"published_at",@"feed_url",news_uuid,news_title,
news_summary,new_story_url,news_feed_title,news_tags,
news_source_verticals,news_source_formats,
news_source_author_types,news_latitude,news_longitude,
news_published_at,news_feed_url];
```

- 16.** Finally, we read from the Local SQLite database table, and render it on the UITableView as follows:

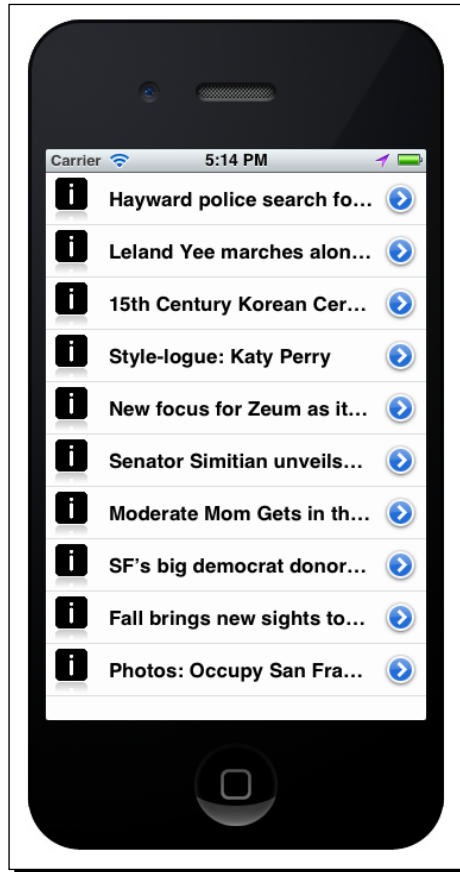
```
-(void) readNewsFromLocal
{
    if(sqlite3_open([sqliteFileName UTF8String],
        &database)==SQLITE_OK)
    {
        selectStatement = [[NSString alloc] initWithFormat:
            @"SELECT * from %@ order by uuid desc",newsTableName];

        sqlite3_stmt *sqlStatement;

        if(sqlite3_prepare_v2(database, [selectStatement
            UTF8String],
            -1, &sqlStatement, NULL)==SQLITE_OK)
        {
            while(sqlite3_step(sqlStatement)==SQLITE_ROW)
            {
                NSString *titleDataText = [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 1)];

                if(![news containsObject:titleDataText])
                {
                    [news addObject:titleDataText];
                }
            }
        }
        // end of if of sqlite3 open
        [myTableView reloadData];
    }
}
```


17. Running the example produces the following result:

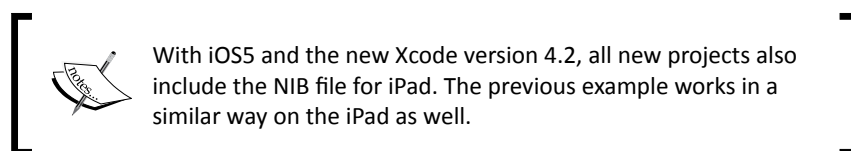


Note that we added an **Information Image** icon to add some zeal. Download the full sample from the book's website: project titled `Hello News`.

What just happened?

We created a simple news example that reads a location-based news API from AOL's Patch.com, and stores the top ten news stories in an SQLite database, which can then be re-used anywhere in our application.

We also saw how to hash our API call with the key and timestamp values, using the MD5 library from the Patch News API. This ensures that the API returns us the latest news entries, based on the time we make the call. This makes fresh content available to the user, every time the app is used. Such smart use of technology makes an app appealing to the end users, and ensures good sales and/or gets us the user's love.



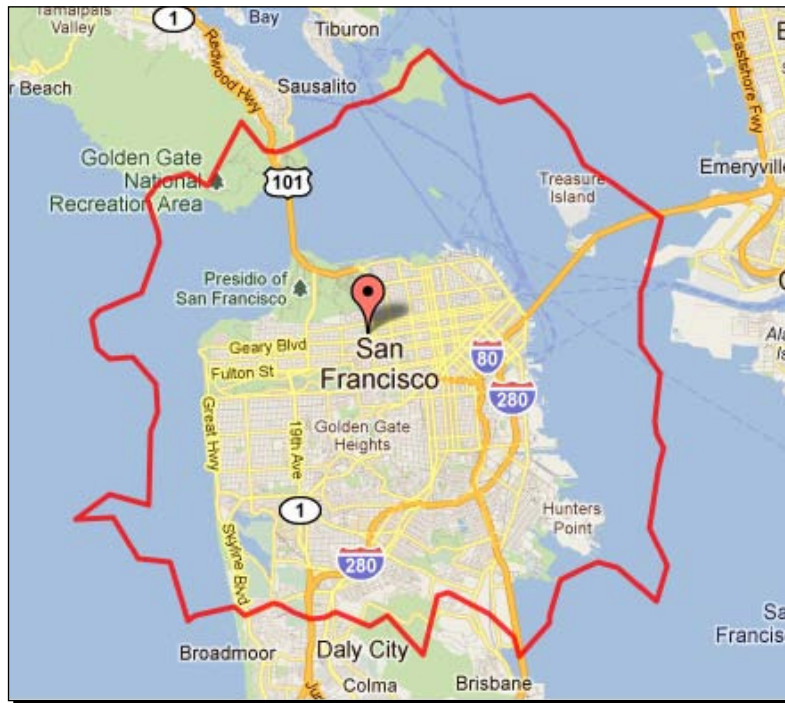
Adding the Geo Fencing support

We have discussed Geo Fencing a couple of times before, as well as we did a small example in *Chapter 3*. Remember the Hello Location – Boundary Monitoring example, where we define a circular region for San Francisco. We use the same logic here, but this time, we create a couple of regions for San Francisco, New York, and Mumbai.

Although there are couple of APIs, such as Location Labs - <https://geofence.locationlabs.com> and GeoLoqi - <https://developers.geoloqi.com/geofencing-api>, which help integrate Geo Fencing easily into your iOS apps, technically they are no different than the functionality provided by `CLRegion`, since the concept of Geo Fencing in each of these approaches is the same; every geo-fencing approach needs a latitude/longitude pair and the radius to monitor.

The best and accurate method of Geo Fencing is more GIS inclined, using a solution such as PostGIS, where the exact physical boundary of a region (city/metro/state/country) can be defined using a polygon, rather than just latitude/longitude pairs, and the user's position matched against this polygon boundary.

The following screenshot shows the physical boundary of the city of San Francisco, while the marker in the center could be the user's position. As the marker moves to a different location, its position can be checked to determine whether it is within the boundary or outside the boundary. This can be easily accomplished with the new Geography data type in PostGIS. For more information, visit - <http://postgis.refrations.net/>.



Time for action – adding the Geo Fencing support

Coming back to the example, we extend the Hello News example, and add three CLRegions for monitoring in the same.

1. Open the `Hello_NewsViewController.m` file. In the `viewDidLoad` method, we create the three regions with a boundary of 5000 meters as follows:

```
CLLocationCoordinate2D regionCords =  
    CLLocationCoordinate2DMake(37.33 , -122.03);  
  
CLRegion *sanFranciscoBoundary =  
    [[CLRegion alloc] initWithCenter:regionCords  
        radius:5000  
        identifier:@"San Francisco"];
```

```

regionCords=CLLocationCoordinate2DMake(40.71490, -74.00679);

CLRegion *newYorkBoundary =
[[CLRegion alloc] initWithCenter:regionCords
radius:5000
identifier:@"New York"];

regionCords=CLLocationCoordinate2DMake(19.142472, 72.841198);

CLRegion *mumbaiBoundary=
[[CLRegion alloc] initWithCenter:regionCords
radius:5000
identifier:@"Mumbai"];

```

2. We then tell the location Manager to start monitoring for these regions.

```

[locationManager startUpdatingLocation];
[locationManager
startMonitoringForRegion:sanFranciscoBoundary];
[locationManager startMonitoringForRegion:newYorkBoundary];
[locationManager startMonitoringForRegion:mumbaiBoundary];

```

3. As the user enters or exits any of these regions, a corresponding message is alerted to the user, using the `didEnterRegion` and `didExitRegion` delegate methods.

```

- (void) locationManager:(CLLocationManager *)manager
didEnterRegion:(CLRegion *)region
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:
    [[NSString alloc] initWithFormat:
        @"You Entered %@", region.identifier]
    message:[ [NSString alloc] initWithFormat:@"Welcome to
    %@", region.identifier]
    delegate:self cancelButtonTitle:@"OK"
    otherButtonTitles:nil, nil];

    [alert show];
}

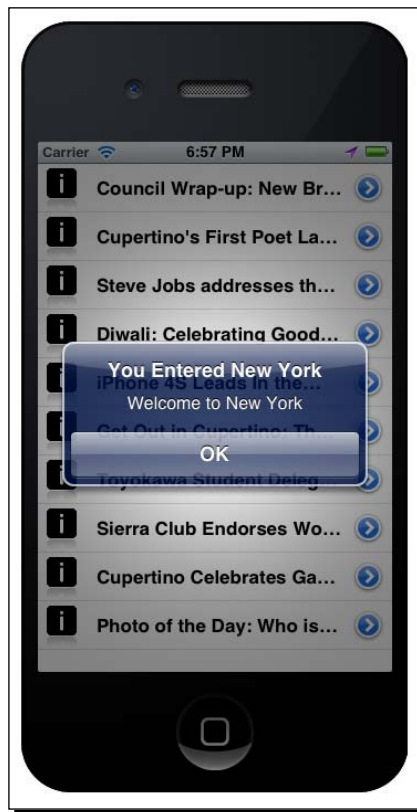
- (void) locationManager:(CLLocationManager *)manager
didExitRegion:(CLRegion *)region
{
    UIAlertView *alert = [[UIAlertView alloc]
    initWithTitle:[ [NSString alloc]
    initWithFormat:@"Thanks you for visiting
    %@", region.identifier]
    message:[ [NSString alloc] initWithFormat:
    @"Hope you come back to

```

```
        %@" ,region.identifier]
        delegate:self cancelButtonTitle:@"OK"
        otherButtonTitles:nil, nil];

        [alert show];
    }
}
```

4. You can use any application logic in these functions to do some smart processing when a user enters or exits a city, maybe purged to the local database?
5. Running the example produces the following result, when we change the location through the iOS simulator's **Location Simulation** option:



What just happened?

We added Geo Fences in our `Hello News` example to monitor three cities for user's `enter` and `exit` trigger. Any number of Geo Fences can be added in our application. However, we should try to keep the alerts to a minimum, so that the users are not irked with too many pop-up alerts.

Also note, we used the `region.identifier` property to identify the city name, thereby making our code shorter and manageable. Code for this example is also available on the book's website: project titled `Hello News Geo Fencing`.

Building our app - PacktNews

Now that we have looked at the Patch News API briefly, it is time to assemble all our tricks that we learned so far, and to build the app.

For the app icon, we used a free-for-commercial use icon from <http://findicons.com/icon/169293/news?id=376465>.

We have also used the Google AdMob SDK as in the case of the `Weather` app, as well the option provided in the app to tweet a news story, using the Twitter Framework of iOS 5 (iPad only for now, due to design constraints). We have used the **Tabbed Application** template for our app, but it has been modified using the new **Storyboard** feature in iOS 5.

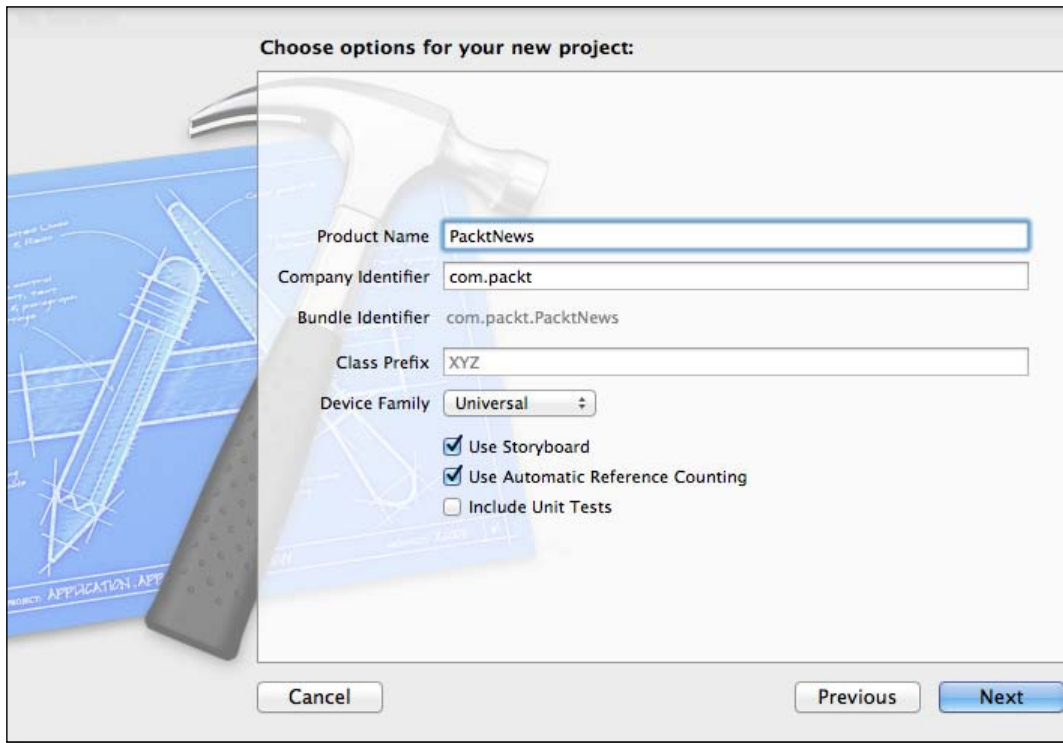
A bit on StoryBoard

As we discussed in *Chapter 2, The Xcoder's World*, **Storyboarding** is a new feature added in Interface Builder to manage the transitions between different views in your app. These transitions are called as **segues**. From a developer's point of view, you can compare a StoryBoard to an entity-relationship diagram from the database world.

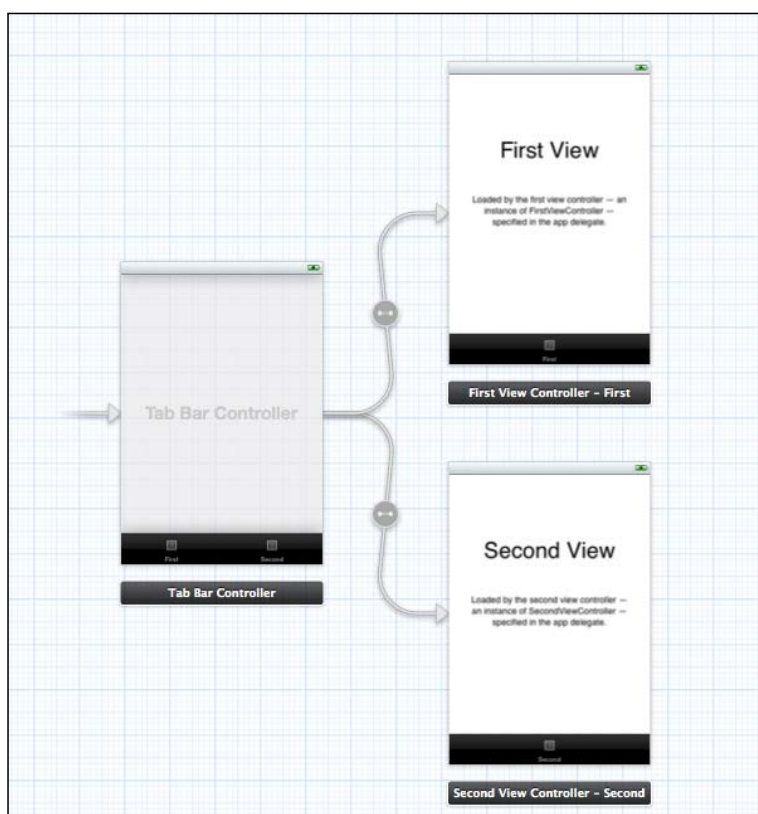
A StoryBoard comprises of a sequence of scenes (of the type `UIViewController`), and these scenes are connected by the segue objects.

Time for action – building PacktNews

1. We begin by creating a new project titled `PacktNews`, using the **Tabbed Application** template. From the project settings, make sure you select the **Use StoryBoard** checkbox



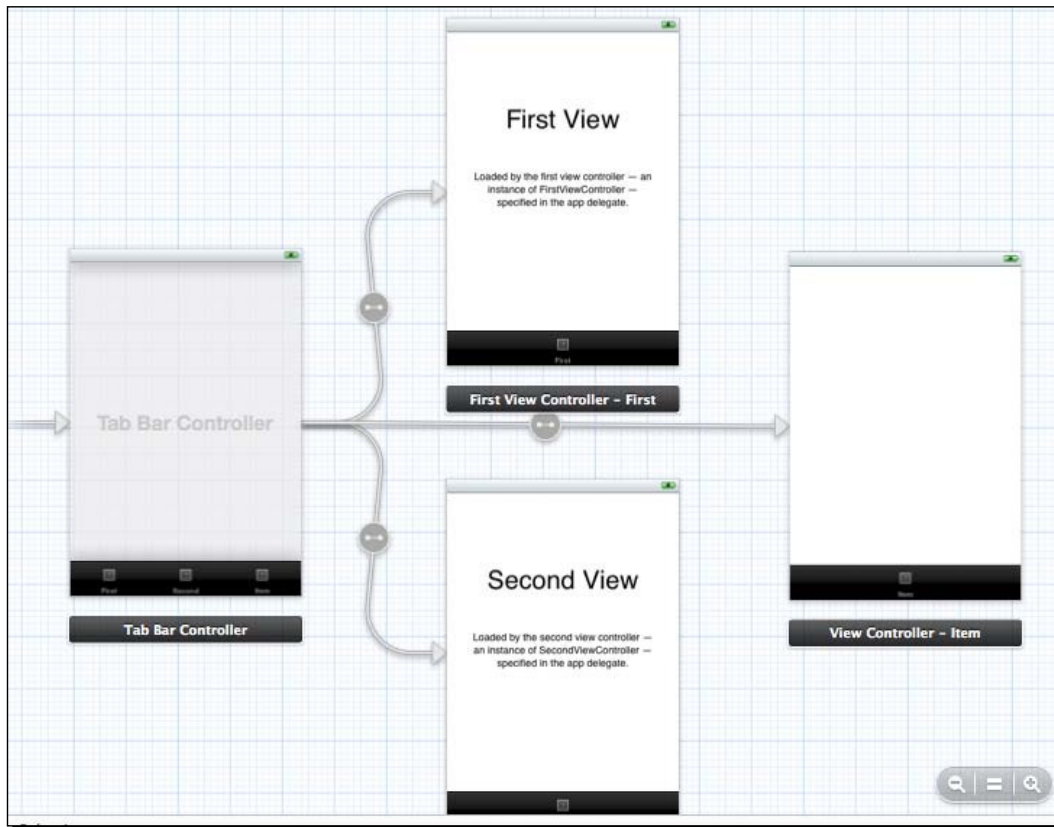
2. Once your project creation is complete, you will find two files in your project with a `.storyboard` extension, one for the iPhone and another for iPad. Files will be named as `MainStoryboard_iPhone.storyboard` and `MainStoryboard_iPad.storyboard`. Double-click on the iPhone storyboard, and you should see the following storyboard for our **Tabbed Application** template:



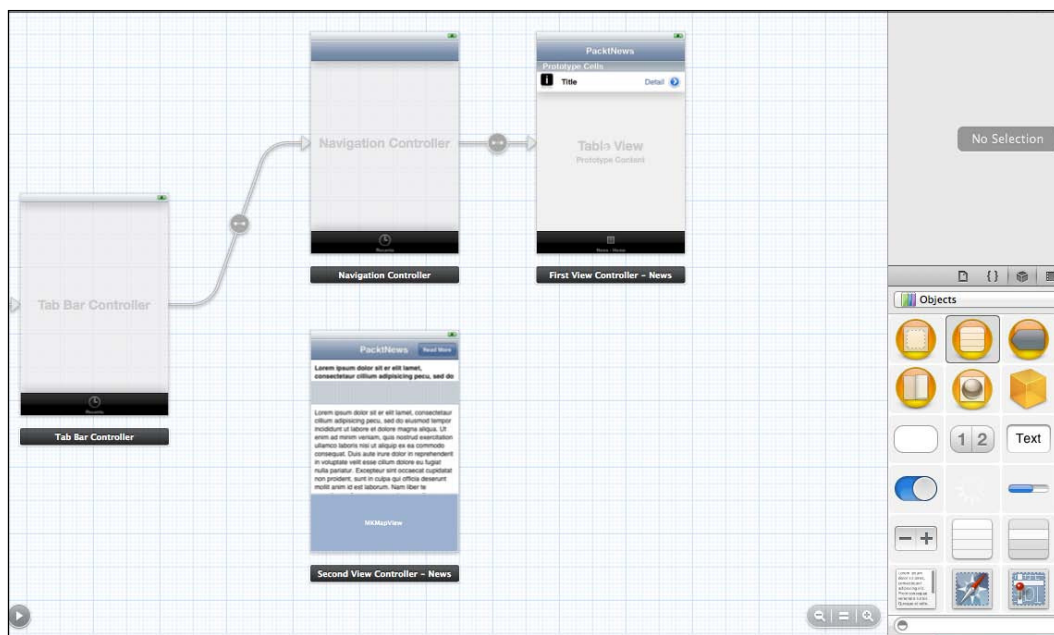
3. It works as follows: The main scene is your **Tab Bar Controller**, by default, it has two tab bar items in the footer, since the template contains two `UITableViewController`s. The arrows you see from the **Tab Bar Controller** to the **First View** and **Second View** controllers are segues.

4. From the Xcode utilities area, add one more `ViewController` to your project, **Control+drag** from your main screen to the third `ViewController`, and select **Relationship – viewControllers** from the menu option.

This will add one more tab in your footer area, automatically!!



5. Ok, so now we know how Storyboards works. This layout was just an example. The final layout for `PacktNews` is as follows:

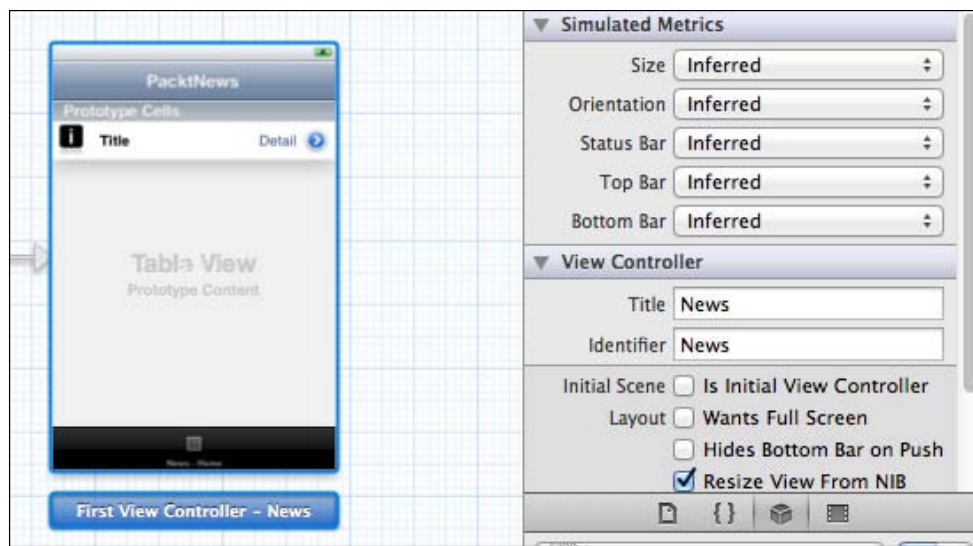


6. We added a **Navigation Controller** and a **Table View Controller**. The **First View Controller** is now the `UITableView` controller, which will be used to show up to twenty nearby news items. The **Second View Controller** is used to display the detailed news on selection from the `UITableView` from the **First View Controller**.
7. The **Navigation Controller** is the parent scene here. Note that we haven't connected the **Second View Controller** to the **Navigation Controller** (hence, only one tab bar item in the footer). We call the **Second View Controller** programmatically though the `didSelectRowAtIndexPath` method of the `UITableView` as follows:

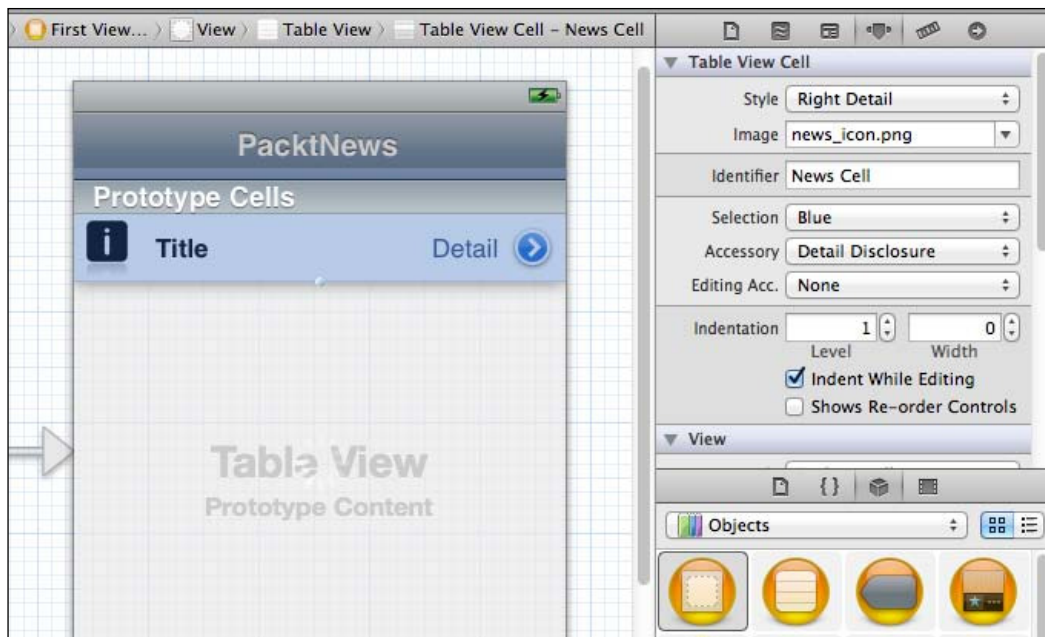

```
(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath
{
    detailNews = [self.storyboard
    instantiateViewControllerWithIdentifier:@"Details"];
    [self.navigationController pushViewController:detailNews
    animated:YES];

    [self getNewsDetails:[news objectAtIndex:indexPath.row]];
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

8. Do not forget to name your ViewControllers in **Interface Builder | Attributes Inspector**. We name the **First View Controller** as `News` and the **Second View Controller** as `Details`:



9. The `pushViewController` method of the **Navigation Controller** pushes the **Second View Controller** – `Details` View to the main scene. Note that the **Back** button is added automatically, which takes the user to the main screen, which is the **First View Controller**.
10. Coming to the design part, we mentioned the **First View Controller** as being a `TableView` controller, which adds a `Row` template to the `TableView`. We can style the `Row` using the **Attributes Inspector** again. In our case, we add a `news_icon` and a **Right Detail** attribute to each cell in the **Table**. The **Right Detail** attribute signifies that the row/cell contains more details, which can be obtained after clicking on the same. This is user interaction perspective to making your apps easier for end-users. We also select the accessory as **Detail Disclosure** with **Blue** as the selection. Try changing these attributes as you might see it fit.

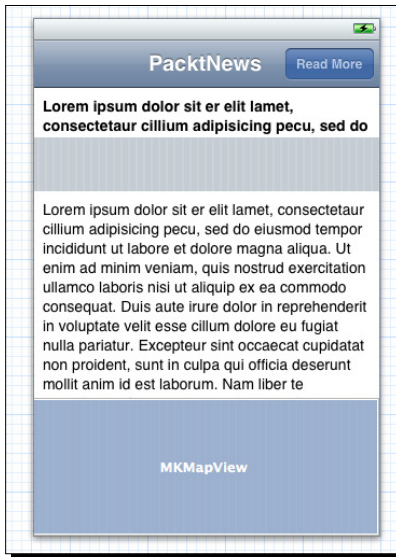


- 11.** In our **Details View Controller** (also known as the **Second View Controller**), we add a Bar Button to the navigation bar, named **Read More** that navigates the user to the external news URL, so that the users can read the original story from the parent source.
- 12.** We add two `UITextView` elements, one for the **News Title** and another for the **News Summary**. We could have used a **Text** field for the **News Title**, but sometimes the titles seem to be long, so a `UITextView` does more justice.

- 13.** We also add a `MapView` component, which shows the user position as well the location of the **News Story** on the map. From the **Attributes Inspector**, we enable the **Shows User Location** for the `MapView` component.



Our **Details** page is where we also show an Ad from the Admob framework, the placement of which is the blank space between the **News Title** and the **News Summary**. This can change as per the app design, however be careful to change this in the **Ad Integration** code as well.



- 14.** The iPad UI is more interesting, since it has ample amount of space to add much more features, such as the Twitter Integration and bigger Map View. The Twitter button is actually an `UIButton` styled with background image (`twitter.png` in the project). We connect this button with an `IBAction` named `sendTweet`, which will initiate the `TweetComposeViewController` modally, and pre-populate the tweet text as the **News Title**, and then link back to the News (from the Patch News API).

```
- (IBAction) sendTweet: (id) sender
{
    if ([TWTweetComposeViewController canSendTweet])
    {
        TWTweetComposeViewController *tweetViewController =
            [[TWTweetComposeViewController alloc] init];
        [tweetViewController addURL:[NSURL URLWithString:storyUrl]];
        [tweetViewController setInitialText:[NSString
            alloc] initWithFormat:@"%s @ #patchnews", titleView.text]];
        [tweetViewController
```

```

setCompletionHandler:^(TWTweetComposeViewControllerResult
result)
{
    NSString *tweetOutput;
    switch (result) {
        case TWTweetComposeViewControllerResultCancelled:
            // The cancel button was tapped.
            tweetOutput = @"The user cancelled the tweet. ";
            break;
        case TWTweetComposeViewControllerResultDone:
            // The tweet was sent.
            tweetOutput = @"You sent a tweet successfully";
            break;
        default:
            break;
    }
    [self dismissModalViewControllerAnimated:YES];
}];
[self presentModalViewController:tweetViewController
animated:YES];
} // end of if canSendTweet
}

```



- 15.** The code is pretty much the same as the Hello News and Hello News - Geofencing examples, with the addition of the StoryBoard and the Tweeting options. The `FirstViewController.h` and `FirstViewController.m` files from the project handle the initial new display with up to 20 news items in the `UITableView`. The new method that we have defined here is the `getNewsDetails:newsTitle;` method, which takes in the News Title as input, and queries the local database for all the related details, which are then passed on to the **Second View Controller** instance variable `detailNews`.
- 16.** We initialize the UI elements of the **Details** page with the new values obtained from the local SQLite database, for a selected news title as follows:

```
-(void) getNewsDetails:newsTitle
{
    if(sqlite3_open([sqliteFileName UTF8String],
        &database)==SQLITE_OK)
    {
        selectStatement = [[NSString alloc] initWithFormat:
            @"SELECT * from %@ where
            title=\"%@\\"",newsTableName,newsTitle];
        sqlite3_stmt *sqlStatement;

        if(sqlite3_prepare_v2(database, [selectStatement UTF8String],
            -1, &sqlStatement, NULL)==SQLITE_OK)
        {
            while(sqlite3_step(sqlStatement)==SQLITE_ROW)
            {
                NSString *titleDataText=[NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 1)];
                NSString *summaryDataText= [NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 2)];
                NSString *storyDataText=[NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 3)];
                NSString *latitudeDataText=[NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 9)];
                NSString *longitudeDataText=[NSString stringWithUTF8String:
                    (char *)sqlite3_column_text(sqlStatement, 10)];

                CLLocationCoordinate2D pinlocation =
                    detailNews.map.userLocation.coordinate;
                pinlocation.latitude = [latitudeDataText doubleValue];
                pinlocation.longitude = [longitudeDataText doubleValue];

                [detailNews.titleView setText:titleDataText];
                [detailNews.descView setText:summaryDataText];
                detailNews.storyUrl=storyDataText;
                detailNews.map.zoomEnabled = TRUE;
                detailNews.map.centerCoordinate = pinlocation;
            }
        }
    }
}
```

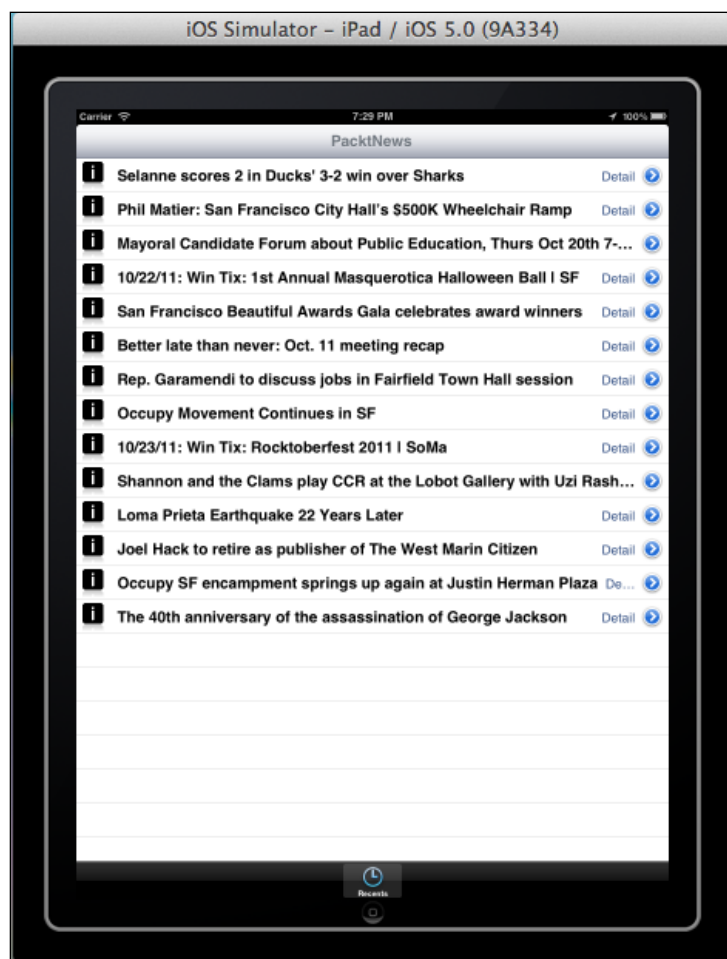
```

MKPointAnnotation *annotation = [[MKPointAnnotation alloc] init];
annotation.title           = titleDataText;
annotation.coordinate      = pinlocation;
[detailNews.map addAnnotation:annotation];

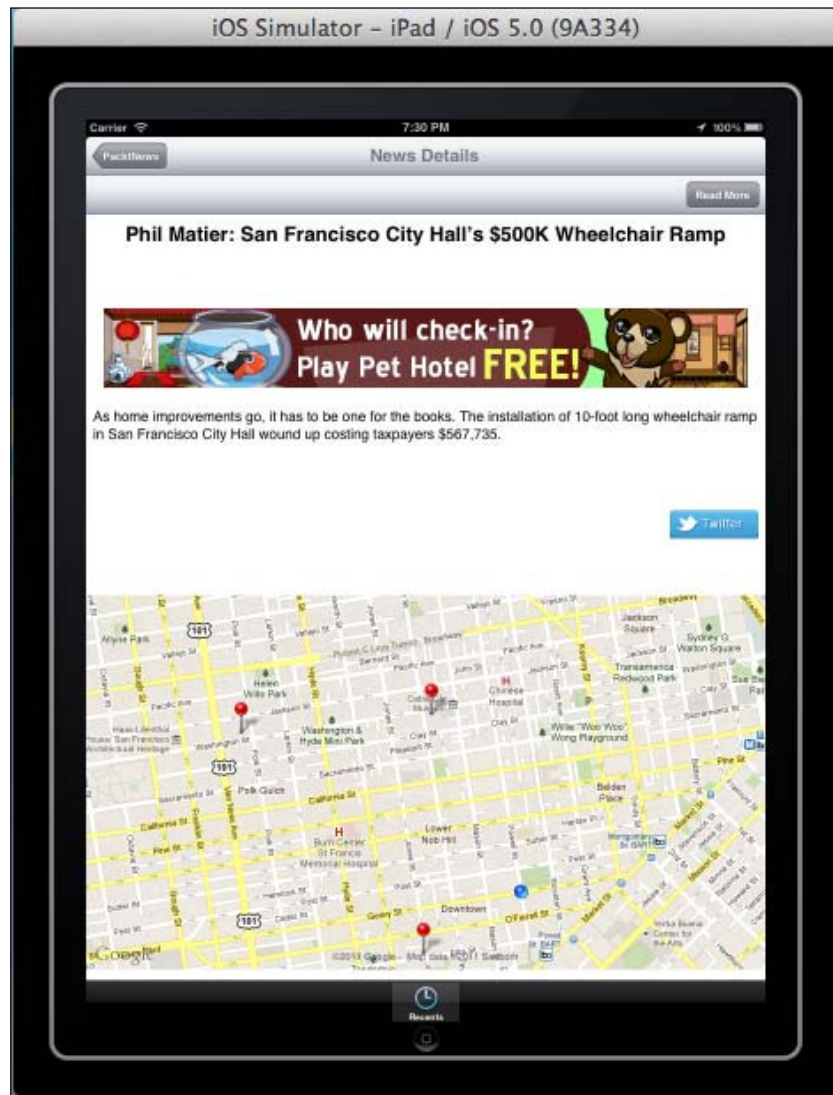
detailNews.map.region      =
MKCoordinateRegionMakeWithDistance(pinlocation, 1000, 1000);
    }
}
} // end of if of sqlite3 open
}

```

- 17.** Running the project with San Francisco's simulated location on the iPad produces the following results:



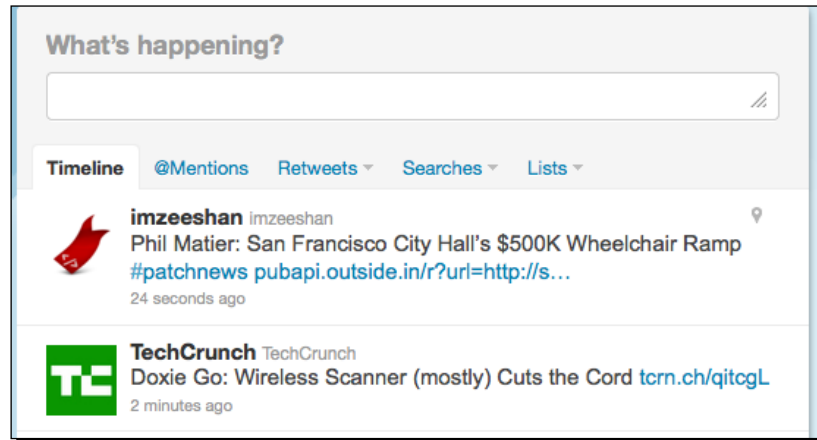
18. Clicking on the **Detail** accessory item loads more information about the news item selected.



- 19.** Check the twitter integration (on iPad build only) by clicking on the **Tweet** button. Make sure that you have enabled your Twitter account on your iOS device.



- 20.** Check your Twitter page on the web, and you should see the news story posted with the hash tag - #Patchnews.



- 21.** When tweeting from the `TweetComposeViewController`, we can also tweet our own location by using the **Add Location** option from the same. The result is what you can see from this screenshot. Your tweets get tagged with your location! Click on the marker on the tweet to see the map on <http://www.twitter.com>.

What just happened?

We created a Geo Aware News Application: PacktNews, using the Patch News API – An AOL product, using some cool iOS 5 APIs – StoryBoard, Twitter Integration, and the new `NSJSONSerialization` class.

The `instantiateViewControllerWithIdentifier` method of the storyboard helps us initialize the **Details** page, and passes it to the Storyboard (as the current scene). The `pushViewController` method of the `UINavigationController` class updates the display to the passed View Controller, in our case, the **Details View Controller (Second View Controller)**.

The Ads from the Admob Ad Network are controlled through the `SecondViewController.m` file's `viewDidLoad` method. You can change the placement of the ads by modifying the following line of code:

```
bannerAdView = [[GADBannerView alloc]
                initWithFrame:CGRectMake(0.0,45.0,
                GAD_SIZE_320x50.width,
                GAD_SIZE_320x50.height)];
```

Summary

In this chapter, we learned how to use AOL's Patch News API that empowers us to build News apps similar to patch.com . We also did some smart application UI modeling using Storyboards in the Interface Builder.

Specifically, we covered the following topics:

- ◆ Consuming the Patch News API
- ◆ Storing news offline in SQLite
- ◆ Adding Geo Fences in our apps
- ◆ Understanding Storyboard and key terms
- ◆ Building the `PacktNews` – News app

In the next chapter, we will build our final app for the book – `TweetGovern: Social Governance`.

10

Social Governance—TweetGovern

The year 2011 saw a lot of world-wide revolutions fuelled by open speech and technology, especially Twitter and Facebook. In fact, the role of Twitter has been greatly acknowledged to a great extent, sometimes being able to deliver news faster than traditional news media and radio. The Twitter integration in Apple's iOS devices brought about by iOS 5, makes building Twitter-based applications for iOS devices, as easy as a breeze.

In this chapter, we set out to create a Twitter-based app (and hopefully a revolution) that helps local residents submit complaints about their society and neighborhood. It could be a simple request from garbage clearance to safety complaints about streetlights or traffic lights.

In this chapter, we will deal with the following topics:

- ◆ Social governance – an overview
- ◆ TweetGovern – behind the scenes
- ◆ Building the home screen
- ◆ Showing nearby issues
- ◆ Submitting an issue
- ◆ Voting for an issue
- ◆ Building the app

So let's get on with it...

Social governance – an overview

The use of social media tools, such as Twitter, Facebook, and so on, for governance along with free and open government data through initiatives, such as <http://datasf.org> and <http://www.data.gov/>, has led to a global movement termed **Gov 2.0** and/or **Social Governance**.

Using technology for the betterment of society should be the ultimate goal for technology. Organizations and governments over the world are now coming forward to bring about this revolution through social governance.

From a simple developer's point of view, it means building apps and consuming free public data to present the right information to the users and authorities, and present a way to resolve issues. The ability to vote an issue as important or non-critical lies in the hands of the denizens; they can choose to vote an issue up to the highest priority.

There are companies that have a successful business around this model, where these companies charge for the technology and apps, while keeping the data public and free. **SeeClickFix** (<http://seeclickfix.com>) is one such company that has a successful business model around social governance. While municipalities and city civic bodies pay a monthly or annual subscription charge for the service, consumers/users/denizens are provided with free website and mobile apps to report issues, vote for an issue, and get their complaints fixed. The issues that are voted to the top are fixed first. Data collected through such initiatives has helped fix as much as 50 percent of the local issues reported.

A note of caution while designing such websites or mobile apps - some users, if not most users, will want to remain anonymous while submitting requests, due to fear of reprisal action by some authorities. So, while developing your apps, you need to provide a platform to allow anonymous submissions of request. But at the same time, it also means more spam, so there is a trade off here, for simple applications, such as **TweetGovern** – we assume that the issues submitted by users are day-to-day essential problems faced by the public, so anonymity is not a big problem. However, if we were to involve police action, smuggling, or drug trafficking, it definitely needs the user's identity management.

TweetGovern – behind the scenes

It takes a lot more to build a successful iOS app product, the product being the iPhone/iPad app in the iTunes store, while the backend (LAMP, Java, Ruby on Rails, or even Microsoft's .Net) provides the data and communication interface to our app, often through a REST interface or a Web Service API. A very important aspect of your app is also its design and layout. Smashing magazine has a very nice article on *How to create your first iPhone application* at <http://coding.smashingmagazine.com/2009/08/11/how-to-create-your-first-iphone-application/>, which covers the complete app-development life cycle from concept to wireframes, from tools to market research, and finally, submitting the app to the iTunes store.

Coming back to TweetGovern, we need to identify how the backend of our app will work, since we need to be able to do the following:

- ◆ Submit an issue
- ◆ Vote for an issue
- ◆ Search for issues by category
- ◆ Display issues by the user's location
- ◆ Allow the app to capture the user's identity

While working on the book, I investigated a couple of approaches for the TweetGovern backend, including a PHP + PostgreSQL + PostGIS backend, for creating a mini CMS for TweetGovern, using a third-party API provider for mobile apps, **StackMob** (<http://stackmob.com/>), and the last option being the great Twitter Integration and API available in iOS 5. We will move with the Twitter approach to building TweetGovern, for the following reasons:

- ◆ It works seamlessly with Twitter, no user database is needed
- ◆ No backend or CMS is needed, hence no API is required, which reduces the time to market
- ◆ Twitter has an excellent Geo Search API for tweets
- ◆ Worldwide coverage on launch
- ◆ Twitter love!!

However, unless any of our readers wants to take a different approach, we propose the following database schema for storing the issues and issue categories structure.

The following database schema represents the **Issues table**:

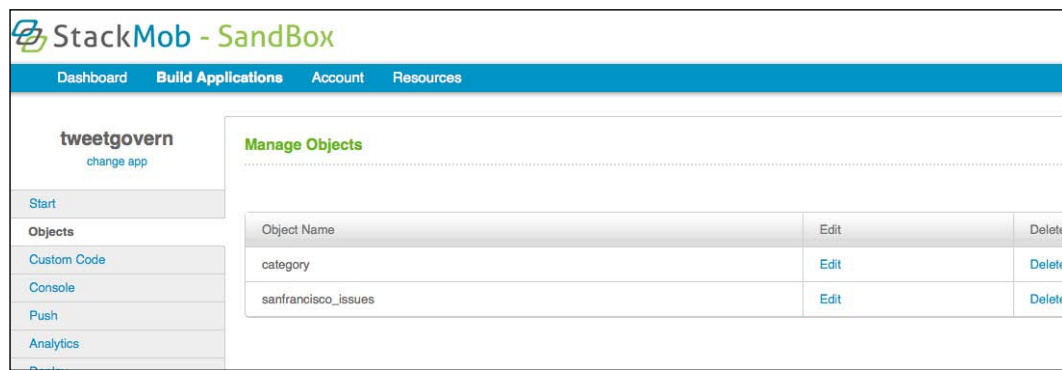
issue_id (integer)	issue_image
issue_title(text or varchar)	issue_votes
issue_category(varchar or integer)	issue_created_date
issue_lat	issue_modified_date
issue_lon	issue_status
issue_description	

The following database schema represents the **Category** table:

category_id (integer)
category_name (text or varchar)

Stackmob

Stackmob is a new startup that provides the data middleware for the mobile applications. It has been called the *Heroku for Mobile*, providing API, OAuth, push notifications, analytics, and social integration for mobile apps developers through their own web console and iOS/Android SDKs. We were lucky to get a **beta invite**, and tried creating the database structure for TweetGovern as discussed previously. You can get access to the service by signing up at <http://www.stackmob.com/>.



We created two objects, one named category that holds all the categories of issues that we plan to incorporate in our app, and the second object being a collection for issues for San Francisco. Note that our goal through the StackMob approach is to create city buckets for issues, so that an object for San Francisco, the other for New York, and so on. The question here is **Why?** The buckets approach is taken to circumvent the Geo Search API, since there is none provided by StackMob, so here, we are using the best approach of creating different objects for each city, so that searching for issues based on regions becomes easier for us, on the Objective-C side.

The city-based **Issues** object is defined as follows:

Object Name: sanfrancisco_issues

Created as a User Object: No

Fields

Add Field

Field Name	Field Type
sanfrancisco_issues_id	string [primary key]
issue_lat	float
createddate	integer
issue_title	string
issue_votes	integer
issue_email	string
lastmoddate	integer
issue_category	integer
issue_description	string
issue_image	string
issue_lon	float
issue_status	boolean

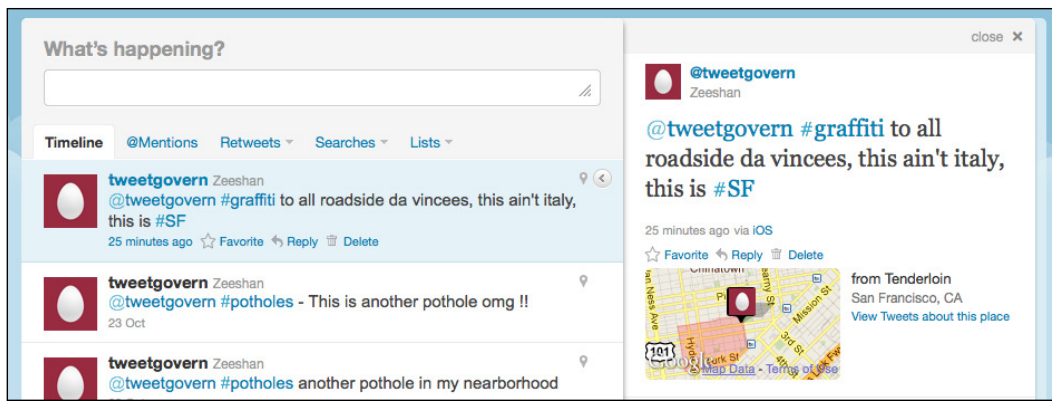
Our approach: Twitter

As we discussed before, we will make our social governance app using Twitter as the backend. However, some questions arise, for example: how will we identify the type of an issue? Whether it is a pothole complaint or graffiti alert? The solution to these questions is the **Twitter Search API**. All our issues will be hash tagged with the issue category and will be directed to a Twitter account that we created just for TweetGovern (@tweetgovern).

Here is an example: We created a simple example app that uses a simple `TWTweetComposeViewController` dialog-box to tweet about issues, by using the `@tweetgovern` handle and a `#graffiti` hash tag, and we added our location.

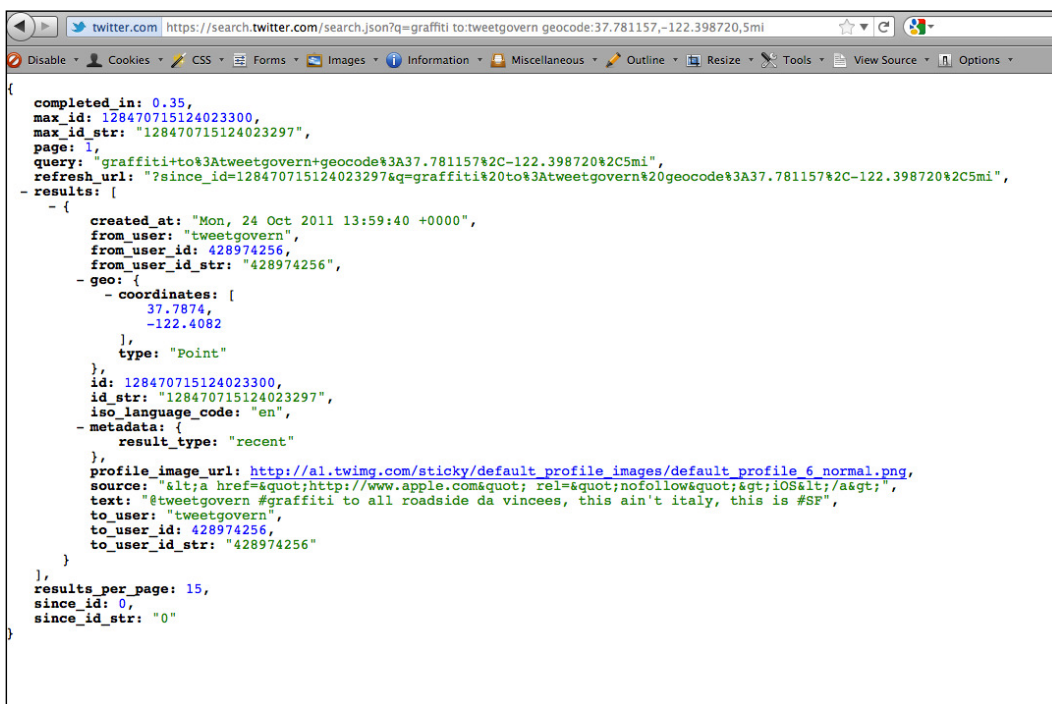


Once you added some text and tweeted it, the tweet appears on your twitter timeline as follows:



Now, we can access these tweets by using the Search API from Twitter with the following URL:

`https://search.twitter.com/search.json?q=graffiti%20to:tweetgovern%20geocode:37.781157,-122.398720,5mi`



We use the following categories and associated hash tags for our TweetGovern app:

Category name	Hash tag
Potholes	#potholes
Graffiti	#graffiti
Garbage	#garbage
Street light	#streetlight
Drainage	#drainage
Health hazard	#healthhazard
Noise	#noise
Traffic light	#trafficlight
Street cleaning	#streetcleaning
Damaged parking	#damagedparking
Others	#others

We begin building our app by assembling all the toolkits, SDK's, icons, images, and so on, and starting with the home screen.

Icons and images

For the app icon, we choose the avatar icon from <http://www.smashingmagazine.com/2008/11/05/dressup-avatars-icon-set/> - designed by *Dante Michael Afrondoza* (<http://www.iconka.com>).



Modify the image for iPhone, iPhone Retina, and iPad by using a size of 57x57, 114x114, and 72x72, respectively.

The application background images have been sourced from <http://allur.co/blog/rounded-pricing-info-callouts-psd/>. They have been resized for iPhone Landscape, iPhone Landscape Retina, iPad Landscape, and iPad portrait sizes with resolutions of 320x480, 640x960, 1024x748, and 768x1004, respectively, and added to the project through Xcode.

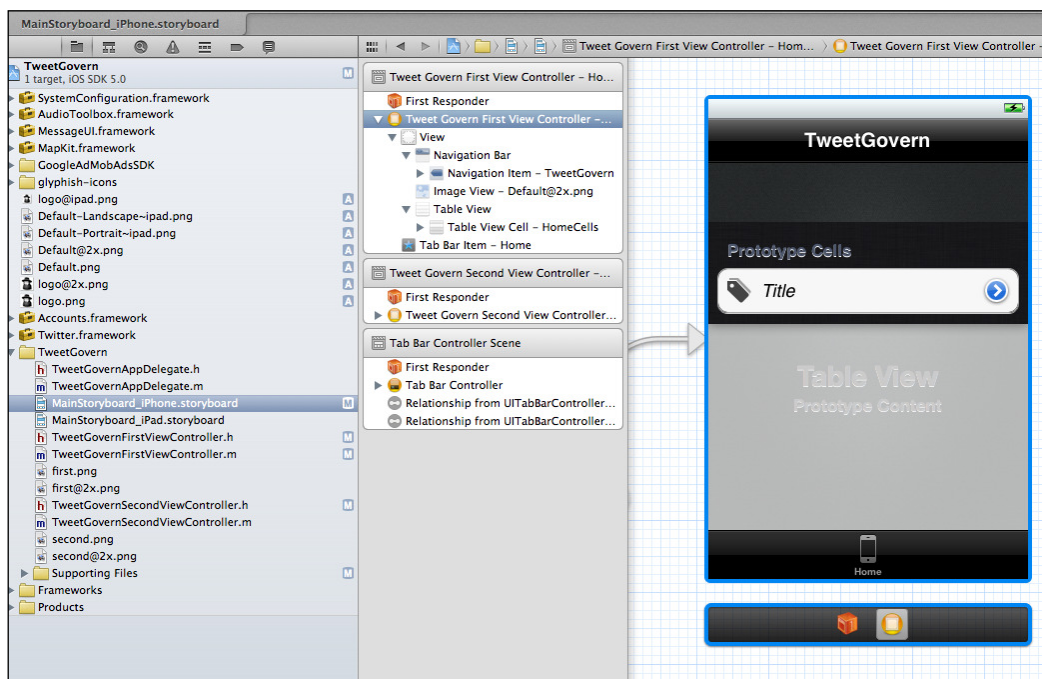
We also add the glyphish-icons, as we used in the WeatherPackt app before.

SDKs and frameworks

We add the Google AdMob SDK and included the Twitter and MapKit framework in our project.

Time for action – creating the UI for TweetGovern

1. From Xcode, create a new project by selecting the **Tabbed Application** template. Name it `TweetGovern`, and class prefix as `TweetGovern` too.
2. Check the **Use StoryBoard** option to enable storyboarding in your app.
3. Open `MainStoryboard_iPhone.storyboard` add a `UIImageView` to the main View. This will be our app background image, which is similar to the loading image.
4. Then add a `UINavigationController` object, with a `UINavigationController` item titled `TweetGovern`.
5. Next, add a `UITableView` from the **Attributes Inspector | View** option. Set the **Background** option to **View Flipside background color**, and change the table style to **Grouped**. Select the **Table View Cell** (prototype cells in Interface Builder – that acts a template for all cell rows in the table view), and change its identifier in **Attributes Inspector** to **HomeCells**, **Selection** as **Blue**, and **Accessory** to **Detail Accessory**. Finally, select the tags images (`15-tags.png`) from the `glyphish` icon set in the **Image** option.
6. Your home screen in Xcode should now look as follows:



- 7.** Now that we have the UI for the TableView done, we add the relevant links to the home screen through code. We create a simple array and add its items to the TableView through the `cellForRowAtIndexPath` UITableView delegate method. Add the following code in your `viewDidLoad` method (in the `TweetGovernFirstViewController.m` file)

```
homeItems = [[NSMutableArray alloc] init];
NSString *createIssue =@"Create an Issue";
NSString *nearbyIssue =@"Show Nearby Issues";
NSString *searchIssue =@"Search for Issues";
NSString *aboutTweetGovern =@"About TweetGovern";

[homeItems addObject:createIssue];
[homeItems addObject:nearbyIssue];
[homeItems addObject:searchIssue];
[homeItems addObject:aboutTweetGovern];
```

- 8.** In your `cellForRowAtIndexPath` method, add the array defined in the cell of TableView.

```
-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    NSString *cellId      = @"HomeCells";
    UITableViewCell *cell  = [tableView dequeueReusableCellWithIdentifier:
Identifier:cellId];

    if (cell==nil)
    {
        cell = [[UITableViewCell alloc] initWithStyle:
UITableViewCellStyleValue1 reuseIdentifier:cellId];
    }

    NSString *cellContent = [homeItems
                             objectAtIndex:indexPath.row];
    cell.textLabel.text   = cellContent;

    return cell;
}
```

9. The total count of the cells for your `UITableView` is controlled by the `numberOfRowsInSection` delegate method, where we pass the array count as the number for cells in our `UITableView`. Try adding one more value in the array, and that should reflect in your UI immediately.

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

10. If everything goes fine, your app should look similar to the following screenshot on the iOS simulator:



What just happened?

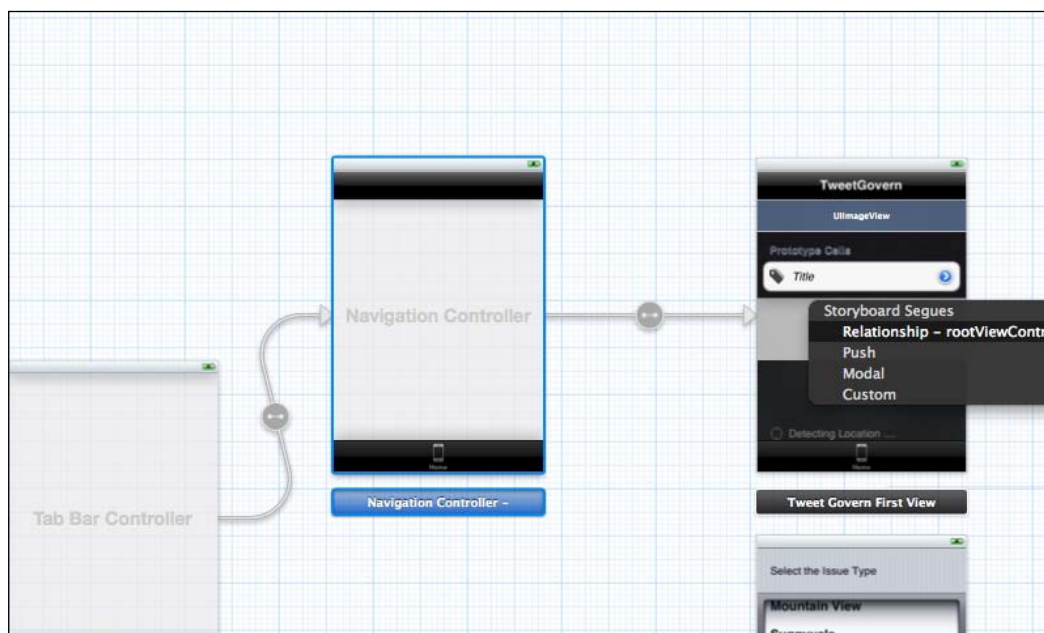
We created the home page of our TweetGovern app by using a **Tabbed Application** template, some free icons and background images and integrated Google Admob SDK within the app.

Having defined the home screen, we now move forward to integrating user location in the app and other functionalities for the app.

Time for action – detecting the user location and showing nearby issues

We begin adding location support in our app and quickly showing nearby issues by querying the Twitter timeline.

1. We modify our project by incorporating a Navigation Controller in our main UI flow. This is done in the Interface Builder, by dragging a UINavigationController object from the **Object Library**. This helps us control the application navigation flow easily.
2. Connect a segue from the Navigation Controller to the home screen UIViewController, by control dragging your mouse pointer from the Navigation Controller to the home screen View, and selecting **Relationship – rootViewController** from the pop-up menu.



3. On the home View UI (First View Controller), name the view controller as Home, and the identifier as home. We will need these View identifier names later on to call each View on demand.

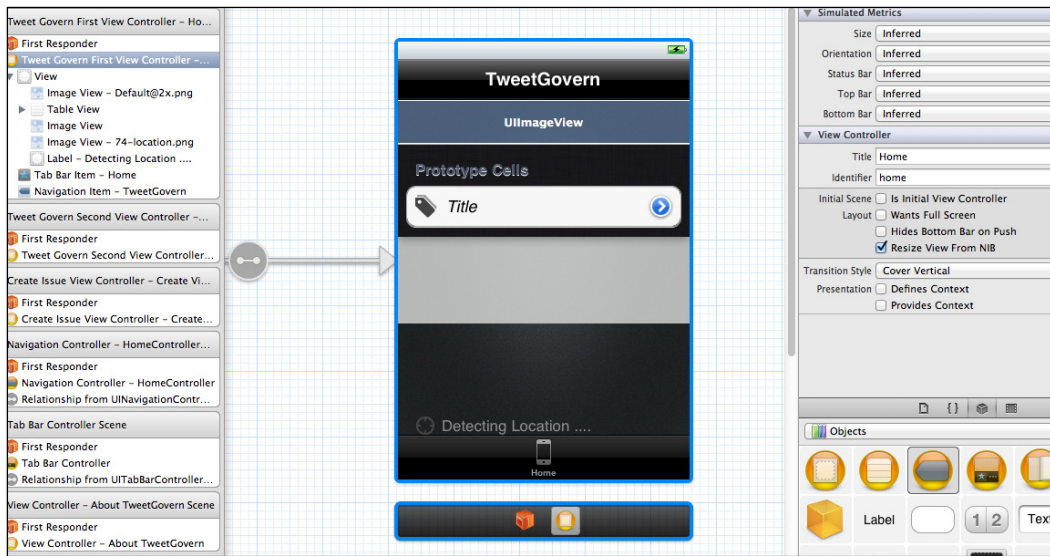
4. In the place where we want an advertisement, we place a dummy UIImageView, and modify our viewDidLoad method to use the bounds (area) of this region for the ads placement. This UIImageView needs to be paired with a corresponding variable in our class declaration. We define that variable as follows:

```
UIImageView      *bannerImage;
```

The code to use this variable in the ads placement is as follows:

```
if ([deviceType isEqualToString:@"iPhone"])
{
    bannerAdView = [[GADBannerView alloc]
                    initWithFrame:bannerImage.bounds];
}
```

5. We want the home screen to be more informative, by showing the user's detected location in the footer. We use a UIImageView and a UILabel to show this information, by showing a Location icon on the UIImageView, and the text **Detecting Location...** on the UILabel. The icon is sourced from the glyphish-icons set.
6. Your home screen should now look similar to the following screenshot:



- 7.** We use the new Reverse Geocoding class in iOS5 CLGeocoder to convert the latitude/longitude values to address, street name, and city name, by using the following code in our `didUpdateToLocation` method.

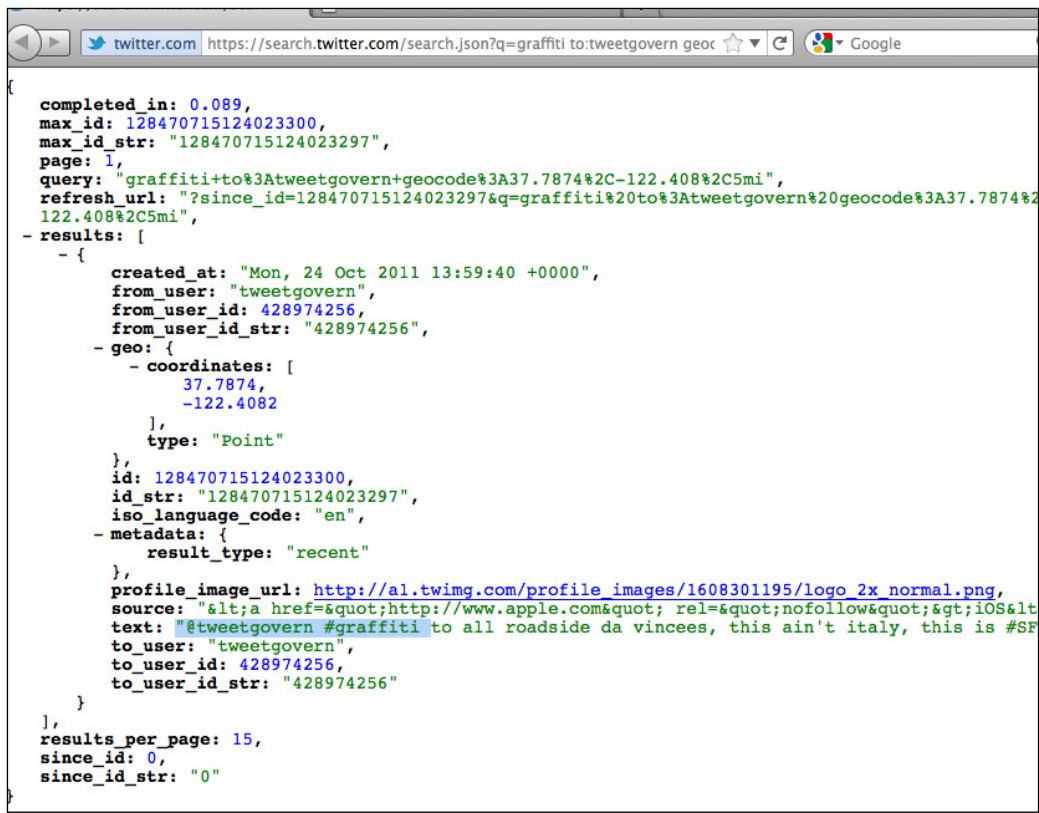
```
// Get City Name, Address with Reverse Geocoding

geocoder = [[CLGeocoder alloc] init];

[geocoder reverseGeocodeLocation:userLocation
completionHandler:^(NSArray *placemarks, NSError *error)
{
    for (CLPlacemark *placemark in placemarks)
    {
        currentCity          = placemark.locality;
        currentStreet        = placemark.thoroughfare;
        currentAddress       = placemark.subThoroughfare;
        if (currentStreet)
        {
            currentLocationLabel.text = [currentCity
            stringByAppendingFormat:@"", "%@", currentStreet];
        }
        else
        {
            currentLocationLabel.text = currentCity;
        }
    }
}];
```

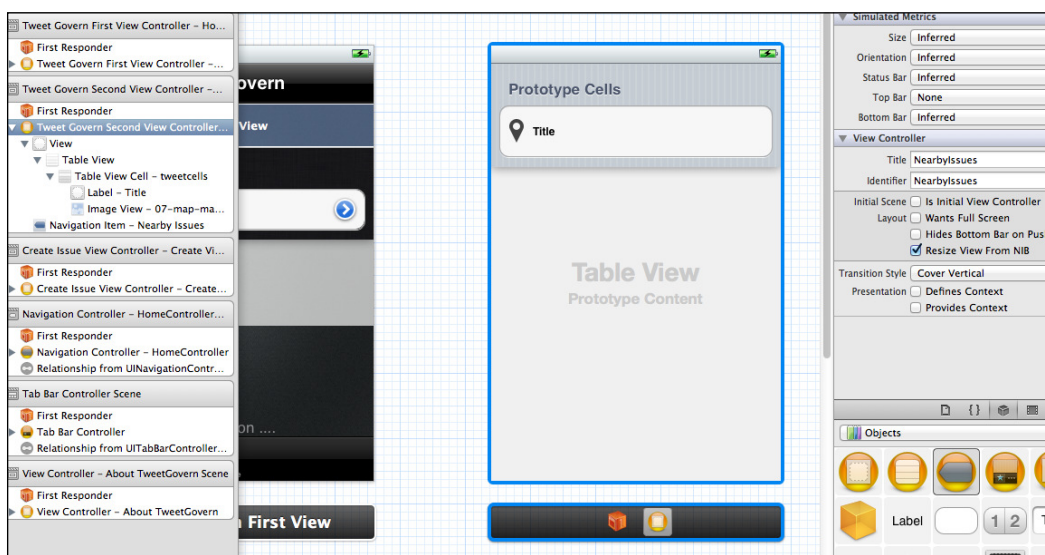
- 8.** `currentLocationLabel` is the variable which is paired to the `UILabel` for **Detecting Location....**

9. Now, we move forward to show the nearby issues. For this, we use the simple Twitter search API - <https://search.twitter.com/search.json?q=to:tweetgovern geocode:37.7874,-122.408,5mi>, where geocode contains the location detected from the iPhone. We will use this API call and its variation to query the Twitter timeline as per our app requirement. For example, to search only for **graffiti** issues, we modify the above API to <https://search.twitter.com/search.json?q=graffiti%20to:tweetgovern%20geocode:37.7874,-122.408,5mi>, which provides us the following result:



```
{
  completed_in: 0.089,
  max_id: 128470715124023300,
  max_id_str: "128470715124023297",
  page: 1,
  query: "graffiti+to%3Atweetgovern+geocode%3A37.7874%2C-122.408%2C5mi",
  refresh_url: "?since_id=128470715124023297&q=graffiti%20to%3Atweetgovern%20geocode%3A37.7874%2C-122.408%2C5mi",
  results: [
    {
      created_at: "Mon, 24 Oct 2011 13:59:40 +0000",
      from_user: "tweetgovern",
      from_user_id: 428974256,
      from_user_id_str: "428974256",
      geo: {
        coordinates: [
          37.7874,
          -122.4082
        ],
        type: "Point"
      },
      id: 128470715124023300,
      id_str: "128470715124023297",
      iso_language_code: "en",
      metadata: {
        result_type: "recent"
      },
      profile_image_url: http://a1.twimg.com/profile_images/1608301195/logo_2x_normal.png,
      source: "<a href='\"http://www.apple.com\"'; rel='\"nofollow\"';>iOS</a>",
      text: "@tweetgovern #graffiti to all roadside da vincees, this ain't italy, this is #SF",
      to_user: "tweetgovern",
      to_user_id: 428974256,
      to_user_id_str: "428974256"
    }
  ],
  results_per_page: 15,
  since_id: 0,
  since_id_str: "0"
}
```

- 10.** Our View Controller that handles the nearby issues display is the `NearbyIssues` View Controller. We add a `UITableView` to it, and style the **Prototype Cell Row** (with identifier as `tweetcell`), by adding a marker icon to it (again from `glyphish`) and changing the text font to `System Bold 12.0`. This View should look as follows:



- 11.** This View will be loaded when the user clicks on the **Show Nearby Issues** option from the app's home screen. This is controlled by the `didSelectRowAtIndexPath` method of the `UITableView`, which is defined in `TweetGovernFirstViewContr` `oller.m` file as follows:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath
:(NSIndexPath *)indexPath
{
    if (indexPath.row==1)
    {
        nearbyIssuesView = [self.storyboard
            instantiateViewControllerWithIdentifier:@"NearbyIssues"];

        nearbyIssuesView.userLocation = userLocation;
        [self.navigationController pushViewController:nearbyIssuesView
            animated:YES];
    }

    else if (indexPath.row==0)
```

```

    {
        nearbyIssuesView = [self.storyboard
        instantiateViewControllerWithIdentifier:@"createview"];
        [self.navigationController pushViewController:nearbyIssuesView
        animated:YES];
    }

    else if (indexPath.row==3)
    {
        nearbyIssuesView = [self.storyboard
        instantiateViewControllerWithIdentifier:@"about"];
        [self.navigationController
        pushViewController:nearbyIssuesView animated:YES];
    }

    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}

```

- 12.** This is where all the views are loaded according to the option selected by the user. In the **Nearby Issues** option, note that we have created another `CLLocation` object in the **TweetGovernSecondViewController** class, to pass the value of the current location from the home screen to the second, which will eventually be passed onto the Twitter Search API.
- 13.** When the **NearbyIssues View** loads, its `viewDidLoad` method is called. Here is where the Twitter Search API is called as follows:

```

jsonContent      = [[NSMutableData alloc] init];
tweets           = [[NSMutableArray alloc] init];
tweetsIds        = [[NSMutableArray alloc] init];
[self showNearByTweets];

```

- 14.** The `jsonContent` variable holds the raw JSON received from `NSURLConnection`, while the `tweets` array contains the tweet's text, and `tweetsIds` contains the corresponding tweet id (this tweet id is needed for voting - also known as **retweet**). The `showNearByTweets` is defined as follows:

```

- (void) showNearByTweets
{
    NSString *userLat = [[NSString alloc]
        initWithFormat:@"%g", userLocation.coordinate.latitude];
    NSString *userLon  = [[NSString alloc]

```

```
        initWithFormat:@"%g",userLocation.coordinate.longitude];
NSString *url      = [NSString stringWithFormat:
@"https://search.twitter.com/search.json?q=to:tweetgovern
geocode:%@,%@,5mi",userLat,userLon];

url                = [url
        stringByAddingPercentEscapesUsingEncoding:
        NSUTF8StringEncoding];

NSURL      *urlToRequest    = [[NSURL
        alloc] initWithString:url];
NSURLRequest *request       = [NSURLRequest
        requestWithURL:urlToRequest];
URLConnection      = [[NSURLConnection alloc]
        initWithRequest:request delegate:self startImmediately:YES];
}
```

- 15.** By now, you would be familiar with the JSON parsing, as we have done it in the examples before. So, we will not go into the details; however, the important line of code here is the parsing of the tweet text and tweet id in the `connectionDidFinishLoading` method:

```
NSString *tweetTitle    = [[[items objectAtIndex:0]
        objectAtIndex:i]objectForKey:@"text"];
NSString *tweetId       = [[[items objectAtIndex:0]
        objectAtIndex:i]objectForKey:@"id_str"];

if(![tweets containsObject:tweetTitle])
{
    [tweets addObject:tweetTitle];
}

if(![tweetsIds containsObject:tweetId])
{
    [tweetsIds addObject:tweetId];
}
```

- 16.** We created two simple arrays to hold the tweet text and tweet id, but you are free to use any other logic, maybe a multi-dimensional array or a full-blown class to hold all the tweet information.

- 17.** The `tweets` array is then used in the `cellForRowAtIndexPath` method of the `UITableView`, which renders the tweets on the `tableView`.

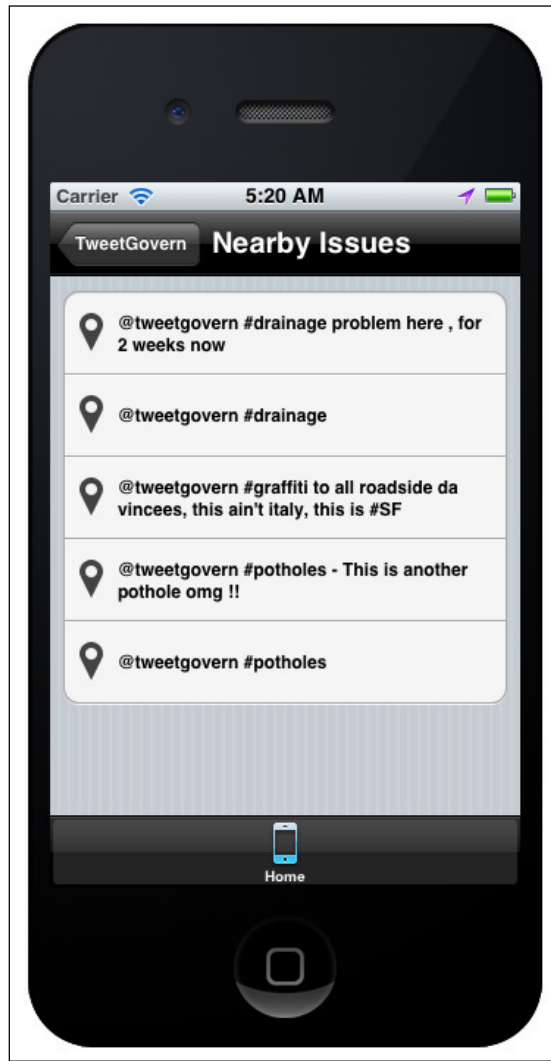
```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPathIndexPath: (NSIndexPath *)indexPath
{
    NSString *cellId      = @"tweetcells";
    UITableViewCell *cell  = [tableView
        dequeueReusableCellWithIdentifier:cellId];

    if (cell == nil)
    {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleValue1
            reuseIdentifier:cellId];
    }

    NSString *cellContent = [tweets objectAtIndex:indexPath.row];
    cell.textLabel.text   = cellContent;
    return cell;
}
```

- 18.** Note that from our discussion about the use of the cell identifier in Interface Builder, we named it `tweetcell`. In our code here, we use the same, so that the UI modeled in Interface Builder is applied to the table row cells that we are using in the code. If you choose a different name, you can notice the difference in the visual UI when you run the app.
- 19.** The `UITableView` in the `TweetGovernSecondViewController` class is named `myTweetTable`. Do not forget to add this outlet in Interface Builder to the table View, and the `datasource` and `delegate` outlet from the table View to the main controller. Also, use the delegates in your class declaration with `<UITableViewDataSource, UITableViewDelegate>`.

- 20.** Running the app now produces the following result. Note some dummy tweets/issues that were created to understand the app workflow:



What just happened?

Storyboards are an exciting new feature in iOS5. We used it successfully to model our home and nearby issues screen, all on the same page in Interface Builder. The **zoom-in** and **zoom-out** feature helps navigate the larger landscape in storyboarding, but the idea of all the app screens on the same page is awesome and helpful to keep a consistent design across the app UI.

We looked at using Twitter eco system for the logic of our app. This is a smart way of problem solving as well. We used Twitter since it fitted all our requirements perfectly, so we didn't need to create a backend, an API, and other management hassle. Anyone working or has worked in a tech startup before would know.

We learned about StoryBoard View management by using the `[self.storyboard instantiateViewControllerWithIdentifier:@"xxxxxxx"]`; line of code, where xxxxxx is the View identifier. The source code of the full app is available on the book's website: project titled TweetGovern.

Now we look at using the Twitter re-tweet logic to build a voting solution for our TweetGovern app.

Time for action – creating and voting for an issue

Our main screen for the app is the **Create Issue** screen, where we will allow the user to choose from a list of issue categories, and allow them to proceed to create an issue with the hashTag associated with the said issue category.

1. Add a new ViewController to your project, by selecting the `UIViewController subclass` option from the **File | New | New File**. Name it `CreateIssueViewController`, and add it to your project. Do not select any of the **Targeted for iPad** or **With XIB for user interface** options, since we will use the Storyboard for user interface.
2. We use a `UIPickerView` to display a category picker, which allows the user to select any of the issue categories, which will be added to the `UIPickerView` through an array called `issuesCategory`. We will need an `IBAction` to show the tweet-box for creating an issue. Open the `CreateIssueViewController.h`, and add the following code:

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

@interface CreateIssueViewController : UIViewController
{
    UIPickerView *pickIssues;
    NSMutableArray *issuesCategory;
    NSString      *hashTag;
}

@property (retain, nonatomic) IBOutlet UIPickerView *pickerView;

- (IBAction)createIssue:(id) sender;

@end
```

- 3.** Now in the `viewDidLoad` method of the `CreateIssueViewController.m` file, we add the list of categories that we want to enable in our application:

```
issuesCategory = [[NSMutableArray alloc] init];
hashTag        = [[NSString alloc] init];

[issuesCategory addObject:@"PotHoles"];
[issuesCategory addObject:@"Graffiti"];
[issuesCategory addObject:@"Garbage"];
[issuesCategory addObject:@"Street Light"];
[issuesCategory addObject:@"Drainage"];
[issuesCategory addObject:@"Health Hazard"];
[issuesCategory addObject:@"Noise"];
[issuesCategory addObject:@"Traffic Light"];
[issuesCategory addObject:@"Street Cleaning"];
[issuesCategory addObject:@"Damaged Parking Meter"];
[issuesCategory addObject:@"Others"];
```

- 4.** The array of categories defined in the previous step are added to the `UIPickerView` by the `titleForRow` method of the `UIPickerView`, where we pass each array from the `issuesCategory` array to the `UIPickerView` object.

```
(NSString *)pickerView:(UIPickerView *)thePickerView
titleForRow:(NSInteger)row forComponent:(NSInteger)component
{
    return [issuesCategory objectAtIndex:row];
}
```

- 5.** The `numberOfRowsInComponent` method of the `UIPickerView` also plays an important role in assigning the categories to the `UIPickerView`, by telling the `UIPickerView` the number of rows expected in its View. Depending on this count, the `titleForRow` will run x number of times, where x is the count.

```
(NSInteger)pickerView:(UIPickerView *)thePickerView numberOfRowsIn
Component:(NSInteger)component
{
    return [issuesCategory count];
}
```

- 6.** Finally, when any row of the `UIPickerView` is selected, the `didSelectRow` method is called. Here we initialize the right Twitter `hashTag` to be tweeted, based on the category selected as follows:

```
- (void)pickerView:(UIPickerView *)thePickerView
didSelectRow:(NSInteger)row inComponent:(NSInteger)component {

    if([[issuesCategory objectAtIndex:row]
```

```
isEqualToString:@"PotHoles"]])
{
    hashTag       =@"#potholes";
}

if([[issuesCategory objectAtIndex:row]
isEqualToString:@"Graffiti"])
{
    hashTag       =@"#graffiti";
}

if([[issuesCategory objectAtIndex:row]
isEqualToString:@"Garbage"])
{
    hashTag       =@"#garbage";
}

if([[issuesCategory objectAtIndex:row] isEqualToString:@"Street
Light"])
{
    hashTag       =@"#streetlight";
}

if([[issuesCategory objectAtIndex:row]
isEqualToString:@"Drainage"])
{
    hashTag       =@"#drainage";
}

if([[issuesCategory objectAtIndex:row] isEqualToString:@"Health
Hazard"])
{
    hashTag       =@"#healthhazard";
}

if([[issuesCategory objectAtIndex:row] isEqualToString:@"Noise"])
{
    hashTag       =@"#noise";
}

if([[issuesCategory objectAtIndex:row] isEqualToString:@"Traffic
Light"])
{
    hashTag       =@"#trafficlight";
}
```

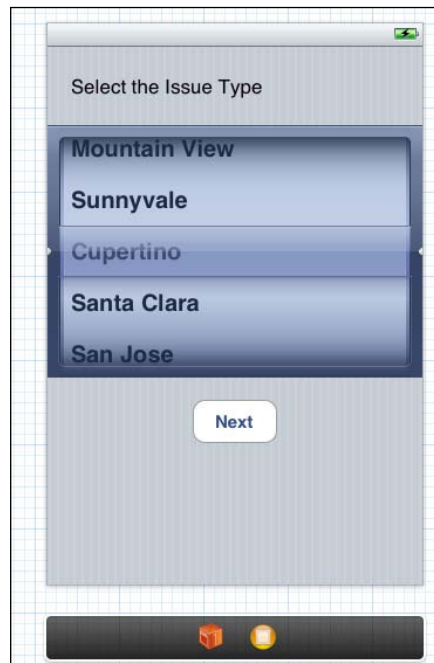
```
    }

    if ([[issuesCategory objectAtIndex:row] isEqualToString:@"Street
Cleaning"])
    {
        hashTag       =@"#streetcleaning";
    }

    if ([[issuesCategory objectAtIndex:row] isEqualToString:@"Damaged
Parking"])
    {
        hashTag       =@"#damagedparking";
    }

    if ([[issuesCategory objectAtIndex:row] isEqualToString:@"Others"])
    {
        hashTag       =@"#others";
    }
}
```

7. Open your Storyboard, and create another UIViewController on it. Change its identifier to `createview`, and add a UIPickerView object from the object library, as well as a UILabel and UIButton. Change the text on the UIButton to **Next**, and connect it to the IBAction - `createIssue` model on the UI as shown in the following screenshot:



8. Now, when we click on the **Next** button, the `createIssue` `IBAction` is fired. Here we show the inbuilt tweet modal pop-up, and prefill it with the `@tweetgovern` tag and the `hashTag` for the category.

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

    if([TWTweetComposeViewController canSendTweet])
    {
        TWTweetComposeViewController *tweetViewController =
            [[TWTweetComposeViewController alloc] init];
        [tweetViewController setInitialText:[NSString
            alloc]initWithFormat:@"@tweetgovern %@",hashTag]];

        [tweetViewController
            setCompletionHandler:^(TWTweetComposeViewControllerResult result)
            {
                NSString *tweetOutput;

                switch (result) {
                    case TWTweetComposeViewControllerResultCancelled:
                        tweetOutput = @"Tweet sending Cancelled by User ";
                        break;
                    case TWTweetComposeViewControllerResultDone:
                        tweetOutput = @"Tweet sent successfully";
                        break;
                    default:
                        break;
                }
                [self dismissModalViewControllerAnimated:YES];
            }];
        [self presentModalViewController:tweetViewController
            animated:YES];
    } // end of if canSendTweet
}
```

9. The users can add more information to the tweet they like, although it should be within 140 characters. Note the **Add Location** option in the tweet box; we will need this to be used always, so that we can determine the location of the tweet while searching, voting, and for other aspects of our TweetGovern app.

- 10.** Coming back to the voting part, we missed adding a vote button to the UITableView in TweetGovernSecondViewController, which handles the view for the nearby issues View. Open the TweetGovernSecondViewController.m file, and within the cellForRowAtIndexPath method, add the following code to add a UIButton the table cell, and fire an event when the button is clicked.

```
UIButton *cellButton = [UIButton
    buttonWithType:UIButtonTypeRoundedRect];
[cellButton setFrame:CGRectMake(260.0, 30.0, 45.0, 20.0)];
[cellButton setTitle:@"Vote" forState:UIControlStateNormal];
[cellButton addTarget:self
    action:@selector(voteForTweet:event:)
    forControlEvents:UIControlEventTouchUpInside];

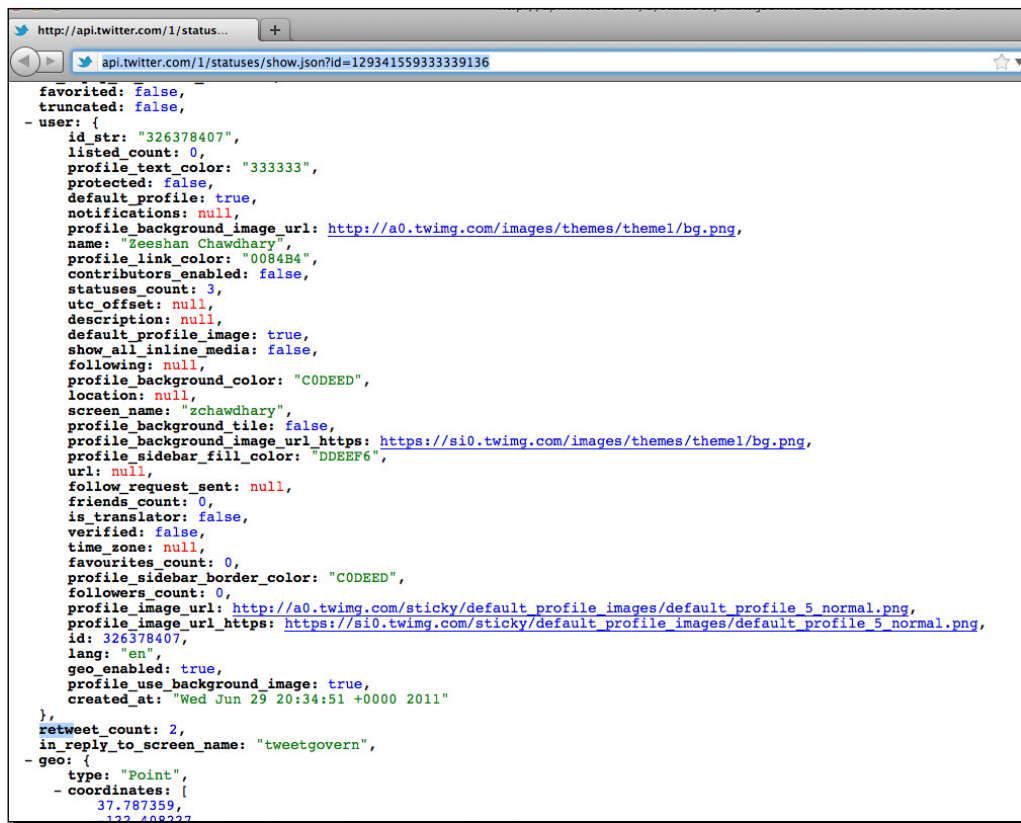
[cell addSubview:cellButton];
```

- 11.** Note the CGRectMake function, which defines the placement of your button on the table view cell. You can play around with the values if you wish to. The following line of code is important here, since it defines a method voteForTweet that will be fired on the button-click using the UIControlEventTouchUpInside definition.

```
addTarget:self action:@selector(voteForTweet:event:)
    forControlEvents:UIControlEventTouchUpInside
```

What is the voteForTweet method, and how does it help us for voting ? Read on...

- 12.** Every tweet has an API with Twitter, which shows its full information, where it was tweeted from and by whom, as well as the retweet count. We use the retweet count as the basis for our vote. Every time a user clicks on the **vote** button, he is in fact retweeting the said tweet. This increments the retweet count, and hence forms the basis of our voting algorithm.
- 13.** For example, the following Twitter API URL shows the retweet count for one of our sample issues created, which returns a retweet count of **2**, since we retweeted by two different twitter accounts to check its validity: <http://api.twitter.com/1/statuses/show.json?id=129341559333339136>



14. Another beauty of the Twitter API is that you can retweet a tweet only once!, which solves another issue for our app – SPAM and increasing vote counts. If you try voting a number of times, you should see the following message in your **Debug Window**:

```

{"errors": "sharing is not permissable for this status (Share
validations failed)\nsharing is not permissable for this status
(Share validations failed)\nsharing is not permissable for this
status (Share validations failed)"}

```

15. Now the `voteForTweet` `IBAction` is defined as follows:

```

- (IBAction)voteForTweet:(id)sender event:(id)event
{

    NSSet *touches = [event allTouches];
    UITouch *touch = [touches anyObject];
    CGPoint currentTouchPosition = [touch
        locationInView:self.myTweetTable];

```



```
NSIndexPath *indexPath = [self.myTweetTable
                           indexPathForRowAtPoint: currentTouchPosition];

NSString *currentTweetId    =    [tweetsIds
                                   objectAtIndex:indexPath.row];
// Make sure to import the Accounts.h file (iOS 5 Twitter API),
// #import <Accounts/Accounts.h>

ACAccountStore *accountStore = [[ACAccountStore alloc] init];

ACAccountType *accountType = [accountStore
                              accountTypeWithAccountTypeIdentifier:
                              ACAccountTypeIdentifierTwitter];

[accountStore requestAccessToAccountsWithType:accountType
withCompletionHandler:^(BOOL accessGranted, NSError *error)
{
    if(accessGranted)
    {
        NSArray *accountsArray = [accountStore
                                   accountsWithAccountType:accountType];
        if ([accountsArray count] > 0)
        {
            // Use the first Twitter account from your iOS device
            ACAccount *twitterAccount = [accountsArray
                                         objectAtIndex:0];

            NSString *url = [NSString stringWithFormat:
                             @"http://api.twitter.com/1/statuses/retweet/%@.json",
                             currentTweetId];

            TWRequest *postRequest = [[TWRequest alloc] initWithURL:[NSURL
            URLWithString:url] parameters:nil requestMethod:TWRequestMethodPOST];

            [postRequest setAccount:twitterAccount];
            [postRequest signedURLRequest];

            [postRequest performRequestWithHandler:^(NSData *responseData,
            NSHTTPURLResponse *urlResponse, NSError *error)
            {
                NSString *output = [[NSString alloc] initWithData:responseData
                encoding:NSUTF8StringEncoding];
            }];
        }
    }
}
```

```

        NSLog(@"%@", output);

        [self performSelectorOnMainThread:@selector(self)
            withObject:output
            waitUntilDone:NO];

        }];
    }
    } // end of if accessGranted
    }];
}

```

- 16.** The first four lines on the previous code convert the user's touch into a corresponding table row ID by using a combination of the `UITouch` and the `CGPoint` classes. The `CGPoint currentTouchPosition = [touch locationInView:self.myTweetTable];` code gets us the current position of the touch location, with respect to our tweet table; note this is in the *x* and the *y* co-ordinate system. The `NSIndexPath *indexPath = [self.myTweetTable indexPathForRowAtPoint: currentTouchPosition];` code converts this *x* and *y* position to a corresponding table cell row position, which will then be used to determine the tweet ID of the said associated **vote** button as follows:
`NSString *currentTweetId = [tweetsIds objectAtIndex:indexPath.row];`
- 17.** Next, we create a **Twitter Account Store Object** (new in iOS5), and use the first available and enabled Twitter account information on the iPhone or any other iOS device as the main Twitter account (since there can be multiple Twitter accounts enabled in your iPhone now).
- 18.** We then create a `TWRequest` object, which is basically an HTTP encapsulation of the Twitter HTTP API, and call the Re-Tweet API for that Tweet ID, passing the Tweet ID obtained earlier to this API call.
- 19.** Before doing so, we need to sign the API call by using `[postRequest signedURLRequest];`, which allows our application to use the device-enabled Twitter account information on the user's behalf. We then retweet the tweet with the `performRequestWithHandler` method. For now, we are just logging the output of this request, but we can extend it to show a visual confirmation of the vote to the end-user.

20. Running the application now gives the following result:



21. Clicking on **Next** after making an issue-type selection shows the next page, which is basically a Twitter modal box.



22. After you are done tweeting about an issue, go back to the home screen, and select **Show Nearby Issues**. This should load the following screen:



23. On the **About TweetGovern** page, we show some Twitter love, by using the official twitter logo from <https://twitter.com/about/resources/logos>.



What just happened?

We complete our app by adding the **Create an Issue** and **Voting** option in our app. We used the Twitter API extensively throughout the various stages of the app, so the last View is a tribute to Twitter with their official logo.

We learnt a new trick of determining a UI element from *x* and *y* values on the screen, using `indexPathForRowAtPoint` method of the `UITableView` class. We also looked at the two new features of iOS5 Twitter integration: **Tweet Box** and the `TWRequest` class.

The source code for the full app is available on the book's website, the author's blog (<http://justgeeks.in>), and Github account (<https://github.com/imzeeshan>). So any of our readers can fork it, contribute to it and hopefully add new exciting features to TweetGovern, and maybe see those changes on the iTunes Store, since we will submit the app to the Apple iTunes Store, hoping that we are in.

Have a go hero - adding search to TweetGovern

You will notice that we modeled our home screen with four options: **Create an Issue**, **Show Nearby Issues**, **About TweetGovern**, and **Search for Issues**. While we discussed the first three in quite detail, we leave the fourth one – **Search for Issues** – as an exercise to the users. Add your own `UIViewController`, and add search capability to the app.



Use simple JSON and `NSURL` requests to the Twitter API for search calls that do not need authentication. For calls that need authentication, use the `TWRequest` class.

We are eager to put your code in our app!!

Pop quiz

1. How does one go about configuring his/her Twitter account in the iPhone, so that the new iOS 5 Twitter APIs can be used in our apps?
 - a. Install the Twitter app from iTunes store.
 - b. Through the **Settings** page on your iOS device.
2. Can you use multiple twitter accounts within your iOS applications?
 - a. Yes.
 - b. No
3. How do you perform a request against the Twitter API?
 - a. HTTP request through `NSURLRequest`
 - b. `TWRequest`

Summary

In this chapter, we learned how to use the new iOS 5 Twitter integration to build a complete iPhone app from scratch, using Twitter API as the core of the app and smart programming logic to enable the app behavior as needed, making our app socially aware by default!

Mobile app developers should keep the following two golden rules in mind while developing or starting their next big iPhone app:

- ◆ Design is the key
- ◆ Simplicity while choosing the application logic

In our case, had we chosen to build a TweetGovern website/backend/CMS first, API thereafter, and business logic for our app later on, we would have never completed the app in time. So, we chose the other way round, and used Twitter as our driving force for both, backend and social.

Specifically, we covered:

- ◆ Building a home screen for our app
- ◆ Detecting the user's location, and showing nearby issues
- ◆ Creating and voting for issues
- ◆ Building the TweetGovern – social governance app

That concludes our last chapter for *iPhone Location Aware Apps – Beginner's Guide*. By now you should have a firm grip on building location-based applications for iOS 5, with location, maps, offline storage, Twitter integration, and speech recognition APIs from Nuance. It has been my sincere effort to teach the readers of this book, how to use the readily available technology to make revenue-generating iOS apps. Companies such as StackMob.com also helps avoid the back-end work for new start-ups by providing the back-end infrastructure to store the business logic. All that the developers have to do now is build a great app and scale.

This book is not the end of our learning. I will be maintaining a forum for this book at <http://books.justgeeks.in>, where I will be updating the source code for our five apps discussed in the book, as well as discussions, suggestions, and errata.

Pop-Quiz Answers

Chapter 1, The Location-Based World

Pop quiz – play safe with location !

1	b
2	b (Pull method, since it queries for location information only on demand and not continuously, thereby saving power)
3	b,

Chapter 2, The Xcoder's World

Pop quiz – so you think you can Xcode

1	b (No, only Intel-based Macs are supported)
2	b (LLVM)

Chapter 3, Using Location in your iOS Apps – Core Location

Pop quiz – location, location, and location

1	b
2	b
3	[CLLocationManager locationServicesEnabled]
4	c
5	b

Chapter 4, Using Maps in your iOS apps - MapKit

Pop quiz – map mania

1	c
2	b
3	a

Chapter 5, Weather App - WeatherPackt

Pop quiz – weather alert

1	b
2	c

Chapter 6, Events App - PacktEvents

Pop quiz – have a blast with events

1	Event Kit Framework and Event Kit UI Framework
2	By using the <code>canSendTweet</code> method from the <code>TWTweetComposeViewController</code> - <code>[TWTweetComposeViewController canSendTweet]</code>)

Chapter 7, Advanced Topics

Pop quiz – the rocket science

1	a
2	a
3	d

Chapter 10, Social Governance - TweetGovern

Pop quiz

1	b
2	a
3	b

Index

Symbols

#damagedparking hash tag 322
#drainage hash tag 322
#garbage hash tag 322
#graffiti hash tag 320, 322
#healthhazard hash tag 322
#noise hash tag 322
#others hash tag 322
#potholes hash tag 322
#streetcleaning hash tag 322
#streetlight hash tag 322
#trafficlight hash tag 322
@tweetgovern handle 320
@tweetgovern tag 339

A

About TweetGovern page 345
accelerometer
 about 229, 230
 accelerometerActive properties 230
 accelerometerAvailable properties 230
 data accessing, motion manager used 231-234
accelerometerActive properties 230
accelerometerAvailable properties 230
active, state 237
Add Location option 339
AddressBook format 74
AGPS 8, 12
airport category, foursquare category hierarchy
 airport food court 252
 airport gates 252

airport lounges 252
airport terminals 252
annotations
 about 117
 adding, to maps 117-119
 custom map annotations 123-125
 draggable annotations 119-121
AnnotationView draggable 122
AOL
 Patch.com 283
API endpoints
 URL 255
API key
 URL 91
AppDelegate class 237
App icon 162
Apple iPhone 14
Apple's rich documentation
 URL 170
application
 didFinishLaunchingWithOptions: method 238
 scheduleLocalNotification method 243
application badge 241
applicationDidBecomeActive: method 238
applicationDidEnterBackground: method 238
applicationWillEnterForeground: method 238
applicationWillResignActive: method 238
applicationWillTerminate: method 238
AR
 about 12, 213
 adding, to iPhone app 213-219
ARC 33

- assistant editor 30
- Assisted GPS. *See* AGPS
- Augmented Reality. *See* AR
- Auth configuration value 254
- authentication, Patch News API 284
- authorizationStatus method 81
- author, taxonomy type 284, 285
- author type, taxonomy type 287, 288
- Automatic Reference Counting. *See* ARC
- avatar icon 322

B

- background app execution
 - about 235
 - iOS application life cycle 236-240
 - local notifications 241
 - local notifications, using 241-245
 - location 236
 - push notifications 240
 - UINavigationController class 241
- background, state 237
- badgeStepper object 241

C

- camera enhancement 32
- cancelAllLocalNotifications method 243
- cancelLocalNotification method 243
- canSendTweet method 212
- category_id (integer) 318
- category_name(text or varchar) 318
- cellForRowAtIndexPath delegate method 253
- cellForRowAtIndexPath method 202, 207, 272, 324, 333, 340
- cellForRowAtIndexPath UITableView delegate method 324
- cell ID positioning 21
- cell tower triangulation 21
- change event 224
- Check-in Here button 280
- checkin IBAction 276
- check-ins 12
- city name
 - location data, converting into 142
- CLGeocode object 74
- CLGeocoder
 - about 74

- MKReverseGeocoder 74
- CLGeocoder class 75
- CLHeadingclass object 222
- CLHeading object 76, 222
- CLLocationManager class 76, 81, 85
- CLLocationManagerDelegate delegate protocol 76
- CLLocationManager object 72
- CLLocation object
 - about 72, 81, 84, 85, 226, 227
 - coordinate property 84
 - course property 84
 - horizontalAccuracy property 84
 - location updates, receiving in application 85-87
 - methods 85
 - speed property 84
 - timestamp property 84
 - user location, with core data 90
 - verticalAccuracy property 84
- CloudMade
 - support for 130
 - URL 130
- CloudMade API
 - OpenStreetMaps, using 131-133
 - used, for building apps 130
- CLPlacemark object 74, 75, 147
- CLRegion class 87, 90
- CMAccelerometerData object 229
- CMDeviceMotion object 229
- CMGyroData object 229
- CMMagnetometerData object 229
- connectionDidFinishingLoading method 259
- connectionDidFinishLoading delegate method 259
- connectionDidFinishLoading method 183, 187, 251, 259, 272, 276, 290, 293, 332
- control segment, GPS 18
- coordinate property 84
- coord variable 194
- Core Location API 15
- core location framework
 - about 70
 - course 75
 - device GPS method 222
 - direction, heading information used 75
 - geocoding 74
 - heading 75

- location debugging 70, 71
- magnetometer method 222
- region monitoring 73, 74
- reverse geocoding 74, 75
- services 72
- significant change 73
- standard location 72, 73
- core location manager**
 - about 76
 - GPS, used by commercial 13
 - GPS, used by government 13
 - GPS, used by US military 13
 - location service availability, checking for 77-79
 - methods 76
 - user authorization, using 80, 81
- CoreLocationManager Delegate directive** 40
- core motion**
 - about 229
 - accelerometer 229, 230
 - CMAccelerometerData object 229
 - CMDeviceMotion object 229
 - CMGyroData object 229
 - CMMagnetometerData object 229
 - conclusion 235
 - device motion 229, 230
 - device motion data 235
 - gyroscope 229, 230
 - magnetometer 229, 230
 - manager framework 229
 - pull method 230
 - push method 230
 - using, steps 230
- CORS**
 - about 286
 - URL 286
- course**
 - about 75, 222
 - used, for direction 226
 - used, for direction in app 226-228
- course property** 84
- Cross-Origin Resource Sharing Requests.** *See* CORS
- currentLocationLabel variable** 328
- custom data, property** 241
- custom map annotations** 123-125

D

- damaged parking category**
 - hash tag 322
- dashcode** 30
- Delegate function** 71
- Department of Defense.** *See* DOD
- desiredAccuracy properties** 72
- desiredAccuracy values** 73
- detailNews, Second View Controller instance variable** 308
- Detect Location button** 43, 72, 75, 78
- device motion**
 - about 229, 230
 - deviceMotionActive properties 230
 - deviceMotionAvailable properties 230
- deviceMotionActive properties** 230
- deviceMotionAvailable properties** 230
- device motion data** 235
- DidEndDocument method** 95
- didEndElement method** 154, 169
- didEnterRegion delegate method** 297
- didEnterRegion method** 88, 90
- didExitRegion delegate method** 297
- didExitRegion method** 90
- didFailWithError event** 87
- didFinishLaunchingWithOptions method** 165
- didReceiveData method** 94, 100
- didSelectRowAtIndexPath method** 202, 303, 330
- didStartElement method** 94, 154, 169
- didUpdateHeadingmethod** 224
- didUpdateLocation method** 115
- didUpdateToLocation delegate method** 186
- didUpdateToLocation event method** 153
- didUpdateToLocation method** 87, 93, 112, 137, 138, 183, 227, 272, 274, 328
- didUpdateToLocationmethod** 224
- direction, course used**
 - about 226
 - in app 226-228
- direction, heading used**
 - about 222
 - app, preparing for direction 222
 - in app 223, 224, 225
 - magnetometer 222

directions

using, with locations 222

distanceFilter properties 72

distanceFilter property 72

distanceFromLocation method 85

Documents Tree View 148

DOD 13

Download button 148

draggable annotations 119-122

drainage category

hash tag 322

E

EKCalendar class 206

EKEvent class 206

EKEventEditViewController controller 210

EKEventEditViewController object 207

EKEventStore class 206

Email app 241

EndElement call 94

EndElement parser method 94

event categories

events display, filtering by 197-204

eventful

about 65

categories 197, 198

features 65

eventful.com 179

event kit framework

used, for adding events to iPhone calendar
205-209

Event Kit UI framework 205

events

adding to iPhone calendar, event kit framework
used 205-209

plotting, on map 191-196

retrieving, with SQLite 181-190

storing, with SQLite 181-190

Events:SQLite application 192

events app 179

events display

filtering, by event categories 197-204

events variable 92

exit trigger 299

Ext.util.GeoLocation package 62

F

Factual

URL 66

fireDate property 242

Firefox SQLite manager

URL 148

FirstViewController.h file 308

FirstViewController.m file 308

forecast

URL 149

format, taxonomy type 284, 285, 287

foundCharacters method 154, 169

Foursquare

about 63, 247

API consumption, use case 64

developers page, URL 64

functions supported, URL 64

HP Web OS 267

user authentication 277

foursquare API. *See* **foursquare venue API**

foursquareAuth class 277

foursquare authentication 269

foursquare client ID 248

foursquare search API 263-266

foursquare venue API

airport category 252

API calls, list 248

categories 248

categories, consuming 248-254

connectionDidFinishLoading method 251

consuming 248

getfoursquareCategories method 250

initializeDatabase method 250

INSERT statement 251

NSURLConnection call 250

popular venues 255-261

recommended venues 255-261

showCategoriesFromLocal function 252

UISegmentedControl instance 255

UITableView instance 255

venues, searching for 262

G

Galileo Positioning System 8

gamification strategy 247

- garbage category
 - hash tag 322
- Geo API Object** 65
- geocodeAddressDictionary:completionHandler**
 - method 74
- geocodeAddressString:completionHandler**
 - method 74
- geocodeAddressString:inRegion**
 - completionHandler method 74
- geocodes** 12
- geocoding**
 - about 12, 23, 74
 - geocodeAddressDictionary:completionHandler method 74
 - geocodeAddressString:completionHandler method 74
 - geocodeAddressString:inRegion completionHandler method 74
 - methods 74
- Geocoding API**
 - URL 23
- geo fencing** 12
- Geo Fencing support**
 - adding 295-298
- Geographic Information System.** *See* **GIS**
- geolocation getCurrentPosition function** 51
- geolocation.getCurrentPosition PhoneGap**
 - method 54
- Geolocation object** 51
- GeoNames API**
 - about 136, 142
 - CLPlacemark object 147
 - Documents Tree View 148
 - Download button 148
 - Firefox SQLite manager, URL 148
 - Hello Location SQLite example 148
 - Show Package Contents option 148
 - SQLite database editor, URL 148
 - URL 147
 - used, for coverting location data into city name 142-146
 - user_position table 148
 - Weather App project 148
- GeoTagging** 12
- getAccelerometerData method** 233
- getDatabaseFullPath method** 137
- getfoursquareCategories method** 250
- getNewsDetails:newsTitle; method** 308
- getSqliteLocation action** 152
- getSqliteLocation method** 140
- Gigaom**
 - URL 13
- GIS** 11, 12
- Git** 30
- Global Positioning System.** *See* **GPS**
- GLONASS** 8, 12
- glyphish-icons set** 327
- Google**
 - location based services, consuming 9-11
- Google Maps**
 - URL 26
- Google Maps API**
 - using, to detect location 23-25
- Gowalla** 64
- GPS**
 - about 8, 11, 17
 - used, by commercial 13
 - used, by government 13
 - used, by US military 13
- GPS, components**
 - control segment 17, 18
 - space segment 17, 18
 - user segment 17
- graffiti category**
 - hash tag 322
- gyroActive properties** 230
- gyroAvailable properties** 230
- gyroscope**
 - about 229, 230
 - gyroActive properties 230
 - gyroAvailable properties 230

H

- heading**
 - about 75, 222
 - used, for direction in app 223-225
- headingAccuracy property** 76
- headingAvailable method** 222
- headingFilter variable** 224
- health hazard category**
 - hash tag 322
- Hello_EventsViewController.h file** 183

- Hello_foursquareFirstViewController.h** file 249, 257
- Hello_foursquareFirstViewController.m** file 249
- Hello_foursquareSecondViewController.h** class 257, 264
- Hello_foursquareSecondViewController.h** file 256, 257
- hello location**
 - building, PhoneGap used 49-53
 - building, Titanium Appcelerator used 54-57
 - extending, for local search 99
 - extending, for nearby events 90-95
 - with Sencha Touch 59
- Hello Location SQLite example** 148
- Hello_LocationViewController** class definition 40
- Hello_LocationViewController.xib** file 223
- Hello_NewsViewController.h** file 289
- Hello_NewsViewController.m** file 290
- Heroku for Mobile** 318
- Home button** 235
- home screen**
 - default location, setting up 168, 169
 - defining 165
 - didFinishLaunchingWithOptions method 165
 - initWithNibName method 167
 - ViewController main file 166
 - viewDidLoad method 166
- horizontalAccuracy** property 84
- HP Web OS** 267
- HTML5**
 - about 12, 46
 - by Apple, URL 48
 - by Google, URL 48
 - by Mozilla Foundation, URL 48
 - features 46
 - markup 47
 - tags 46
- hyper local application** 283
- HyperLocal News**
 - about 283
 - Outside.in 283
- I**
- IBAction** 256
- iCloud** 31, 33
- IDE** 29
- id** parameter 276
- iMessage** 31, 33
- inactive, state** 237
- inCategories** flag 200, 202, 272
- inCheckin** flag 278
- indexPathForRowAtPoint** method 346
- indoor navigation** 26
- inEvents** flag 202
- Info.plist** file 222, 224, 235
- inForeCast** flags 161
- initializeDatabase** method 184, 250, 290
- initWithCoordinate:altitude**
 - horizontalAccuracy:verticalAccuracy:course:speed:timestamp: 85
- initWithCoordinate:altitude:horizontalAccuracy:verticalAccuracy:timestamp:** 85
- initWithLatitude**
 - longitude: method 85
- initWithNibName** method 167
- initWithURL:parameters:requestMethod:** 210
- inLiveWeather** flags 161
- inSearch** flag 265, 266
- insert** statement 293
- instantiateViewControllerWithIdentifier** method 312
- instruments tools**
 - about 31, 33, 225
 - automation 33
 - network connections instrument 33
 - system trace 33
- integrated build system** 31
- Integrated Development Environment.** *See* IDE
- interface builder** 30
- iOS 5**
 - about 31
 - features 31
- iOS 5, features**
 - air play, mirroring for iPad 2 32
 - calendar 32
 - camera and photo enhancements 32
 - computer-free operation 32
 - game center 32
 - iCloud 31, 33
 - iMessage 31, 33
 - mail 32
 - multitasking gestures 32

- newsstand 32
- notification center 32
- reminders 32
- safari 32
- Speech Recognition (Siri) 33
- Twitter 31
- twitter integration 32
- Wi-Fi Sync 32
- iOS 5 SDK**
 - and Xcode 4.2, new features 33, 34
 - new features 32, 33
- iOS application life cycle**
 - about 236-240
 - active, state 237
 - application
 - didFinishLaunchingWithOptions: method 238
 - applicationDidBecomeActive: method 238
 - applicationDidEnterBackground: method 238
 - applicationWillEnterForeground: method 238
 - applicationWillResignActive: method 238
 - applicationWillTerminate: method 238
 - background, state 237
 - inactive, state 237
 - methods 238
 - not running, state 237
 - states 237
 - suspended, state 237
- iOS device**
 - background location tasks 236
- iOS SDK**
 - core location framework 70
 - downloading, from Apples Developer Site 36
- iOS simulator 30**
- iPhone**
 - location track, turning off 15, 16
- iPhone app**
 - AR, adding 213-219
 - Twitter capabilities, adding 210-212
- iPhone calendar**
 - events adding, event kit framework used 205-209
- iphone-sdk 130**
- issue_category(varchar or integer) 317**
- issue_description 317**
- issue_id (integer) 317**
- issue_lat 317**

- issue_lon 317**
- issuesCategory array 336**
- issue_title(text or varchar) 317**
- items object 101**

J

- jsonContent variable 331**
- JSONObjectWithData method 101**
- JSON request 346**

K

- kCLAuthorizationStatusAuthorized status 81**
- kCLAuthorizationStatusDenied status 81**
- kCLAuthorizationStatusNotDetermined status 81**
- kCLAuthorizationStatusRestricted status 81**
- kCLLocationAccuracyBest, constant value 73**
- kCLLocationAccuracyBestForNavigation, constant value 73**
- kCLLocationAccuracyHundredMeters, constant value 73**
- kCLLocationAccuracyKilometer, constant value 73**
- kCLLocationAccuracyNearestTenMeters, constant value 73**
- kCLLocationAccuracyThreeKilometers, constant value 73**

L

- Last.fm API 65, 97, 98**
- Latitude text field 78**
- latitudeText outlet 42**
- Layar**
 - about 213
 - URL 213
- Layar Player 213**
- Layar Player SDK**
 - URL, for downloading 213
- LBA 12**
- LBS**
 - about 7, 8
 - consuming, with Google 9-11
 - example 8
 - requisites 8
 - used, in iPod Touch 14

- uses 13
- libsqlite3.0.dylib library** 136
- live fix it** 31
- live weather**
 - URL 149
- LLDB Debugger** 31
- LLVM compiler** 33
- loadingIcon** 153
- localNotification object** 242, 243
- local notifications**
 - about 241
 - adding, to WeatherPackt application 246
 - UINavigationController class 241
 - using 241-245
- local search** 247
- local search app**
 - building, with foursquare 100-102
 - hello location, extending 99
- Location-Based Advertising.** *See* LBA
- location-based**
 - APIs 63
 - applications 63
 - events and entertainment 63
 - SDKs 63
 - search, for stories 285
- Location-Based Services.** *See* LBS
- location data**
 - converting, into city name 142
- locationDetect** 227
- locationDetect function** 82
- location detection**
 - Skyhook Wireless' location 19
 - through Google Maps API 23-25
 - through SkyHook Wireless Loki framework 22, 23
 - through Wi-Fi 19
- locationDetect method** 85, 112, 115, 118
- Location Labs**
 - about 66
 - URL 66
- locationManager:didEnterRegion** 74
- locationManager:didExitRegion** 74
- locationManager:didUpdateToLocation:fromLocation method** 85
- LocationManager object** 41, 86
- location, Patch News API**
 - name-based search 286

- location services**
 - pull service 19
 - push service 18
- locationServicesEnabled method** 80
- location tracking**
 - in iPhone, turning off 15, 16
- Loki.com** 23
- Longitude text field** 78

M

- magneticHeading property** 76
- magnetometer**
 - about 222, 229
 - magnetometerActive properties 230
 - magnetometerAvailable properties 230
- magnetometer** 230
- magnetometerActive properties** 230
- magnetometerAvailable properties** 230
- map**
 - dynamics, adding 197
 - events, plotting 191-196
- map annotations**
 - customizing 126-128
- map.centerCoordinate method** 115
- Map Co-Ordinates**
 - to Map Points 108
 - to Points 108
- map geometry** 106, 107
- MapKit framework**
 - about 105
 - capabilities 106
 - conversion functions 108
 - co-ordinates 108
 - Map co-ordinate system 108
 - map geometry 106, 107
 - Mercator projection 107
 - using, in app 108-[114]
- map overlays**
 - about 126
 - map annotations, customizing 126-128
 - uses 126
- Map Points**
 - to Map Co-Ordinates 108
 - to Points 108
- maps**
 - annotating 117

- annotations, adding 117-119
 - panning 115
 - zooming 115
- mapView:viewForAnnotation delegate function 121
- mapView class 133
- MapView component 306
- Master Control Station. *See* MCS
- MCS 17
- MD5 signature 292
- Mercator projection model 107
- Message App 241
- Michael Pidwirny 107, 108
- MKAnnotation protocol 117, 126
- MKAnnotationView class 117, 119
- MKCircleView 126
- MKCoordinateRegion object 111
- MKMapView component 129
- MKMapViewDelegate delegate 121
- MKMapView object 110
- MKOverlayView, subclasses
 - MKCircleView 126
 - MKPolygonView 126
 - MKPolylineView 126
- MKPointAnnotation object 117, 120, 197
- MKPolygonView 126
- MKPolylineView 126
- MKReverseGeocoder 74
- MKUserTrackingModeFollow 129
- MKUserTrackingModeFollowWithHeading 129
- monitoredRegions property 74
- motion manager
 - using, to access accelerometer data 231-234
- MutableString variable 92

N

- names-based search
 - for location 286
- network connections instrument 33
- newsstand 32
- noise category
 - hash tag 322
- no-special character based keyword 198
- notification center 32
- notification type, property 241

- not running, state 237
- NSArray directoryPath variable 137
- NSArray variable 252
- NSDateFormatter class instances 259
- NSDocumentDirectory parameter 137
- NSJSONReadingAllowFragments option 100
- NSJSONSerialization class 100, 102, 312
- NSJSONSerialization object 183
- NSString object 81
- NSString variable 199
- NSURLConnection 153
- NSURLConnection call 250
- NSURLConnection class 91, 93, 98
- NSURLConnection object 183, 292
- NSURLConnection request 259
- NSURLCoonection 93
- NSURLRequest 93
- NSUserDefaults class 169
- NSXMLParser class 91, 94, 169
- Nuance Mobile SDK 174
 - documentation, URL 175
- NULL value 282
- numberOfRowsInComponent method 336
- numberOfRowsInSection delegate method 325

O

- objectAtIndex 191
- objectForKey 191
- Object-relational mapping. *See* ORM
- offline map
 - about 130
 - creating 133
- offline-online app 180
- onDeviceReady function 51
- onError function 52
- onSuccess function 52
- OpenGL ES Debugging 33
- OpenStreetMaps
 - using, with CloudMade API 131-133
- ORM 90
- others category
 - hash tag 322
- outdoor navigation 26
- Outside.in 283

P

PacktEvents

- about 179, 180
- app, building 219
- architecture 180
- events, retrieving with SQLite 181-190
- events, storing with SQLite 181-190
- Nuance Speech Mobile SDK using 180
- offline-online app 180
- tabs 180

PacktLocal app

- about 278
- add venue endpoint, implementing 282
- building 268-282
- UI, building 267
- venue information, saving on device 268

PacktNews application

- building 299-312

Palm WebOS. *See* HP WebOS

panning

- achieving, ways 115

parserDidEndDocument method 169

Patch.com 284

Patch News API

- about 283
- author type, taxonomy type 287, 288
- components 284
- consuming 286-294
- format, taxonomy type 287
- HyperLocal News 283
- vertical, taxonomy type 286, 287

Patch News API, components

- authentication 284
- stories, finding by location 285
- stories, finding by names 286
- taxonomy 284

performRequestWithHandler method 343

PhoneGap

- about 48
- used, for building WeatherPackt application 174
- using, to build hello location app 49-53

photo enhancement 32

Points

- to Map Co-Ordinates 108
- to Map Points 108

PostGIS

- URL 296

potholes category;hash tag 322

presentLocalNotificationNow method 243

process device motion data. *See* device motion data

ProgrammableWeb

- about 135
- URL 135

Pull method 230

pull service 19

Push method 230

push notifications 240

push service 18

pushViewController method 304, 312

R

RAD 30

Radio signals. *See* RFID

Rapid Application Development. *See* RAD

Read Accelerometer button 234

readCategoriesFromApi method 199

readCategoriesFromLocal method 199, 201

readEventFulApi method 183, 186, 198

readEventsFromLocal method 189, 194

readNews method 290, 292

region.center property 115

region.identifier property 299

region monitoring

- about 73, 74
- locationManager:didEnterRegion 74
- locationManager:didExitRegion 74
- monitoredRegions property 74
- startMonitoringForRegion:desiredAccuracy method 74
- startMonitoringForRegion method 74
- stopMonitoringForRegion method 74

reminders 32

repeatInterval property 243

Research and markets

- URL 13

resignFirstResponder method 265

returnCategoryIdForName:categoryName method 199

returnCategoryIdForName:categoryName method 202

- returnCategoryIdForName** method 203
- retweet** 331
- reverseGeocodeLocation:completionHandler** method 75
- reverse geocoding**
 - about 12, 74, 75
 - CLGeocoder class 75
 - reverseGeocodeLocation:completionHandler** method 75
- RFID** 26
- Right Detail** attribute 304
- Root.plist** file 170
- Root.string** file 170
- Round Rect Button** 40
- Russian Global Navigation Satellite System.** *See* GLONASS

S

- SAAS** 63
- Safari** 32
- scheduled time, property** 241
- scheduleLocalNotificaton** method 243
- scheme** 35
- science geek**
 - URL 106
- Search API** 262
- searchBar:textDidChange** method 265
- searchBarCancelButtonClicked** method 265
- searchBarSearchButtonClicked** method 265
- searchBarTextDidBeginEditing** method 265
- searchBarTextDidEndEditing** method 265
- searchForVenues** method 265
- Second View Controller** instance variable 308
- SeeClickFix**
 - URL 316
- segues** 299
- select statement** 252
- Sencha Touch**
 - about 59
 - for standalone HTML5 web apps 59-63
 - URL 63
- setCenterCoordinate** method 115
- setRegion:animated** method 115
- setRegion** method 115
- Settings Bundle** 170
- setUserTrackingMode** method 129

- sharedApplication** method 243
- showAlert** method 153
- showCategoriesFromLocal** function 252
- showForeCast** method 153, 159
- showLiveWeather** action 152
- showLiveWeather** method 153
- Show Map** button 110
- showNearbyVenues** method 272
- Show Package Contents** option 148
- showPopularVenue** method 258
- showTrendingVenues** method 261
- showVenuesFromLocal** method 273
- significant change location service**
 - about 73
 - startMonitoringSignificantLocationChanges** function 73
 - stopMonitoringSignificantLocationChanges** function 73
- SimpleGeo**
 - services 66
 - URL 66
- Siri** 33
- SkyHook Wireless** coverage
 - URL 19
- Skyhook Wireless' location** 19
- SkyHook Wireless Loki** framework
 - using, to detect location 22, 23
- Slider** object 226
- social governance** 316
- Software-as-a-Service.** *See* SAAS
- space segment, GPS** 18
- spatial database** 12
- Speech Recognition.** *See* Siri
- speed** property 84
- SQLite**
 - events, retrieving with 181-190
 - events, storing with 181-190
- sqlite3_exec()** method 138
- sqlite3_open** method 137
- sqlite3_step** method 141
- SQLite database**
 - used, to retrieve user location 140, 141
 - used, to store user location 136-140
- SQLite database editor**
 - URL 148
- Stackmob**
 - about 318

- city-based Issues object 319
- StackMob**
 - URL 317
- Stackmob, city-based Issues object 319**
- standalone HTML5 web apps**
 - Sencha Touch for 59-63
- standard location**
 - about 72, 73
 - CLLocationManager object 72
 - CLLocation object 72
 - desiredAccuracy values 73
 - distanceFilter property 72
 - startUpdatingLocation function 72
 - stopUpdatingLocation function 72
- startAccelerometerUpdates method 233, 234**
- startAccelerometerUpdatesToQueue method 233, 234**
- StartElement call 94**
- startMonitoringForRegion:desiredAccuracy method 74**
- startMonitoringForRegion method 88**
- startMonitoringSignificantLocationChanges function 73**
- startMonitoringSignificantLocationChanges method 87**
- startUpdatingHeading method 76**
- startUpdatingLocation function 72**
- startUpdatingLocation method 41, 78, 85**
- stepperChanged IBAction 243**
- stopAccelerometer function 233**
- stopAccelerometerUpdates method 230, 233**
- Stop button 234**
- stopDeviceMotionUpdates 230**
- stopGyroUpdates 230**
- stopMagnetometerUpdates 230**
- stop methods**
 - about 230
 - stopAccelerometerUpdates 230
 - stopDeviceMotionUpdates 230
 - stopGyroUpdates 230
 - stopMagnetometerUpdates 230
- stopMonitoringForRegion method 74**
- stopMonitoringSignificantLocationChanges function 73**
- stopNotifications action 243**
- stopNotifications, UIButton variable 241**
- stopUpdatingHeading 76**

- stories, Patch News API**
 - location-based search 285
- storyboarding**
 - about 33, 254, 299
 - PacktNews application, building 300-312
- storyboards 334**
- street cleaning category**
 - hash tag 322
- street light category**
 - hash tag 322
- subCategories variable 252**
- suspended, state 237**
- switchVenueType 256**
- switchVenueType IBAction 261**
- system trace 33**

T

- Tabbed Application template 164, 248, 323**
- TableView controller 303, 304**
- taxonomy, Patch News API**
 - about 284
 - author type 284
 - format type 284
 - vertical type 284
- taxonomy table 290**
- Ti.Geolocation.getCurrentPosition method 57**
- timestamp property 76, 84**
- Titanium Appcelerator**
 - used, for building hello location 54-57
- Titanium.UI.createWindow method 57**
- Titanium UI framework (Ti.UI) 58**
- titleForHeaderInSection method 97**
- titleForRow method 336**
- Ti.UI.createView method 57**
- tracking modes**
 - about 129
 - MKUserTrackingModeFollow 129
 - MKUserTrackingModeFollowWithHeading 129
- traffic light category**
 - hash tag 322
- trueHeading property 76**
- TweetGovern**
 - about 316
 - avatar icon 322
 - categories 322
 - category table 318
 - Google AdMob SDK, adding 322

- hash tag 322
- icon 322
- issue, creating 335-345
- issues table 317
- issue, voting for 335-345
- MapKit framework, including 322
- nearby issues, showing 326-333
- search, adding 346
- twitter approach used 317
- UI, creating 323-325
- user location, detecting 326-333
- TweetGovern, category table**
 - category_id (integer) 318
 - category_name(text or varchar) 318
- TweetGovern, issues table**
 - issue_category(varchar or integer) 317
 - issue_description 317
 - issue_id (integer) 317
 - issue_lat 317
 - issue_lon 317
 - issue_title(text or varchar) 317
- TweetGovernSecondViewController class 331, 333**
- tweets array 333**
- Twitter**
 - about 31
 - capabilities, adding to iPhone app 210-212
- Twitter Account Store Object 343**
- twitter framework**
 - about 210
 - in iOS 5 210
 - TWRequest class 210
 - TWTweetComposeViewController class 210
- Twitter Search API 319**
 - @tweetgovern handle 320
 - TWTweetComposeViewController dialog-box 320
 - URL 321
- TWRequest class 210, 346**
- TWRequest object 343**
- TWTweetComposeViewController class 210**
- TWTweetComposeViewController dialog-box 320**
- TWTweetComposeViewControllerResultCancelled 211**
- TWTweetComposeViewControllerResultDone 211**

U

- Ui**
 - creating, for TweetGovern 323-325
- UIApplication Delegate object 237**
- UIApplicationMain function 237**
- UIApplication method 240**
- UIBackgroundModes key 235, 236**
- UIButton variable**
 - stopNotifications 241
- UIImage class 97**
- UINavigationController class**
 - custom data, property 241
 - notification type, property 241
 - properties 241
 - scheduled time, property 241
- UINavigationController object 323**
- UINavigationController class 312**
- UINavigationController item 323**
- UIPickerView object 336, 338**
- UIRequiredDeviceCapabilitieskey 222**
- UIResponder class 265**
- UISearchBarDelegate protocol 265**
- UISearchBar variable 263**
- UISegmentedControl 258**
- UISegmentedControl instance 255, 258**
- UISlider object 223**
- UIStepper object 241, 243**
- UITableView Cell Clicked event 98**
- UITableView class 98**
- UITableViewDataSource delegate 151, 184**
- UITableView didSelectRowAtIndexPath method 97**
- UITableView instance 255**
- UITableView object 183**
- UITableView variable 91**
- UITableView class 346**
- UITextField variable 231**
- UITextView elements 305**
- UIViewController class 271**
- UIViewController subclass option 335**
- user authorization**
 - authorizationStatus method 81
 - CLLocationManager class 81
 - using, for location 80, 81
- user location**
 - detecting 326-333

- didUpdateToLocation method 137, 138
- GeoNames API 136
- getDatabaseFullPath method 137
- getSqliteLocation method 140
- latitude value 141
- libsqlite3.0.dylib library 136
- Location update example 136
- longitude value 141
- NSArray directoryPath variable 137
- NSDocumentDirectory parameter 137
- retrieving, SQLite database used 140, 141
- sqlite3_exec() method 138
- sqlite3_open method 137
- sqlite3_step method 141
- storing, SQLite database used 136-140
- userLocation object 81**
- userLocation variable 257**
- user_position table 148**
- User Position table 182**
- user segment, GPS 17**
- Use StoryBoard option 323**

V

- venue API 248**
- Venue class 272-275**
- venues**
 - about 262
 - URL 262
- venuesearchBar.text property 265**
- venueTypeControl 258**
- verticalAccuracy property 84**
- vertical, taxonomy type 284-286**
- ViewController main file 166**
- ViewController.m file 232**
- viewDidLoad function 41, 84, 122**
- viewDidLoad method 77, 81, 82, 86, 88, 92, 111, 118, 123, 127, 132, 153, 166, 183, 199, 227, 232, 234, 242, 258, 265, 276, 290, 296, 312, 327, 331, 336**
- ViewDidLoad method 153, 227**
- viewForAnnotation delegate method 123**
- viewForAnnotation method 121**
- viewForOverlay delegate method 128**
- View object 194**

- vote button 340**
- voteForTweet IBAAction 341**
- voteForTweet method 340**

W

- W3C 48**
- Weather Alert function**
 - URL, for creating 162
- weather alerts**
 - URL 149
- weather app**
 - about 135
 - URL 135
- Weather App project 148**
- WeatherBug API**
 - about 148
 - app, running in emulator 160
 - array, declaring 151
 - didEndElement method 154
 - didStartElement method 154
 - didUpdateToLocation event method 153
 - documentation, URL 148
 - forecast, URL 149
 - foundCharacters method 154
 - functions, declaring 151
 - getSqliteLocation action 152
 - inForeCast flags 161
 - inLiveWeather flags 161
 - live weather, URL 149
 - loadingIcon 153
 - NSURLConnection 153
 - registration, URL 148
 - showAlert method 153
 - showAlerts action 152
 - showForeCast method 153
 - showForeCast method, defining 159
 - showLiveWeather action 152
 - showLiveWeather method 153
 - UITableViewDataSource delegate 151
 - using 150-154, 160
 - variables, declaring 150
 - variables, declaring for forecast variables 151
 - ViewDidLoad method 153
 - weather alert function creating, URL 162
 - weather alerts, URL 149

- Weather Bug Live Weather API, calling 153
- Weather queries 148
- weather services 149
- Weather Bug Live Weather API, calling 153**
- WeatherPackt application**
 - about 162
 - alerts page, adding 173
 - App icon 162
 - Apple's rich documentation, URL 170
 - building, with PhoneGap 174
 - default background images (launch images) 163
 - didEndElement method 169
 - didStartElement method 169
 - enabled_preference 171
 - foundCharacters method 169
 - IBAction 175
 - local notifications, adding 246
 - NSUserDefaults class 169
 - NSXMLParser class 169
 - Nuance Mobile SDK, adding 174
 - parserDidEndDocument method 169
 - Root.plist file 170
 - Root.string file 170
 - Settings Bundle 170
 - settings page 169
 - Tabbed Application template 164
 - text-to-speech 174-176
 - UIButton 175
 - weather API, formatting for display 169
- WebGL 46**
- Wi-Fi**
 - using, to detect location 19
- WiFiSLAM 26**
- Wi-Fi Sync 32**
- Worldwide Web Consortium. *See* W3C**
- X**
- X-axis deviation 226**
- Xcode**
 - interface builder 31
 - toolkit 30
 - workspace window 31
 - Xcode IDE 31
- Xcode3, transitioning from**
 - about 34-36
 - hello location app, building 38-45
 - iOS Developer Account, signing up for 36
 - iOS SDK, installation 37
 - Xcode 4.2 for Snow Leopard, downloading 36
 - Xcode, installation 36
- Xcode 4**
 - about 29
 - features 30, 31
 - prerequisites 30
- Xcode 4.2**
 - and iOS 5 SDK, new features 33, 34
- Xcode 4.2, features**
 - about 33
 - ARC 33
 - instruments tools 33
 - LLVM compiler 33
 - location simulation 33
 - OpenGL ES Debugging 33
 - storyboard 33
- Xcode 4, features**
 - assistant editor 30
 - build system, integrated 31
 - DashCode 30
 - instruments tool 31
 - integrated documentation 31
 - interface builder 30
 - iOS Simulator, with Location Simulation 30
 - live fix it 31
 - LLDB Debugger 31
 - tools 31
 - URL 31
 - version editor 31
- Xcode IDE 31**
- Xcode project**
 - App icon 162
 - default background images (launch images) 163
 - starting 162
 - Tabbed Application template 164
- xmlContent method 92**
- x property 76**
- XVALUE outlet 225**

Y

Yahoo! Query Language. *See* YQL

y property 76

YQL 66

YVALUE outlet 225

Z

zoomEnabled property 115

zoom- in feature 334

zooming

about 115-117

controlling 115

zoom-out feature 334

z property 76

ZVALUE outlet 225



**Thank you for buying
iPhone Location Aware Apps by Example
Beginner's Guide**

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.PacktPub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



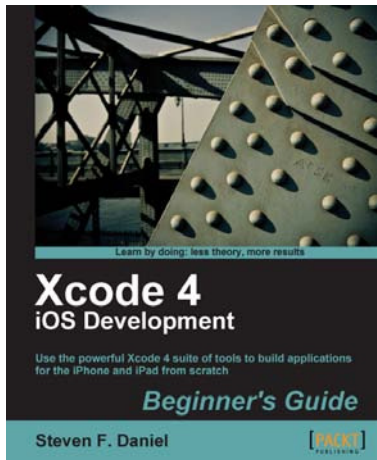
iPhone Applications Tune-Up

ISBN: 978-1-84969-034-8

Paperback: 256 pages

High performance tuning guide for real-world iOS projects

1. Tune up every aspect of your iOS application for greater levels of stability and performance
2. Improve the users' experience by boosting the performance of your app
3. Learn to use Xcode's powerful native features to increase productivity
4. Profile and measure every operation of your application for performance



Xcode 4 iOS Development Beginner's Guide

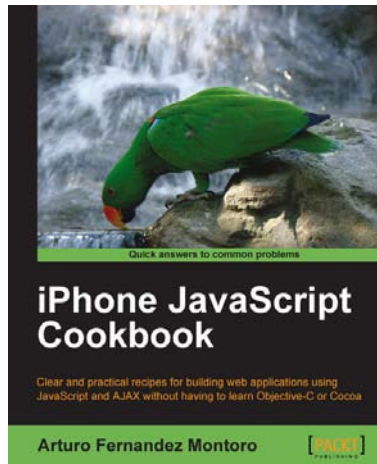
ISBN: 978-1-84969-130-7

Paperback: 432 pages

Use the powerful Xcode 4 suite of tools to build applications for the iPhone and iPad from scratch

1. Learn how to use Xcode 4 to build simple, yet powerful applications with ease
2. Each chapter builds on what you have learned already
3. Learn to add audio and video playback to your applications
4. Plentiful step-by-step examples, images, and diagrams to get you up to speed in no time with helpful hints along the way

Please check www.PacktPub.com for information on our titles



iPhone JavaScript Cookbook

ISBN: 978-1-84969-108-6

Paperback: 328 pages

Clear and practical recipes for building web applications using JavaScript and AJAX without having to learn Objective-C or Cocoa

1. Build web applications for iPhone with a native look feel using only JavaScript, CSS, and XHTML
2. Develop applications faster using frameworks
3. Integrate videos, sound, and images into your iPhone applications
4. Work with data using SQL and AJAX
5. Write code to integrate your own applications with famous websites such as Facebook, Twitter, and Flickr



Core Data iOS Essentials

ISBN: 978-1-84969-094-2

Paperback: 340 pages

A fast-paced, example-driven guide to data-driven iPhone, iPad, and iPod Touch applications

1. Covers the essential skills you need for working with Core Data in your applications
2. Particularly focused on developing fast, light weight data-driven iOS applications
3. Builds a complete example application. Every technique is shown in context
4. Completely practical with clear, step-by-step instructions

Please check www.PacktPub.com for information on our titles

