



Community Experience Distilled

Learning iOS 8 for Enterprise

Design and develop stunning iOS applications for
business environments

Mayank Birani

[PACKT]
PUBLISHING

Learning iOS 8 for Enterprise

Design and develop stunning iOS applications
for business environments

Mayank Birani



BIRMINGHAM - MUMBAI

Learning iOS 8 for Enterprise

Copyright © 2014 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: December 2014

Production reference: 1201214

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-182-9

www.packtpub.com

Credits

Author

Mayank Birani

Project Coordinator

Harshal Ved

Reviewers

Nguyen Thai Duong
Vivek Pandya
Shuichi Tsutsumi

Proofreaders

Stephen Copestake
Ameesha Green

Commissioning Editor

Usha Iyer

Indexer

Monica Ajmera Mehta

Acquisition Editor

Neha Nagwekar

Production Coordinator

Nilesh R. Mohite

Content Development Editor

Shubhangi Dhamgaye

Cover Work

Nilesh R. Mohite

Technical Editors

Shali Deeraj
Dennis John

Copy Editors

Karuna Narayanan
Laxmi Subramanian

About the Author

Mayank Birani has 3 years of experience in the software industry and a strong association with the technical industry. After several years of programming experience in different programming languages, he started developing applications for iOS devices. He started software development while he was pursuing his graduation and was really interested in learning new technologies. He then joined a software company and started developing iOS applications for them, focusing on real content-based iOS applications.

I would like to send a special thanks to my mom, Neelu Birani, and dad, Prakash Birani, for their relentless efforts in assisting me in every way imaginable as well as for helping me keep my life together. Finally, I would like to thank all my friends for sharing my happiness when I started working on this project and encouraging me when it seemed too difficult to be completed.

About the Reviewers

Nguyen Thai Duong is 29 years old. He lives in Hanoi with his wife and daughter. He likes computers and reads fairy tales, mystery science, and natural science. He started studying computer programming when he was 13 years old. He has a Bachelor's degree in Information Technology from Vietnam National University, Hanoi. He worked for DeNA, Vietnam (earlier Punch Entertainment), a mobile game development and distribution company, for 5 years. Currently, he is working for a start up, Gigatum. The company's product is Clingme – a location-based mobile application. You can download Clingme from the Apple Store or Play Store. Clingme is a kind of consumer social networking site such as Yelp or Dianping, but for Vietnam. Some of the company's other products include Vi-Chat (a Yahoo! instant-messenger client for MIDlet devices), Jacos2D-X (an open source game engine based on Cocos2d-x with V8 JavaScript integration), V8 JavaScript library for iOS (open source MIT), and the SpriteBuilder library for Cocos2d-x (open source MIT). With a large population (about 92 million people in 2013) and more than 20 million people using smartphones, Nguyen hopes his business will be successful in Vietnam.

This is his first book.

I would like to thank Paushali Desai and Venitha Cutinho who trusted in me and introduced me to this project.

Vivek Pandya is a software developer and is passionate about Apple Technologies. He is also interested in algorithms and operating systems. He supports open source software and works for the Drupal project. His interests outside computers include fitness, trekking, and biology.

Shuichi Tsutsumi is a freelancer and an iOS developer since 2009 and has developed more than 30 iOS apps. Some of the apps that he developed includes the following:

- Bound Monsters, which he developed as a lead programmer and recorded over 2 million downloads in Japan.
- TAP NINJA, for which he was responsible for all of the implementation. This app was awarded Best of App Store 2012.
- Domino's App, for which he was the lead programmer. This app received various awards, including Bronze Lion at Cannes Lions 2011, Direct Lotus (Bronze), at ADFEST 2011, and was a Bronze Winner at London International Awards 2010.

He worked as an image processing engineer at Canon Inc. (2007 to 2009) where he designed image processing functions and invented patents for printers.

He also worked at NTT DATA Corporation (2003 to 2007) where he was involved in the R&D on speech recognition interface.

He completed his MS in Communications and Computer Engineering from the Graduate School of Informations, Kyoto University (2001 to 2003) and his BS in Computer Science, Kyoto University (1997 to 2001).

He has also worked on a book on iOS (*iOS Expert Recipes 100, Hidekazu system*) that can be found at <http://www.amazon.co.jp/dp/4798038180>.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

I would like to dedicate this book to my family, who gave me the courage and confidence to write this book and supported me throughout the process.

Table of Contents

Preface	1
Chapter 1: Getting Started with iOS	7
Interface and implementation	9
Code snippets for interface and implementation	9
Types of methods in iOS	9
Creating objects	10
Important datatypes	10
Array	11
NSArray	11
NSMutableArray	11
Property and Synthesize	11
Delegates	13
Building our first iPhone app	15
Summary	24
Chapter 2: Exploring More UI Components	25
Adding UI components programmatically	25
Adding a view	26
Adding a label	27
Creating a new button	28
Some featured UI components	29
The map view	29
UIPickerView	32
The web view	34
The image view	36
Using the image view	36
Understanding the anatomy of the table view	38
Important methods for the table view	38
Working with the table view	39
Scroll view and its usage	42

Table of Contents

Navigation controller	44
Summary	48
Chapter 3: Exploring Various Frameworks	49
Exploring various UI components with libraries	53
Database integration	58
SQLite	59
Core Data	63
Social integration in our application	71
Summary	76
Chapter 4: APIs Introduced in iOS 7	77
Using AirDrop	78
SpriteKit	84
The iOS native game framework	84
Text Kit	97
NSTextStorage	98
NSLayoutManager	98
NSTextContainer	98
Kerning	99
Ligatures	99
Line breaking	100
Justification	100
Summary	106
Chapter 5: Frameworks Introduced with iOS 8	107
Working with PhotoKit	107
Photos Framework	107
Photos UI	108
Features of PhotoKit	109
Handoff for seamlessly resuming activities	115
Compatibility with Handoff	115
App framework support	116
Handoff interactions	116
Implementing Handoff directly	117
Creating the user activity object	117
Specifying an activity type	117
Populating the activity object's user info dictionary	119
Adopting Handoff in responders	119
Continuing an activity	119
Native App-to-Web Browser Handoff	121
Web Browser-to-Native App Handoff	122
Using continuation streams	123
Summary	124

Table of Contents

Chapter 6: Using iCloud and Security Services	125
Working with iCloud	125
Keychain Services	135
Encryption and decryption	136
iOS keychain concepts and structure	136
Understanding the application flow	136
The Touch ID API	146
Touch ID through touch authentication	147
Touch ID through Keychain Access	148
Using the Local Authentication framework	148
Summary	149
Chapter 7: The App-distribution Program	151
Setting up a developer account	151
Getting started	152
Joining the iOS Developer Program	158
Getting started with the iOS Developer account	166
The iOS Provisioning Portal	168
iTunes Connect	169
Certificates, Identifiers, and Profiles	170
Submitting the app	191
Submitting with Xcode	193
Summary	195
Index	197

Preface

This book will help you get started and understand the development of an enterprise application for iOS. It serves as a step-by-step guide to help you to understand the core concepts of iOS application development. It provides a deeper understanding of the way to develop and deploy your apps. It has seamlessly organized content that will help you learn, in an incremental way, how to enhance your conceptual and programming skills.

This book will also help you build fundamental logic skills and teach you ways to handle architecture-related problems for big enterprise applications. The book also includes API integration, social media integration, and integration of third-party frameworks. This will enable you to understand how to make your app easily customizable using other third-party frameworks. Such content is the heart of this book and makes it rich in content.

The main focus is on making you read and implement the details of this book. Throughout this book, you will find activities and minor projects that teach you the core concepts and logic. This technique of teaching makes this book special for you and will help you find it interesting, as there is always something to play with. At the end of the book, you will have some beautiful working applications with their implied logic.

What this book covers

Chapter 1, Getting Started with iOS, teaches you the basics of iOS, such as methods, arrays, properties, delegates, and so on. After going through this chapter, you will be able to use Xcode and create simple apps using the UI component.

Chapter 2, Exploring More UI Components, teaches you about components and how to create them programmatically. It will give you an understanding about the concept of table views and teach you how to use them. Also, we will make a simple application using the Navigation Controller.

Chapter 3, Exploring Various Frameworks, provides some more details on the UI and teach you how to integrate frameworks in your projects. We will also create a simple Student Registration app by exploring each section. By the end of the chapter, you will have created a fully loaded app with UI and functionality. You will also learn about databases with social media integration.

Chapter 4, APIs Introduced in iOS 7, comprises a lot of fun stuff, including games, putting an image between text, and sharing with AirDrop. All these APIs are introduced in iOS 7. After this chapter, you should try to extend all the activities mentioned in it; this will make these concepts more clear for you.

Chapter 5, Frameworks Introduced with iOS 8, discusses the new iOS 8 APIs and a little code snippet of Swift. In this chapter, you will learn about the PhotoKit framework and Handoff, with some more code snippets.

Chapter 6, Using iCloud and Security Services, teaches you how to push your data to iCloud and save your private data, such as passwords, account numbers, and ATM pins, to Keychain. We will also focus on the Touch ID API, which was introduced in iOS 8.

Chapter 7, The App-distribution Program, teaches you to set up and understand your Developer account. You will also learn how to set up a provisioning profile to publish your app on the Store, and discuss the anatomy of an enterprise-level distribution using the Apple Enterprise account.

What you need for this book

You'll need the following things to get started with writing applications for iOS devices:

- An Intel-based Macintosh running Mavericks (OS X 10.9.5 or later)
- Xcode
- You must be enrolled as an iPhone developer in order to test the example projects on your device
- An iOS device running iOS 7.0 and above

Who this book is for

If you are an iPhone application developer or even if you are a beginner, this book will help you explore your technical skills. The phrases and code are written in such a manner that even beginners will understand them. This book will be very helpful for those who want to learn about the new frameworks of iOS 7 and iOS 8, along with their activities and uses. This book will be an aid for students and also for experienced iOS developers.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:
"In the AppDelegate.h file, find the application:didFinishLaunchingWithOptions: method."

A block of code is set as follows:

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {  
    ...  
    // Register for push notifications  
    [application registerForRemoteNotificationTypes:  
        UIRemoteNotificationTypeBadge |  
        UIRemoteNotificationTypeAlert |  
        UIRemoteNotificationTypeSound];  
    ...  
}
```

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

```
curl -s https://www.parse.com/downloads/cloud_code/installer.sh | sudo /  
bin/bash
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Start your Xcode. Navigate to **File** | **New** | **Project**."



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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in 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 at 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 from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/1829OT_GraphicsBundle.pdf.

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 could 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/submit-errata>, 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.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Getting Started with iOS

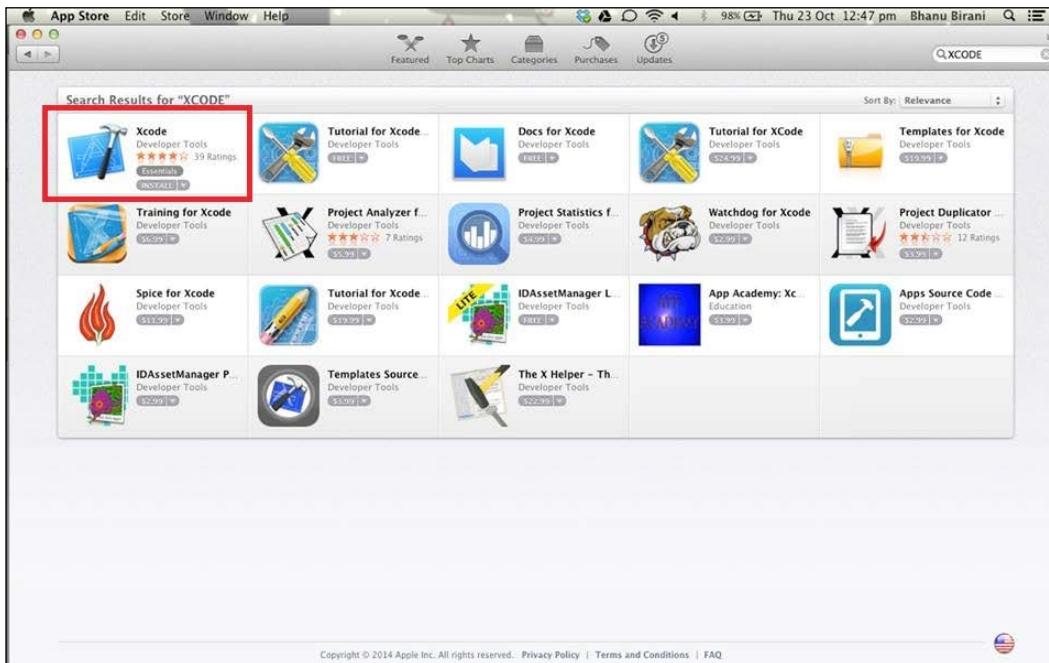
The mobile industry is the fastest growing domain among other IT domains. iOS plays a key role in the mobile industry; nowadays, even the key IT players are incorporating mobility in the enterprise way. This title will help you understand the basics of iOS development and its implementation from the perspective of enterprises.

To get started with iOS development, we will need the following things:

- A Mac system
- Xcode
- The iOS SDK

Basically, iOS development is based on the Objective-C language. Objective-C is an extension of the C programming language; this includes the **OOP (object-oriented programming)** concept and adds small-talk style messaging to C. Xcode is built to help you build great apps for iPad, iPhone, and Mac. Xcode is an **IDE (Integrated Development Environment)** for iOS.

You can download Xcode from the Apple store as shown in the following screenshot:



The Xcode option

It is free; after downloading Xcode, it automatically appears in the launch pad. Xcode provides different features such as coding, design, a user interface, and testing for you.

In this chapter, we will cover the following topics:

- Interface and implementation classes
- Types of methods in iOS
- Datatypes
- Arrays
- Property and synthesize
- Delegates
- Building a simple app

Interface and implementation

In Objective-C, there are two types of files that represent a single class: one is the interface file and the other one is the implementation file. In the interface file, the declaration of a method and variable is done; in the implementation file, we define this method and use the declared variable.

Code snippets for interface and implementation

The following is the code snippet for the interface class:

```
@interface MyClass: NSObject{  
    // declare class variables here  
}  
    // class methods and instance methods are declared here  
@end
```

The `MyClass.h` file is an interface class. `NSObject` is a root class of all the classes, and it is a must to import the `NSObject` class. The `@end` keyword indicates that our interface block is completed.

The following is our implementation class named as `MyClass.m`:

```
@implementation MyClass  
    // class methods are defined here  
@end
```

Types of methods in iOS

The methods that we declare in the interface file are defined in the implementation file. Methods are declared in the following way:

```
- (returnType)methodName: (typeName) variable1 : (typeName)variable2;
```

There are two methods in iOS: one is the class method and the other is the instance method. In the class method, we do not have to create an object; we can directly access these methods using their class names. The class method is indicated by the `+` symbol. We can find static methods in languages such as Java and C++, and we will find class methods in languages such as Objective-C and Ruby. The difference between the static and class methods is that they have different language concepts.

The principal differences are:

- Static methods are shared between all instances (this doesn't exist in Objective-C).
- The class method is a method on a class. In languages such as Objective-C and Ruby, a class itself is an instance of another class (metaclass). Using `+` before a method declaration means that the method will be defined on the class. Technically, it's just an instance method on a different object. The syntax for the class method is as follows:

```
+ (void) classMethod;
```

In the instance method, we have to create an object. We can't access the instance method without creating an object. Memory is allocated for an object. The `alloc` keyword is used to allocate memory. The following syntax is used for the instance method:

```
- (void) InstanceMethod;
```

Creating objects

An object is created in the `implementation` file as follows:

```
MyClass *object= [[MyClass alloc] init] ;  
[object InstanceMethod];
```

Here, we used the `alloc` and `init` keywords. The `alloc` keyword is used to allocate memory to the object, and the `init` keyword is used to initialize that object.

Important datatypes

Like any other programming language, iOS also has different datatypes such as `int`, `float`, `double`, `char`, and `id`. Datatypes are used to specify the kind of data that is being stored in a variable. There are four important datatypes in iOS:

- `NSString`: This is used to represent a string
- `NSInteger`: This is used to declare an integer
- `CGFloat`: This is used to declare float values
- `BOOL`: This is used to declare a Boolean (yes or no) value

Array

Array is a collection of homogeneous datatypes with contiguous memory allocation. In iOS, arrays are of two types:

- NSArray
- NSMutableArray

NSArray

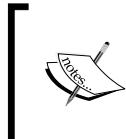
NSArray is an immutable array. In Objective-C, by default, arrays are immutable, that is, as the name indicates, its object can't be changed or removed after the initialization of the array:

```
NSArray *xyz = [[NSArray alloc] init];
xyz = @[@"Harry", @"Tom", @"Jack"];
```

NSMutableArray

NSMutableArray is a subclass of NSArray. This is a modifiable array, and its object can be removed or modified after the initialization of array:

```
NSMutableArray *xyz = [[NSMutableArray alloc] init];
xyz = @[@"Harry", @"Tom", @"Jack"];
```



The concept of String is the same as in C. String also has two types: NSMutableString and NSString. Consider the following example for the NSString type:

```
NSString *myName = @"Jack";
```



Property and Synthesize

The properties of an object are defined to let other objects use or change their state. However, in object-oriented programming, it's not possible to access the internal state of an object directly from outside the class (except public accessors). Instead, accessor methods (getters and setters) are used to interact with the objects. The goal of the `@property` is to make it easier to create and configure properties by automatically generating these accessor methods:

- `@property`: This method implements the setter/getter methods in our code automatically; we don't have to write the code manually.

- `@synthesize`: This method synthesizes the properties with the given attributes, and the compiler will generate the setter and getter methods for our variables. However, now we do not use `synthesize`; instead of `@synthesize`, we use an underscore (`_`) or the `self` keyword.

Let's understand these methods with a code snippet. This is the interface file:

```
#import <Foundation/Foundation.h>
@interface MyClass : NSObject
@property void methodname;
@end
```

This is the implementation file:

```
#import "MyClass.h"
@implementation Class
@synthesize methodname = _methodname;
@end
```

In Objective-C, every object holds a reference count. When an object is created, its reference count increases by one; when it releases the object, the reference count decreases by one. When the reference count reaches zero, it deallocates the memory by itself. The attributes of `@property` is as follows:

- `atomic`: By default, every property is atomic. It will ensure that a whole value is always returned by the getter method or set by the setter method. Only a single thread can access a variable to get or set a value at a time. So, `atomic` is also thread-safe.
- `nonatomic`: In `nonatomic`, there is no guarantee that the value returned from a variable is the same one that the setter method sets. At the same time, more than one single thread can access a variable at a time.
- `strong`: The `strong` attribute owns the object. The compiler will ensure that any object that we assign to this property will not be destroyed as long as we (or any other object) point to it with a strong reference.
- `weak`: In a `weak` reference, we don't want to have control over the object's lifetime. The object we are referencing weakly only lives on because at least one other object holds a strong reference to it.
- `retain`: This specifies that `retain` should be invoked on the object upon assignment. It takes ownership of an object.
- `assign`: This specifies that the setter uses simple assignment. It uses an attribute of the scalar type, such as `float` or `int`.
- `copy`: This copies an object during assignment and increases the retain count by one.

Consider the following simple example using the attributes:

```
@property (nonatomic, assign) float radius;
@property (atomic, strong) NSString *name;
```

Delegates

A delegate is a tool through which one object can communicate with another; in turn, the objects can stay connected to each other. It is a method by which one object can act on behalf of another object. The delegating object keeps a reference to the other object and, at the appropriate time, sends a message to it. The message informs the delegate of an event that the delegating object is about to handle or that is to be handled.

Consider the following example of how to use a delegate.

Let's define the `FirstViewController.h` interface file as follows:

```
#import "SecondViewController.h"

@interface FirstViewController : UIViewController
<SecondViewControllerDelegate>
{
    IBOutlet UITextField *userNameTextField;
}

@property (nonatomic, strong) UITextField *userNameTextField;

-(IBAction)goNext:(id)sender;

@end
```

Now, let's define the `FirstViewController.m` implementation file:

```
#import "FirstViewController.h"

@interface FirstViewController ()

@end

@implementation FirstViewController

@synthesize userNameTextField;

-(void)goNext:(id)sender{
    SecondViewController *secondVC = [[SecondViewController alloc]
init];
```

```
secondVC.delegate = self;
[self.navigationController pushViewController:secondVC
animated:YES];
}

-(void)done:(NSString*)name{

NSLog(@"%@", @"BACK in firstVC");
userNameTextField.text = name;
}

@end
```

Next, we will define the `SecondViewController.h` interface file as follows:

```
#import "FirstViewController.h"

@protocol SecondViewControllerDelegate <NSObject>

-(void)done:(NSString*)someText;

@end
@interface SecondViewController : UIViewController{

IBOutlet UITextField *someText;
IBOutlet UIButton *returnButton;
id delegate;
}

@property (assign, nonatomic) id <SecondViewControllerDelegate>
delegate;
@property (strong, nonatomic) UITextField *someText;

-(IBAction)goBack:(id)sender;

@end
```

Now, we will define the `SecondViewController.m` implementation file as follows:

```
#import "SecondViewController.h"

@interface SecondViewController ()

@end

@implementation SecondViewController

@synthesize someText;
@synthesize delegate = _delegate;
```

```
- (void)goBack:(id)sender{  
  
    [self.delegate done:someText.text];  
  
    FirstViewController *firstVC = [[FirstViewController alloc] init];  
  
    [self.navigationController pushViewController:firstVC  
    animated:YES];  
}  
  
@end
```

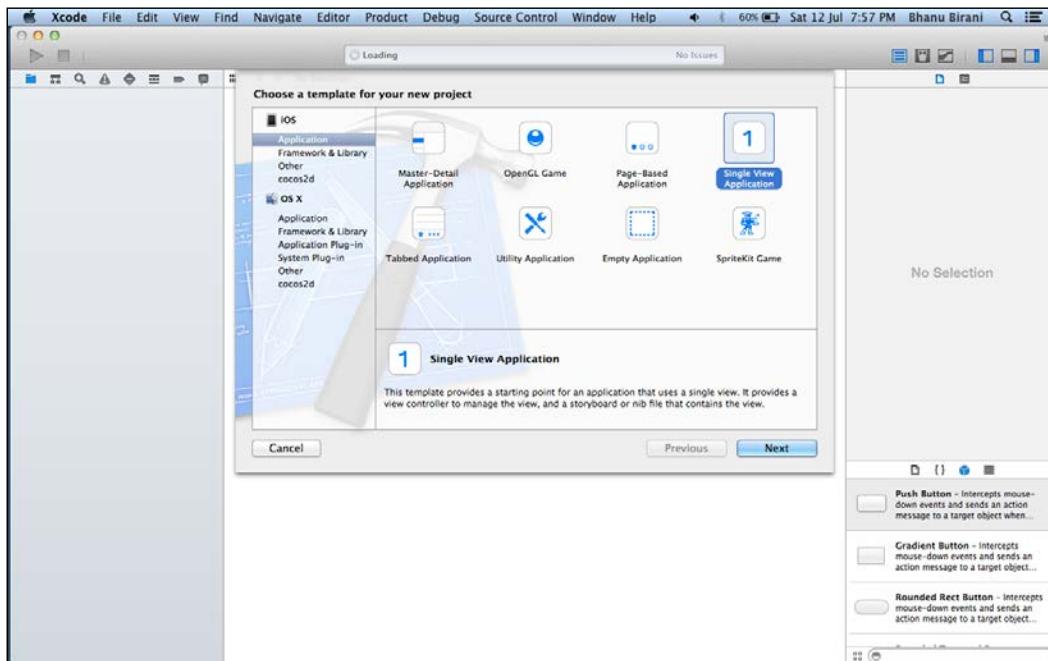
Building our first iPhone app

Let's make our first iPhone app by performing the following steps:

1. Open Xcode; you will see the following screen. In the panel on the right-hand side, you can see your existing projects. You can open your project directly by selecting from the list of projects; for a new project, on the other hand, select **Create a new Xcode project**.



2. There are eight templates provided by Xcode. In the panel on the left-hand side, you can see there are two options: **iOS** and **OS X**. **iOS** is for Apple touch devices, and **OS X** is for desktop devices. Initially, choose **Single View Application**. Then, click on **NEXT**, as shown in the following screenshot:

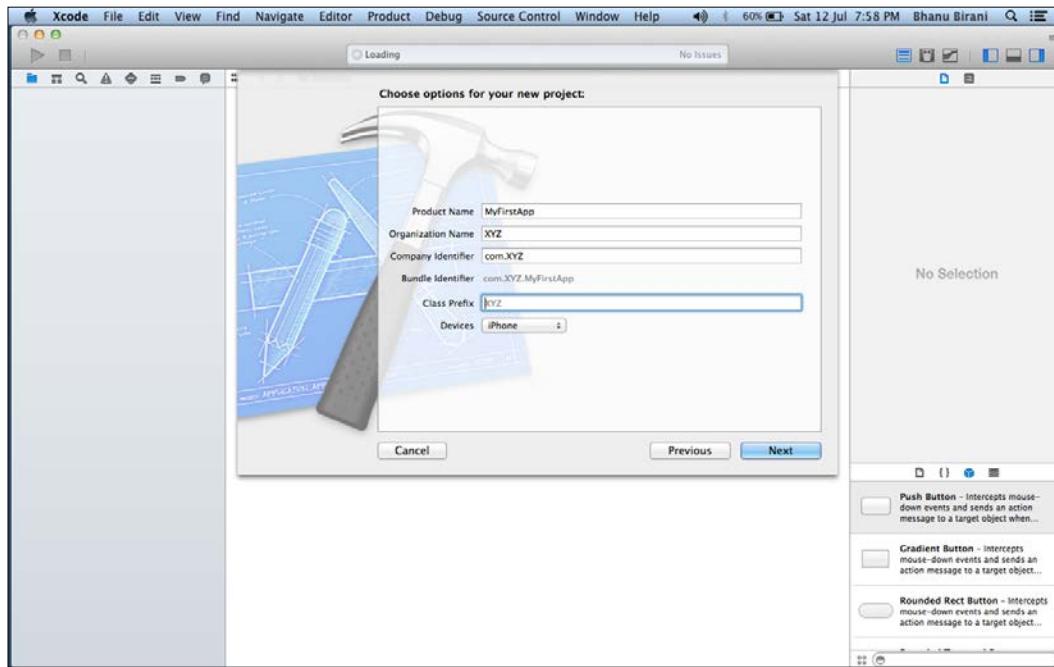


3. Now, it is time to give a name to your project and choose **iPhone** from the **Devices** dropdown. Then, click on **Next**. The following screenshot will appear:

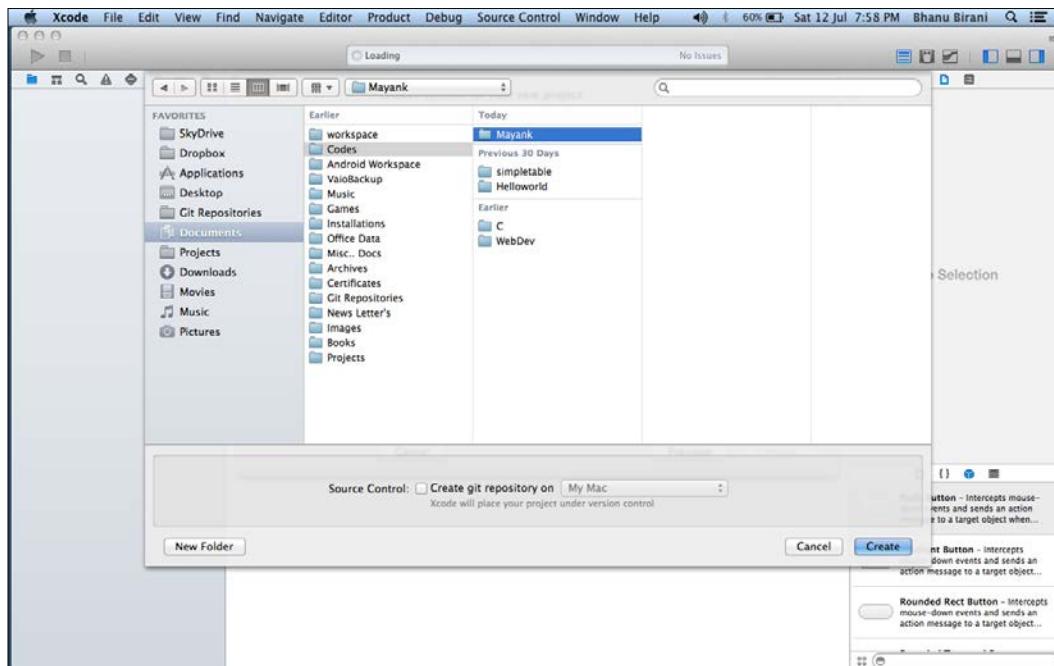


Downloading the example code

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.

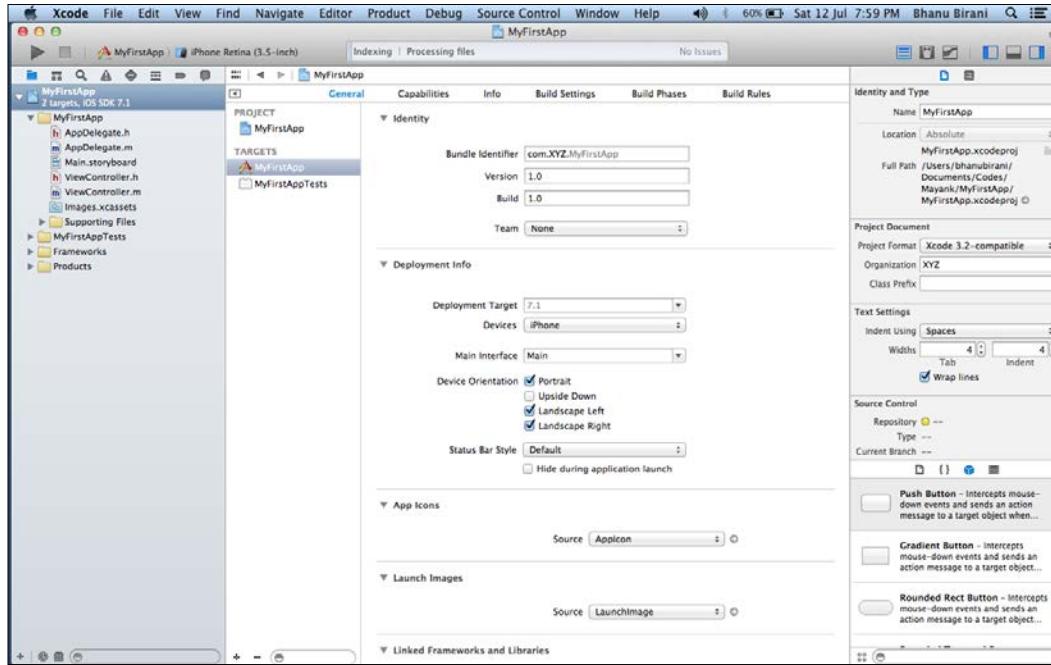


4. Save your project in the directory of your choice:



Getting Started with iOS

- Now, your editor will look like the following screenshot. In the left-hand-side panel, there is a declaration of classes. Select the storyboard from the left-hand-side panel. Storyboard provides the view to your application.

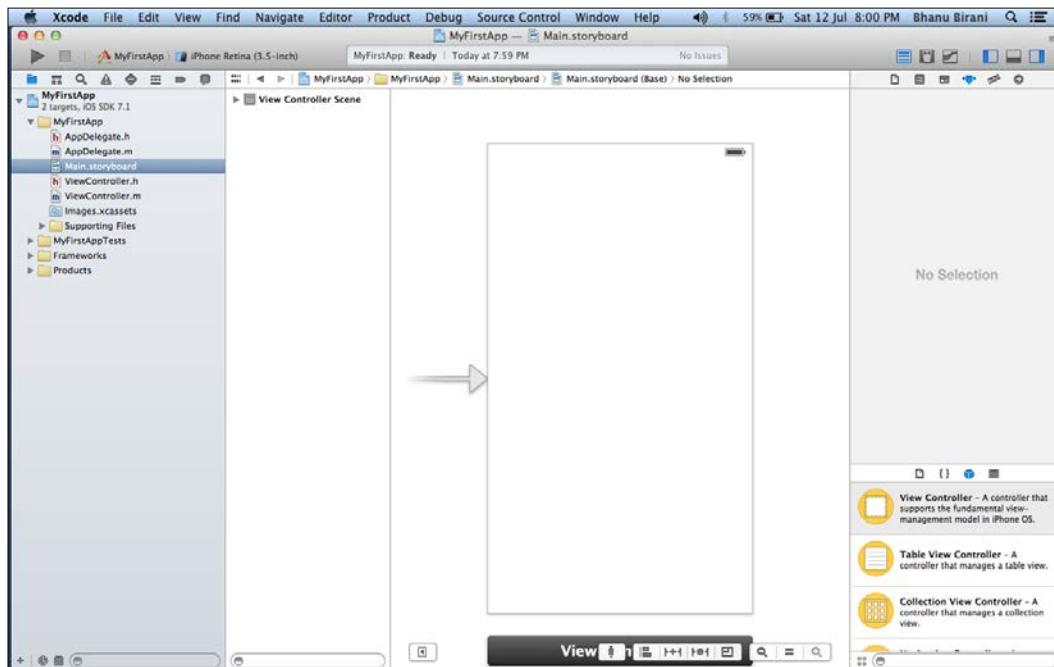


This is your storyboard. At present, it is an empty view controller. Storyboard has many areas, such as the navigation area, editor area, utility area, and debug area, that are described as follows:

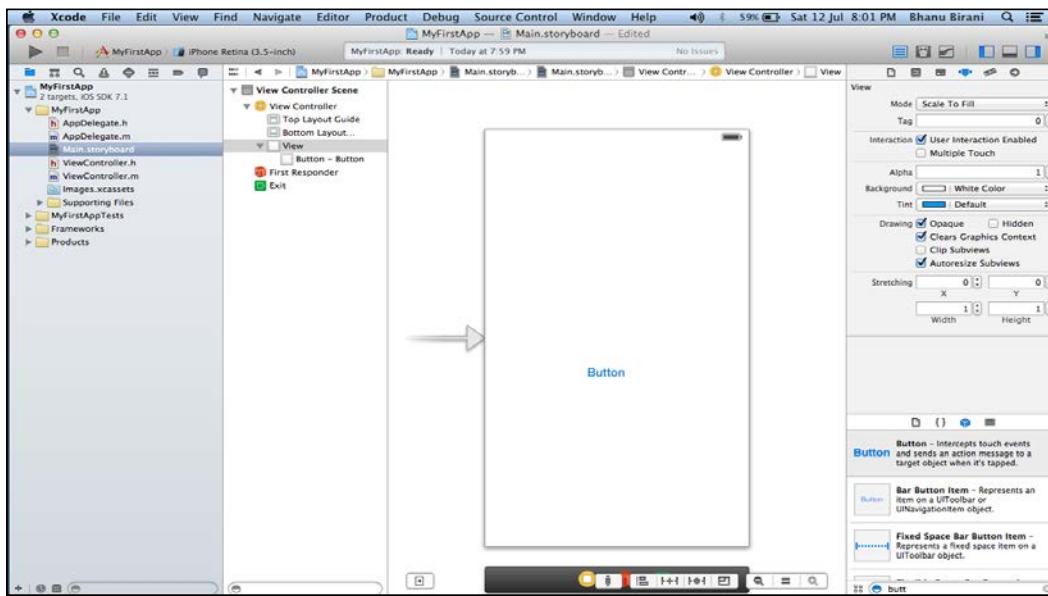
- Navigation area:** In this pane, there are various navigators that we can switch between using the Navigator selector bar. The three navigators that we will use a lot are the Project, Search, and Issue navigators.
- Editor area:** The editor area is where we'll probably be spending most of our time! This is where all the coding happens.

- **Utility area:** The Xcode utility area comprises two panes: the Inspector pane and the Library pane. The Inspector pane will give us details about the file. However, when we are looking at the storyboard, the Inspector pane will show us the different attributes that you can modify for a selected element. The Library pane won't be very useful until we look at a storyboard. When we use the Interface Builder part of Xcode, we can drag UI elements from the Library pane onto the editor area to add them to our user interface.
- **Debug area:** The debug area will show us the console output and the state of various variables when you run your application.

In the following screenshot, we can see an arrow before the view; this indicates that the view is a starting view of the application. When the application launches, this view will launch first:



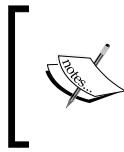
- On the right-hand side, a lot of components, such as button, label, and text fields, are present (we will learn about these components in the upcoming chapters). Drag-and-drop a button from the right-hand-side panel as shown in the following screenshot. Rename it by double-clicking on it; give it any name. For example, name it Hello.



This is all our UI part. Now, let's move on to the coding section.

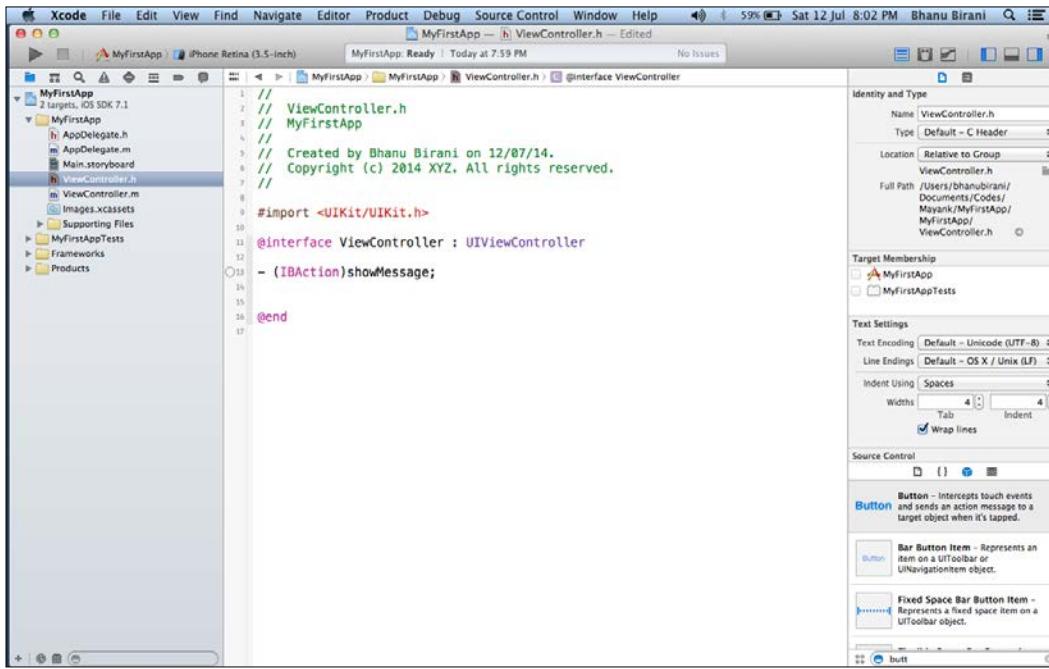
- Go to `ViewController.h` and write the following method in it. The `.h` file is an interface file of our project where we declare the property and method. If we want to declare variables, then they are declared under the braces of the interface; the property and methods are defined outside the braces:

```
@interface ViewController : UIViewController
{
    Int x;
}
@property (nonatomic, strong) NSString*recipeName;
```



Coming back to our program, we have to describe one method, `showMessage`, where we describe the `UIAlertView` function; this appears as a pop-up window with a message.

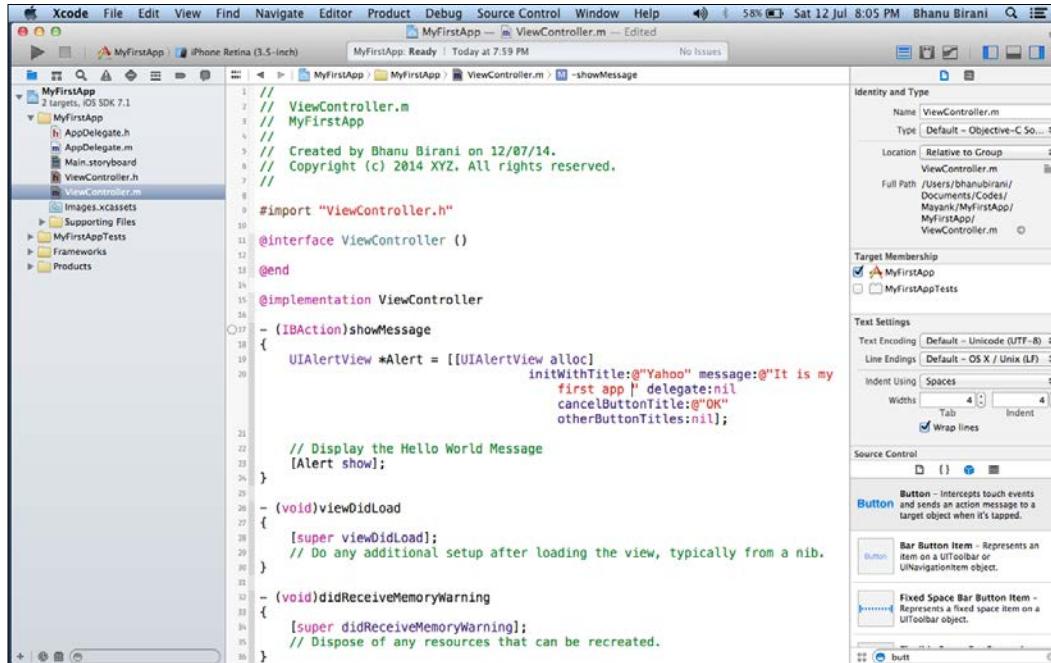
The following screenshot shows the editor area with the preceding code snippet:



Let's understand the code line by line:

- `#import <UIKit/UIKit.h>`: This is a header file that we import in our code. UIKit is a framework that contains all the inbuilt library files for the UI part. The UIKit header file imports all the other header files available in the UIKit framework; after importing this header file, we don't have to import other UIHeader files such as `UIViewController.h`, `UIView.h`, or `UIButton.h` manually.
- `@interface ViewController : UIViewController`: This is an interface for the `ViewController.h` class. It inherits the `UIViewController` class, which is used to handle the flow of our screen or view.
- `(IBAction) showMessage`: This is a method that we created manually. When we want to perform some action on tapping a button, we will use `IBAction`. This is a kind of return type in iOS. Here, the name of the method is `showMessage` (we can give any name). `IBAction` tells the UI Builder that the method can be used as a selector (event receiver).
- `@end`: This code indicates that our interface part is over.

8. Now, go to `ViewController.m` and describe the method that is defined in the `.h` file. This is also called an **implementation** file. The following screenshot illustrates the code used in the implementation file:



```
// View Controller.m
// MyFirstApp
// Created by Bhangu Birani on 12/07/14.
// Copyright (c) 2014 XYZ. All rights reserved.

#import "ViewController.h"

@interface ViewController : UIViewController

@end

@implementation ViewController

- (IBAction)showMessage {
    UIAlertView *Alert = [[UIAlertView alloc]
        initWithTitle:@"Yahoo" message:@"It is my
        first app" delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
    // Display the Hello World Message
    [Alert show];
}

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

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

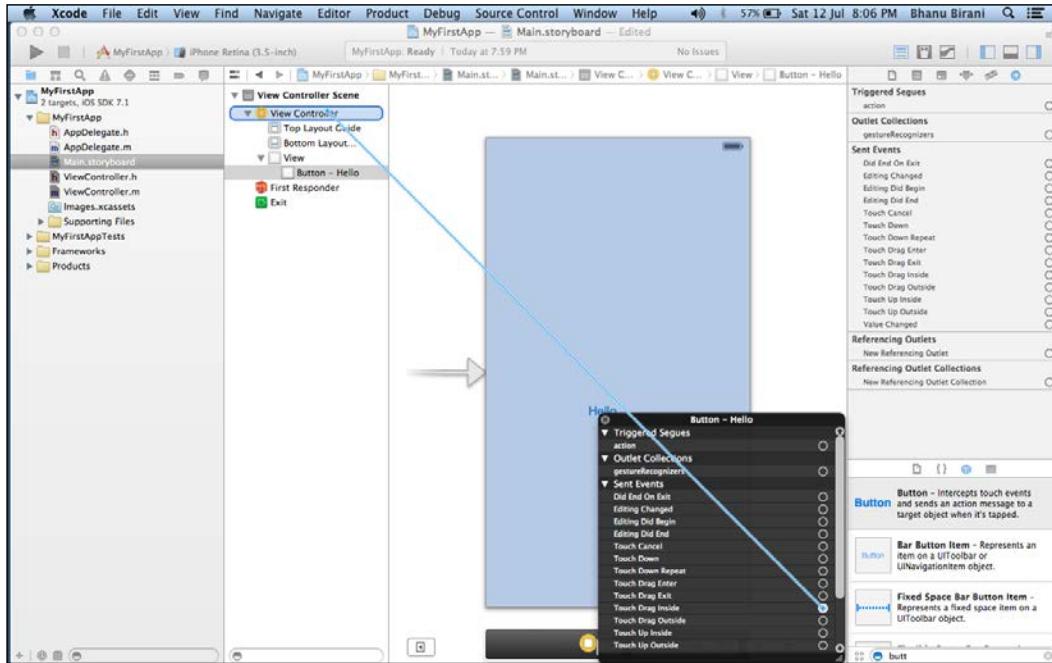

```

Once again, let's understand the code line by line:

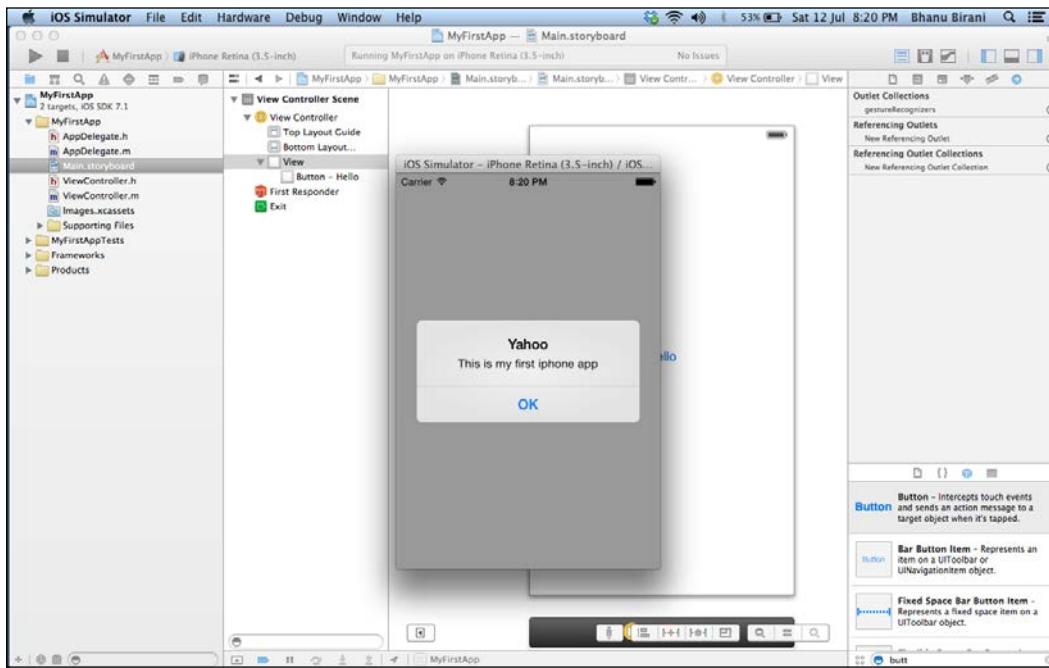
- `#import "ViewController.h"`: This imports our `.h` class (interface class).
- `@implementation ViewController`: From this snippet, our implementation of the method starts. The method that we declare in the interface file will be implemented here. It also contains some inbuilt methods such as `- (void)viewDidLoad`, `- (void) didReceiveMemoryWarning`, and so on. We can code inside these methods as needed. If we choose the **Empty application** template, then these inbuilt methods won't be provided.
- `UIAlertView`: Alert views are the pop-up views or messages that appear over the current view. We can use this by making an object of it. Here, `Alert` is an object. `alloc` is a keyword use to allocated memory for an object.
- `[Alert show]`: This snippet is used to display the pop up over the screen.

Now we have to connect this button to the method that we declare. Without connecting the button, it won't work. Use the following steps to connect the button.

- Right-click on the button. A black pop-up window will appear. Select **Touch Up Inside** and connect it to **View Controller**, as shown in the following screenshot. After releasing the mouse, select **showMessage** from the pop up.



10. Now, your button is connected to the `Alert` method. This is the time to execute your project. Run your project and click on the **Hello** button. Your output will look like the following screenshot:



Summary

In this chapter, we learned the basics of iOS, such as methods, arrays, properties, delegates, and so on. After this chapter, we will also be able to use Xcode, and we can make simple apps using the UI component. In the next chapter, we will learn more about components.

2

Exploring More UI Components

UI Elements is the visual part that we can see in our applications. These elements respond to user interactions such as buttons, text fields, and other labels. Some UI elements are used to make our graphical application; examples include images, pickers, Map kit, and many more. **Xcode** helps you to build many interfaces using an interface builder. The **UIKit** framework provides the classes needed to construct and manage an application's components or user interface. The **UIKit** framework is responsible for handling **UIComponents**, managing views and windows, and creating connections between components and code. UI elements can be used in two ways:

- By dragging-and-dropping them from the interface builder
- By programmatically adding the components to the view

We will cover the following topics in this chapter:

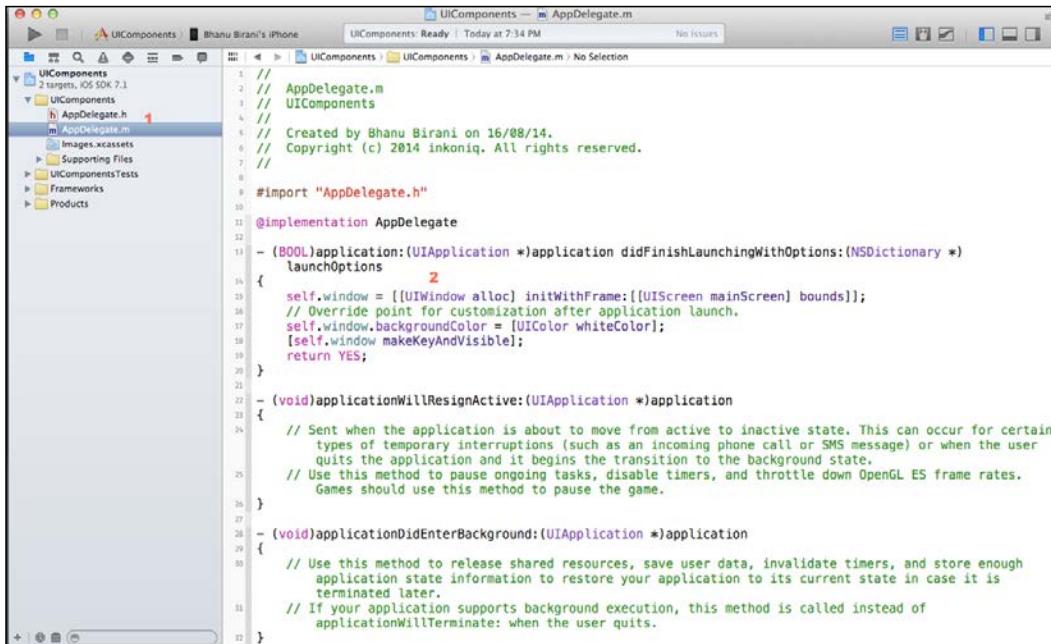
- Adding UI components programmatically
- Some featured UI components
- Understanding the anatomy of Table view
- Scroll view and its usage
- Navigation controller

Adding UI components programmatically

As mentioned in the introduction, we can add the components through coding. Let's begin with some simple examples.

Adding a view

In Xcode, on the left-hand panel, you can find the `AppDelegate.m` file as shown in point 1 of the following screenshot. Navigate to the file. There is a lot of code, but we will focus on only one function, `application:didFinishLaunchingWithOptions`, as shown in point 2 of the following screenshot:



The screenshot shows the Xcode interface with the project 'UIComponents' selected. In the left sidebar, the file 'AppDelegate.m' is highlighted with a red circle labeled '1'. In the main editor area, the code for the `didFinishLaunchingWithOptions` method is visible, with a red circle labeled '2' next to the opening brace of the method definition.

```
// AppDelegate.m
// AppDelegate
// Created by Bhana Birani on 16/08/14.
// Copyright (c) 2014 inkonig. All rights reserved.

#import "AppDelegate.h"

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}

- (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, this method is called instead of applicationWillTerminate: when the user quits.
}
```

Just add the following code in the function (`didFinishLaunching`) to make it look similar to following:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    UIView *view = [[UIView alloc] initWithFrame:self.window.bounds];
    [view setBackgroundColor:[UIColor blueColor]];
    [self.window addSubview:view];

    [self.window makeKeyAndVisible];
    return YES;
}
```

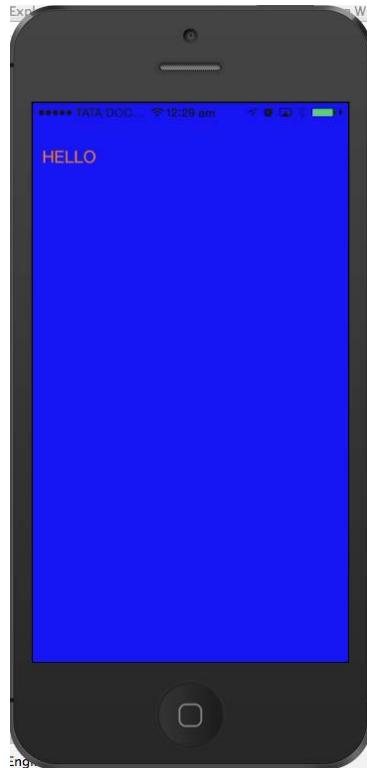
Build and run the program; if you see the blue screen on the simulator, it means we have completed it. In the same way, we can code for all the UI components. You can find some more examples in this chapter to learn faster.

Adding a label

To add a subview, you have to code in the `AppDelegate.m` class, as done in the previous case. Here is the code that should be added to the same function after the line `[self.window addSubview: view];`:

```
CGRect labelFrame = CGRectMake( 10, 40, 100, 30 );
UILabel* label = [[UILabel alloc] initWithFrame: labelFrame];
[label setText: @"HELLO"];
[label setTextColor: [UIColor orangeColor]];
[view addSubview: label];
```

In previous code, we created one object of the `UILabel` class. Allocate memory and initialize the object. Then, we can set the text and text color by using the inbuilt methods. Finally, add the label on the view. The following screenshot displays the label named **HELLO** on the screen:



Creating a new button

To add a new button, write the following code after the line `[view addSubview:label]`, and run it:

```
CGRect buttonFrame = CGRectMake( 10, 80, 100, 30 );
UIButton *button = [[UIButton alloc] initWithFrame: buttonFrame];
[button setTitle: @"My Button" forState: UIControlStateNormal];
[button setTitleColor: [UIColor redColor] forState:
UIControlStateNormal];
[button addTarget:self action:@selector(buttonAction:) forControlEvents
s:UIControlEventTouchUpInside];
[view addSubview: button];
```

This code will create a button and the `buttonAction` method will be called on clicking the button. Here, the object of the `UIButton` class is `button`. Set the target of `button` to `self`; `@selector` calls our `buttonAction` method and sets our event to `TouchUpInside`. The `buttonAction` function should look something similar to the following:

```
- (void)buttonAction:(id)sender
{
    NSLog(@"%@",@"button clicked");
}
```

When you run the code, you will see the button just below the label. You can change the position by changing the coordination from the code. Now, you can try for each component one by one. The file should look like this after appending all of the previous code:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWith
hOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen
mainScreen] bounds]];

    UIView *view = [[UIView alloc] initWithFrame:self.window.bounds];
    [view setBackgroundColor:[UIColor blueColor]];
    [self.window addSubview:view];

    CGRect labelFrame = CGRectMake( 10, 40, 100, 30 );
    UILabel* label = [[UILabel alloc] initWithFrame: labelFrame];
    [label setText: @"HELLO"];
    [label setTitleColor: [UIColor orangeColor]];
    [view addSubview: label];

    CGRect buttonFrame = CGRectMake( 10, 80, 100, 30 );
    UIButton *button = [[UIButton alloc] initWithFrame: buttonFrame],
```

```

[button setTitle: @"My Button" forState: UIControlStateNormal];
[button setTitleColor: [UIColor redColor] forState:
UIControlStateNormal];
[button addTarget:self action:@selector(buttonAction:) forControlEvents:
UIControlEventsTouchUpInside];
[view addSubview: button];

[self.window makeKeyAndVisible];
return YES;
}

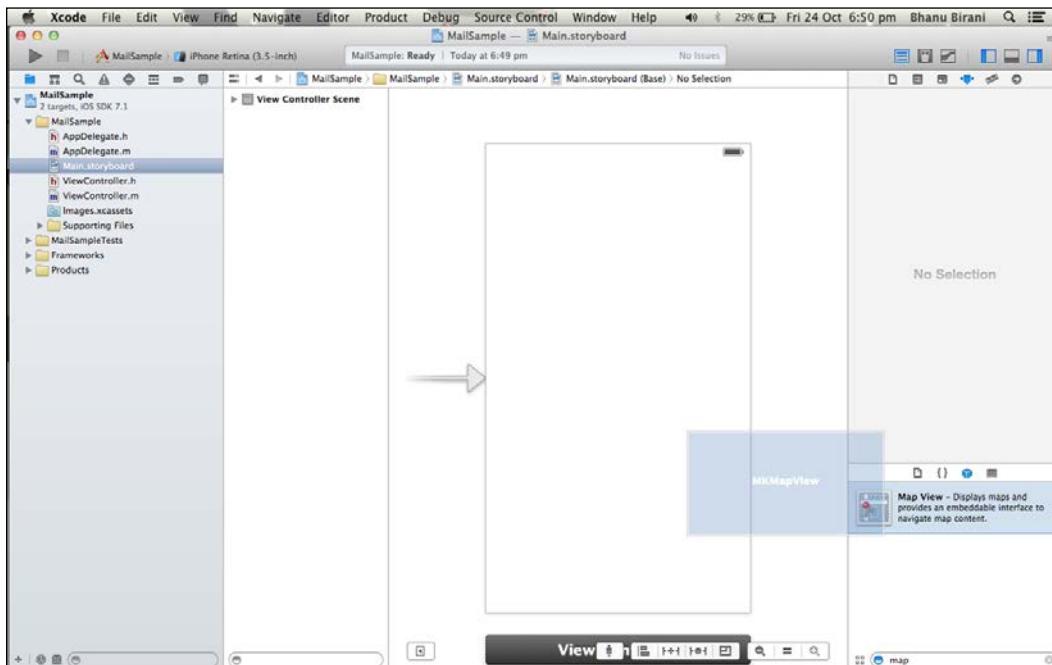
```

Some featured UI components

There are several new components in iOS and some changes in the existing components. Let's discuss these components in this section.

The map view

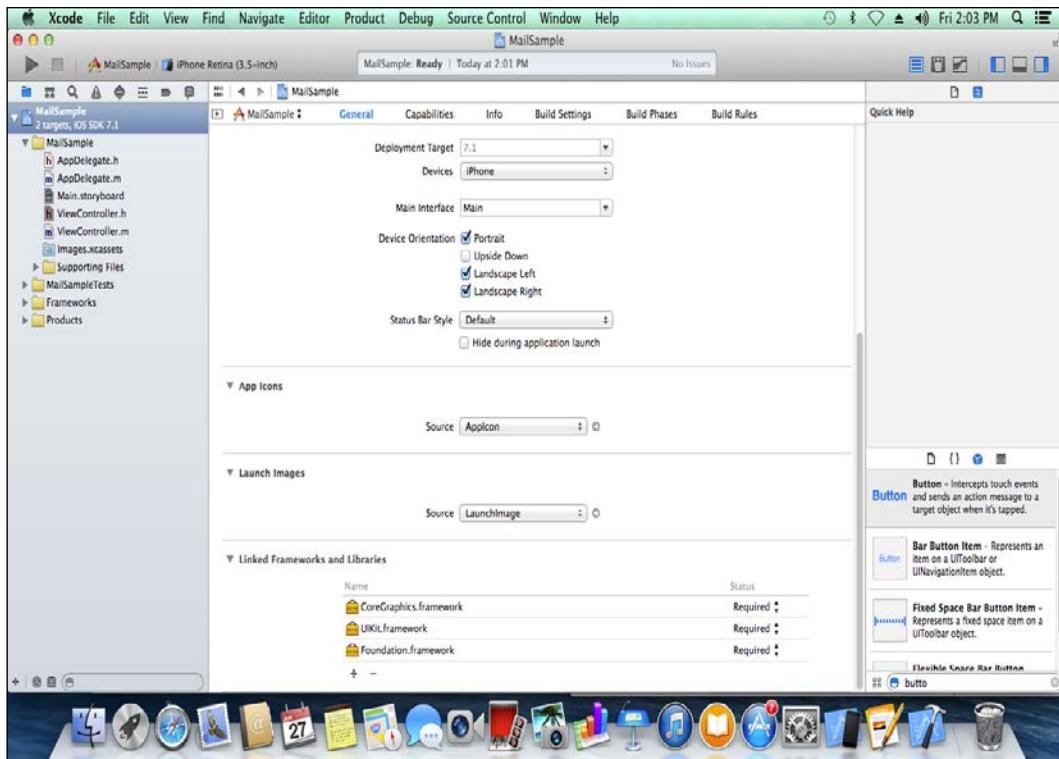
The map view is a very good method for working with locations and maps. It is available in the component list. We can directly drag-and-drop the map view in our storyboard (as shown in the following screenshot) or we can create it by coding:



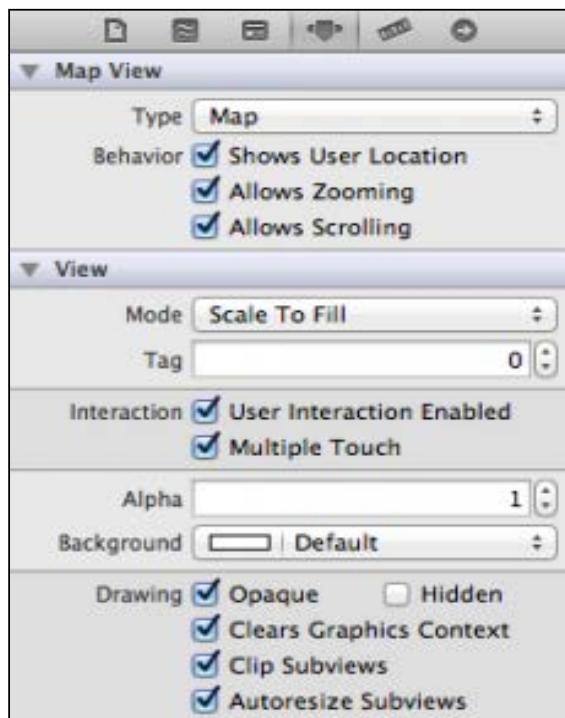
Exploring More UI Components

In order to use map view, you need to add the MapKit framework to your project. You can add the framework by using the following steps:

1. Click on the project name in the left-hand panel and you can see the screen as shown in the following screenshot:

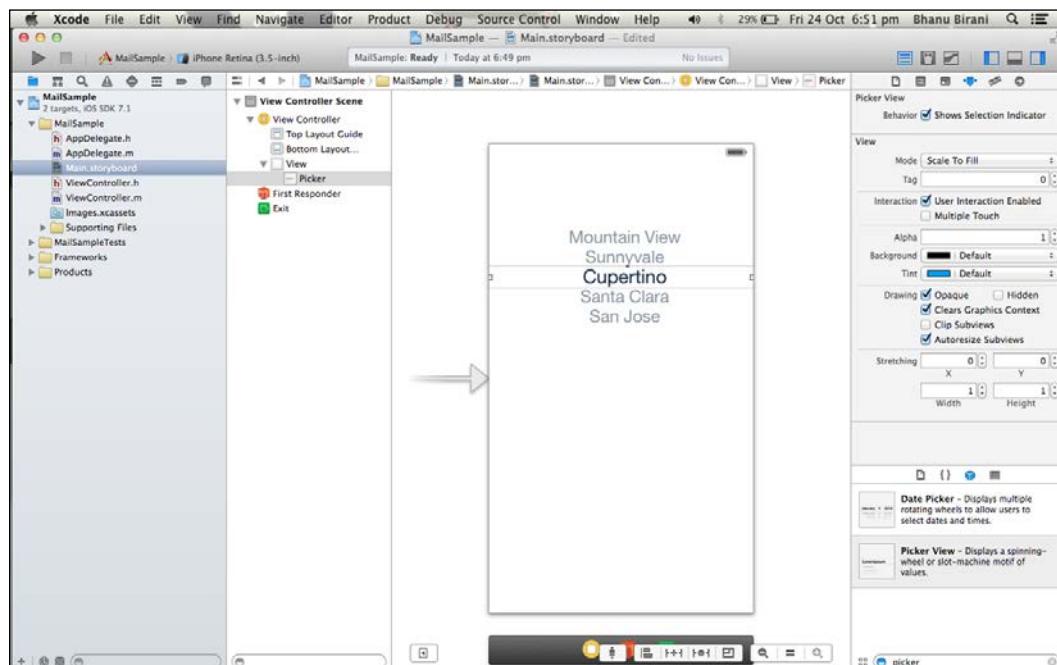


2. In the preceding screenshot, you can see the **Linked Frameworks and Libraries** option. The frameworks are included in our project by default. If you want to add a new framework, then click on the + (plus) sign, find the MapKit framework, and click on **Add**.
3. Now, if you want to show the user's location, go to the inspector toolbar in the right-hand panel and select the fourth tab, which is the **Attributes** inspector. Click on the **Shows User Location** checkbox, as shown in the following screenshot:



UIPickerView

UIPickerView is a UI element that can be used to make a selection from multiple choices (similar to what dropdown does for a web page). We can find the UIPickerView element in the component panel and use it simply by dragging-and-dropping it, as shown in the following screenshot. There is no need to add a framework for UIPickerView.



Drag-and-drop Picker view to the storyboard and run the program. You can see the Picker view on the simulator. However, to make better use of the Picker view, or to insert our data in Picker view, we need to code it in the following manner:

1. First of all, connect the UIPickerView element to the .h file.
2. While connecting, give a name to the Picker view (for example: Picker).
3. After connection, one property is made in the .h file like this:

```
@property (weak, nonatomic) IBOutlet UIPickerView *Picker
```

Picker is the name that we gave to the Picker view. We set the property as follows:

- **weak:** This automatically releases the object after use.
- **nonatomic:** This is thread-safe. Only one thread at a time can use the object.

4. In order to use the Picker view, we need to add two protocols in the .h file; they are `UIPickerViewDataSource` and `UIPickerViewDelegate` inclined within `<>`, as shown in the following code snippet:

```
#import <UIKit/UIKit.h>
@interface ViewController : UIViewController<UIPickerViewDataSource,
UIPickerViewDelegate>
@property (weak, nonatomic) IBOutlet UIPickerView *picker;
@end
```

5. Now, go to the .m file. There is a function named `viewDidLoad`. Add the following code to it:

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

    // Initialize Data
    NSArray *pickerData;
    pickerData = @[@"Item 1", @"Item 2", @"Item 3",
                  @"Item 4",@"Item5"];

    // Connect data
    self.picker.dataSource = self;
    self.picker.delegate = self;
}
```

6. In order to use `UIPickerView` in a proper way, we need to add Picker view methods in the .m file, as follows:

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

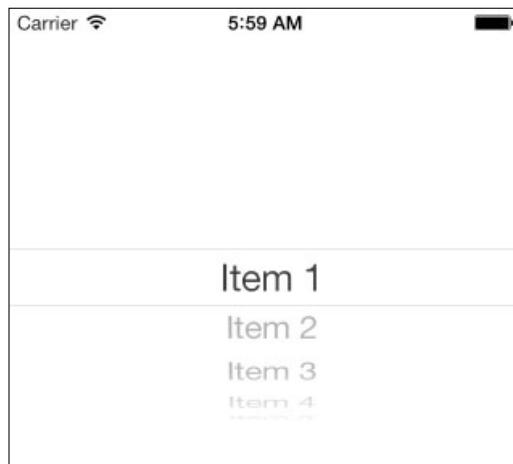
// The number of columns of data
- (int)numberOfComponentsInPickerView:(UIPickerView
    *)pickerView
{
    return 1;
}

// The number of rows of data
- (int)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
```

```
        return pickerData.count;
    }

    // The data to return for the row and component (column)
    // that's being passed in
    - (NSString*)pickerView:(UIPickerView *)pickerView
        titleForRow:(NSInteger)row forComponent:(NSInteger)component
    {
        return pickerData[row];
    }
```

7. Now, compile and run the code. Our simulator will look as shown in the following screenshot:



The web view

We need web view to interact with web content. Web view displays the websites on our devices. Safari and Chrome are examples of web view, but it contains many UI elements such as labels, Refresh buttons, Next buttons, and Back buttons. However, we will not go that much deeper. To begin with, we will just display the website on our device. Along with all the components, web view is also available in the left-hand component panel.

Let's begin with the following steps:

1. Drag-and-drop web view to the storyboard.
2. Connect the web view to the .h file and give a name to web view (for example, `viewSite`). After connection, one property is automatically created, as follows:

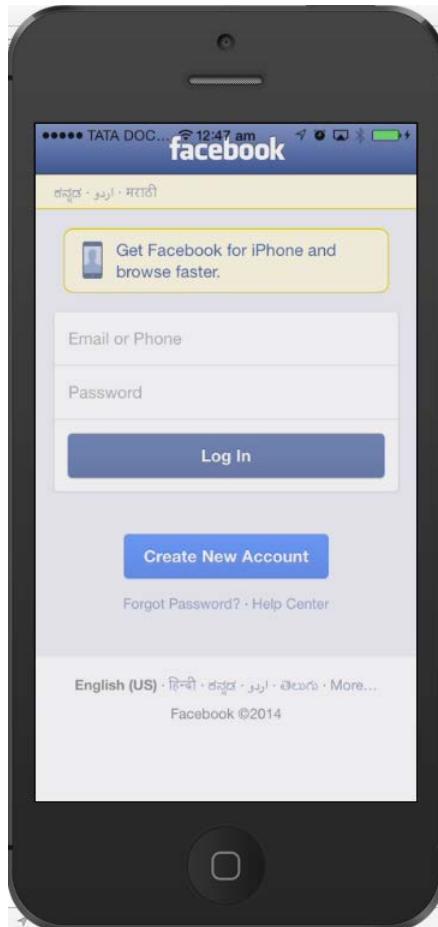
```
@property (weak, nonatomic) IBOutlet UIWebView *viewSite;
```

3. Now, in the .m file, go to the `viewDidLoad` function and add the following code:

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

    NSString *String = @"http://www.facebook.com";
    NSURL *url = [NSURL URLWithString:String];           //passing
string in url
    NSURLRequest *requestObj = [NSURLRequest requestWithURL:url];
    [self.viewSite loadRequest:requestObj];
}
```

4. Simply compile and run the code. You will see the Facebook login page on the simulator, as shown in the following screenshot:



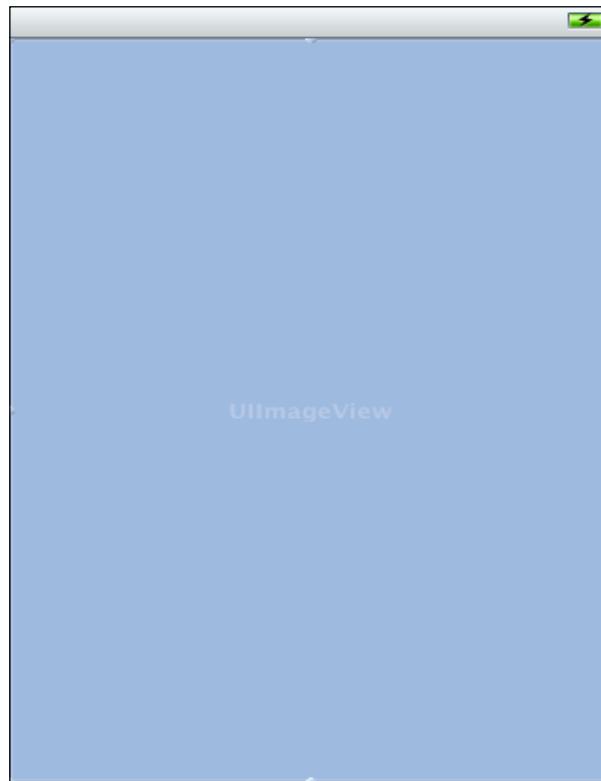
The image view

Image view is used to display an image on the screen. In order to use the image view, there is a component available in the interface builder named `UIImageView`. We can create many animations using the image view, and we can use more than one image view at a time as needed. There is no need to add any framework for the image view. Image view accepts images in many formats such as PNG, JPEG, BMP, and so on. We can add images in both ways, programmatically and by dragging-and-dropping.

Using the image view

Let's have a look at the following steps to understand how to use the image view:

1. Find the image view in the interface builder and drag and drop it to the storyboard. After inserting the image view, our view looks like this:



2. Add the image in the program by simply dragging-and-dropping from another file to Xcode in the right-hand panel below all the classes. A pop-up will appear where you need to tick the **Destination** checkbox.
3. Now, go to the Attribute inspector in the inspector toolbar, which is available in the left-hand panel of Xcode. Give a name to the image that is the same as the image name we are using (for example, `image.png`), as shown in the following screenshot:



4. Compile and run the program. Your image will appear on the simulator, as follows:



Understanding the anatomy of the table view

Table view is a common UI element in iOS. It is used to display a list of data in tabular form. We use it every day in many forms such as a playlist for songs, contacts in the phone, and so on. It is used to display a vertically scrollable view that consists of a number of cells. It has special features such as headers, footers, rows, and sections. There is also a feature for reusing the cell. Thus, we can make unlimited cells as needed.

An empty table view will look like the following screenshot:



Important methods for the table view

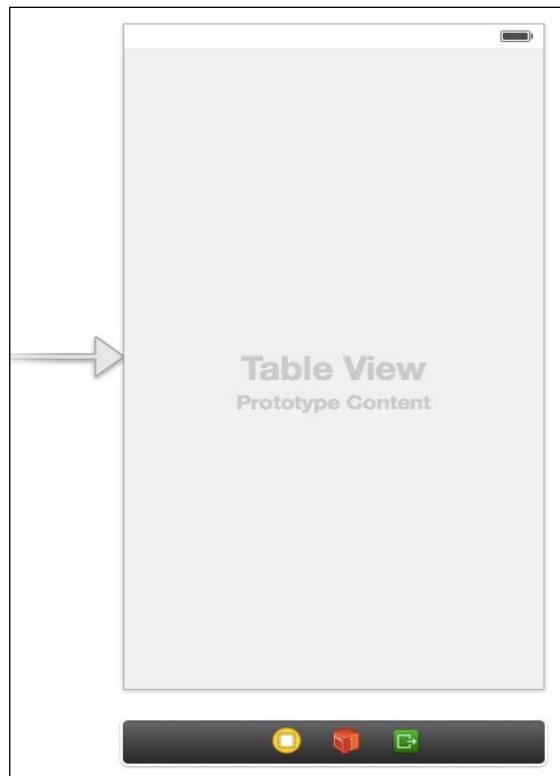
There are various methods available that we can perform on table view. They are as described in the code snippet:

```
- (UITableViewCell *)cellForRowAtIndexPath:(NSIndexPath *)indexPath  
- (void)deleteRowsAtIndexPaths:(NSArray *)indexPaths  
    withRowAnimation:(UITableViewRowAnimation)animation  
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier  
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier  
    forIndexPath:(NSIndexPath *)indexPath  
- (void)reloadData  
- (void)reloadRowsAtIndexPaths:(NSArray *)indexPaths  
    withRowAnimation:(UITableViewRowAnimation)animation  
- (NSArray *)visibleCells
```

Working with the table view

Table view is a common element that is available in interface builder. The following steps will help us to understand how to use the table view:

1. Select the table view component from the interface builder, drag-and-drop it upon our view, and adjust the table view on the view controller properly.
2. After inserting the table view, our storyboard looks like the following screenshot:



3. In order to use table view, we need to add two protocols, `UITableViewDelegate` and `UITableViewDataSource`, in our .h file (our interface file):

```
@interface TableViewController : UIViewController  
    <UITableViewDelegate, UITableViewDataSource>
```

4. Now, in the .m file, we need to declare an array. Here, an array is used to store data cell by cell. After declaring an array, insert data into the array that is defined in the viewDidLoad method, as shown in the following code:

```
@implementation TableViewControler
{
    NSArray *Recipelist;
}
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Initialize table data
    Recipelist = @[@"Crispy Egg", @"Braised Pork",
                   @"Domestic Cheese", @"Roasted baby beets"];
}
```

5. Now, we have to add two table view methods, numberOfRowsInSection and cellForRowAtIndexPath, to our code in the .m file. The first method is used to inform the table view about the number of rows present in the section. The second method simply returns the number of items in the tableData array.

6. Add the code as shown here:

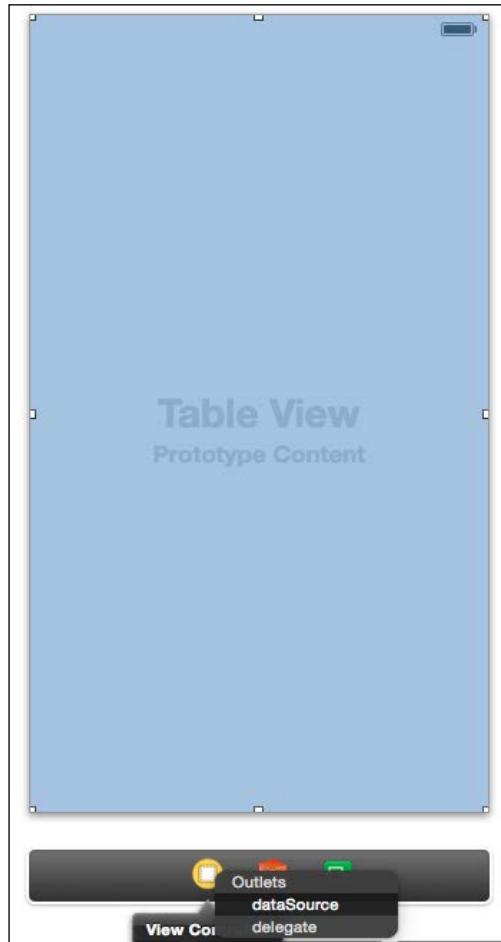
```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [Recipelist count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    //create identifier for cell
    NSString *TableIdentifier = @"TableViewCell";

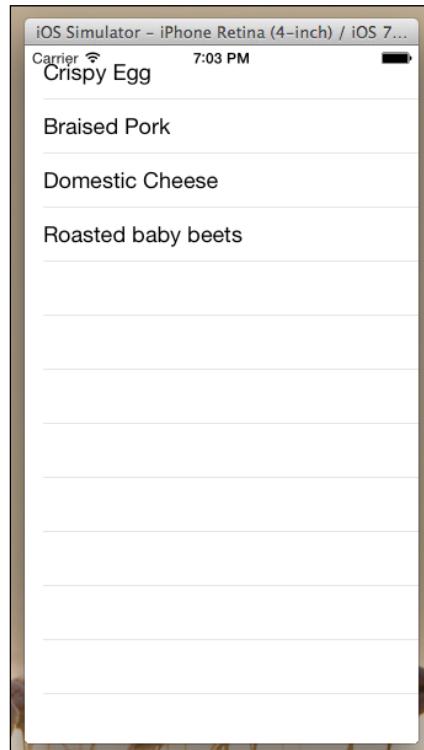
    // dequeReusableCell is method to reuse the cell
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:TableIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:TableIdentifier];
    }

    //set the text for each cell as per our array objects
    cell.textLabel.text = [Recipelist
        objectAtIndex:indexPath.row];
    return cell;
}
```

7. Now, go to the storyboard. We need to establish the connection between the data sources and the delegates.
8. Select the table view from our view. Right-click on it and connect to the yellow button, shown below, that is our table view controller.
9. Select both data sources and delegate them one by one, as shown in following screenshot:



10. Finally, we can compile and run the program. Our simulator looks like this:

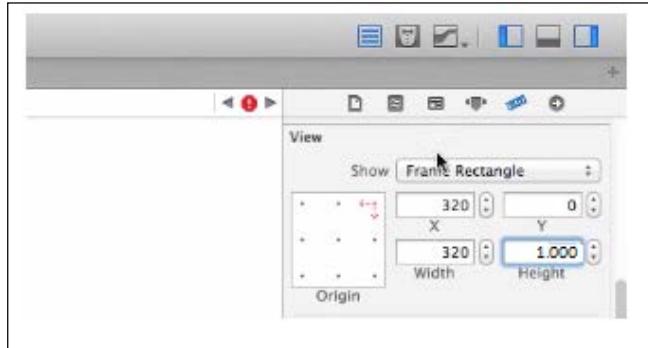


Scroll view and its usage

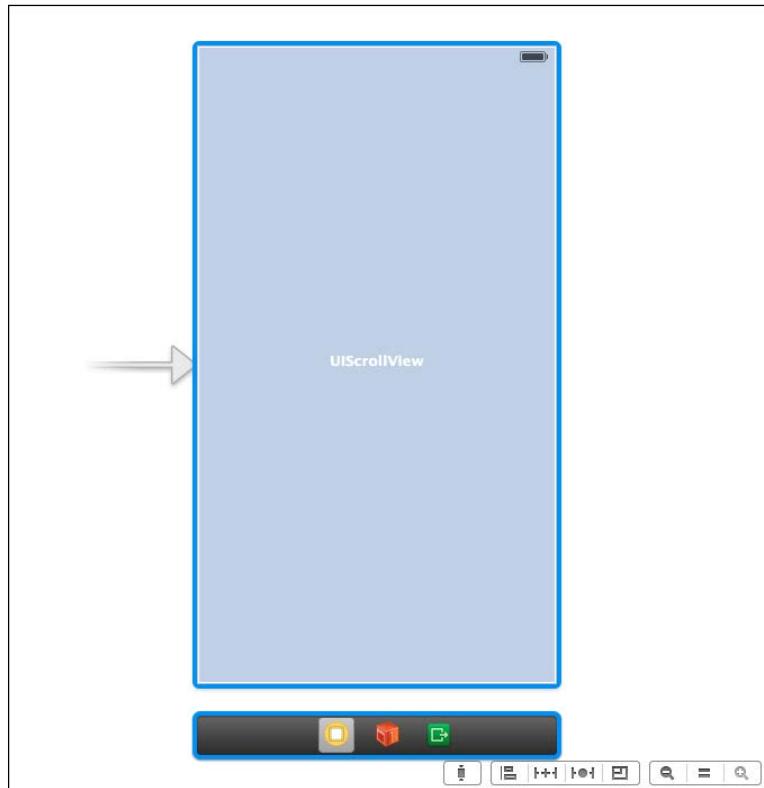
ScrollView is used to display content more than the size of the screen. It can contain all of the other UI elements such as image views, labels, text views, and even another scroll view also. UIScrollView is one of the most useful controls in iOS. It is a great way to represent data that is larger than the screen; the data can be an image, list, forum, and so on. The scroll view element is present in interface builder and we can directly use it by dragging it to the storyboard. Let's do this simple activity to understand the scroll view:

1. Make a single view application and select the storyboard from the left-hand panel.
2. Move to the storyboard and select view from the view controller.

3. Now, go to the Inspector toolbar and then to the **View** coordinates; change the height of the view to **1,000**, as shown in the following screenshot. So, your view will go longer than the screen.



4. Search for the scroll view in the interface builder from the right-hand panel and drag it to the view, as shown in the following screenshot. Don't forget to adjust the scroll view on top of the view properly:



5. We need to add some element to check whether scroll view is working or not. So, add two labels, one to the top of the view and the other to the bottom of the view, and name them.
6. Compile and run the program.

You will see only one top label; now, scroll down and see the bottom label. So, through the scroll view, we can scroll to any elements such as images, tables, and so on.

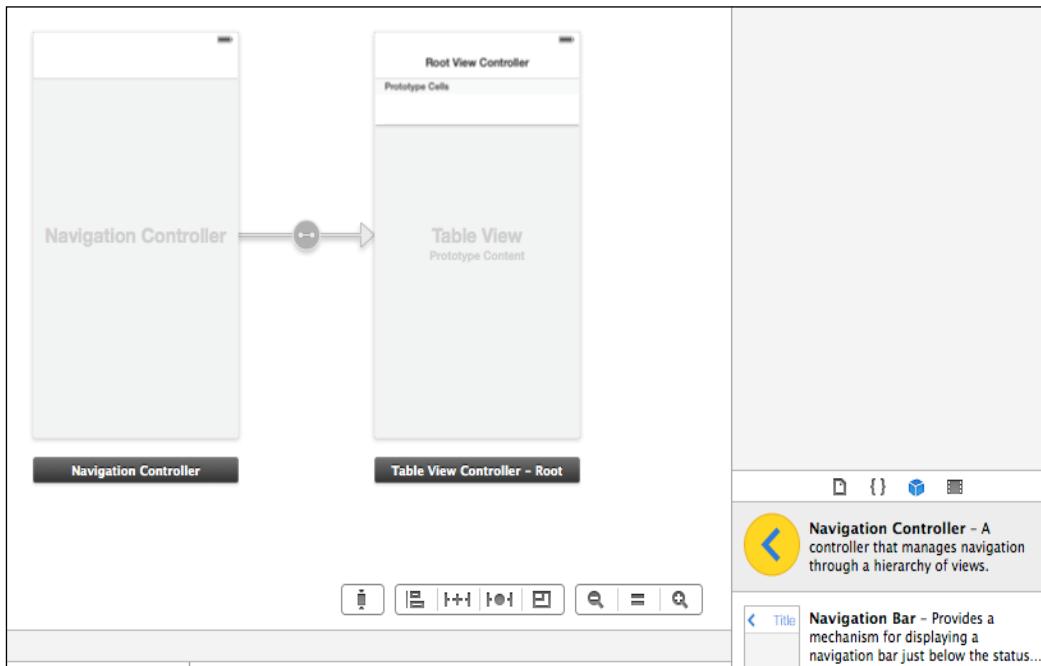
Navigation controller

Navigation controller is another UI element you commonly find in iOS apps. Many applications such as photo apps, YouTube, and contacts are made from a navigation controller. The navigation controller manages several subviews including a navigation bar at the top and back buttons. We can show or hide the navigation bar as per our application requirement. It is used when we want to use more than one view on a single application. Through navigation controller, we can move from one view to another view. Navigation controller is also a type of element and is available in the interface builder. Navigation controller provides the back button facility through which we can go to the previous view. We can add more than one view to the storyboard and can connect them together by using segue. **Segue** is a special feature provided by the navigation controller that pushes the navigation controller to the next view.

Let's make a simple app through which we can understand navigation controller easily:

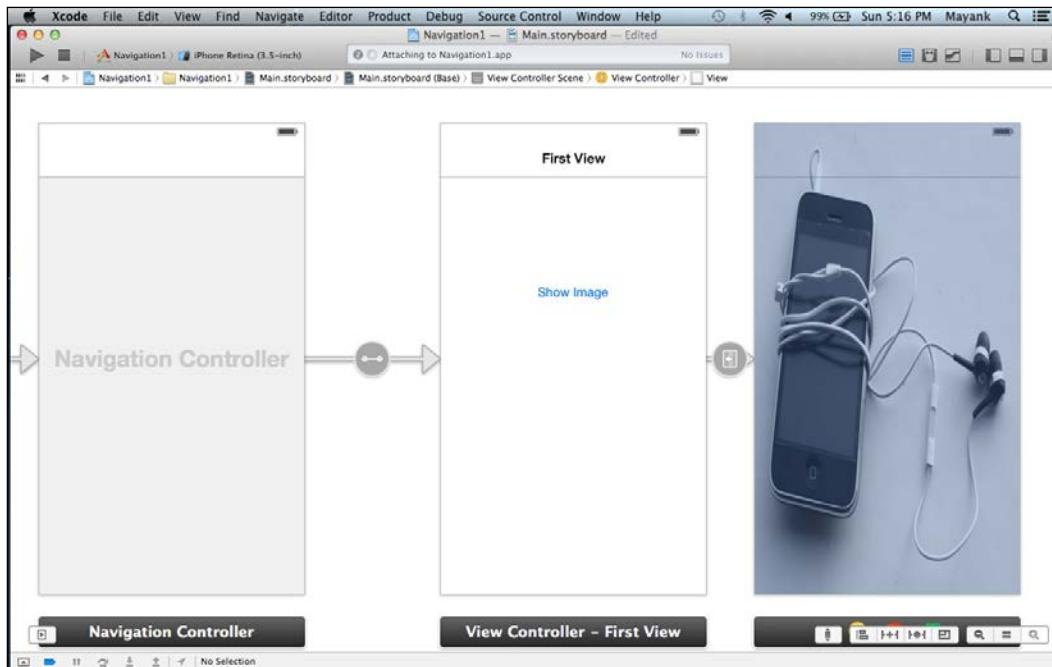
1. Make a single view application and move to the storyboard.
2. Delete the existing view from the left-hand panel.

3. Search for navigation controller in the interface builder and drag it to the storyboard. Now, your screen looks like this:

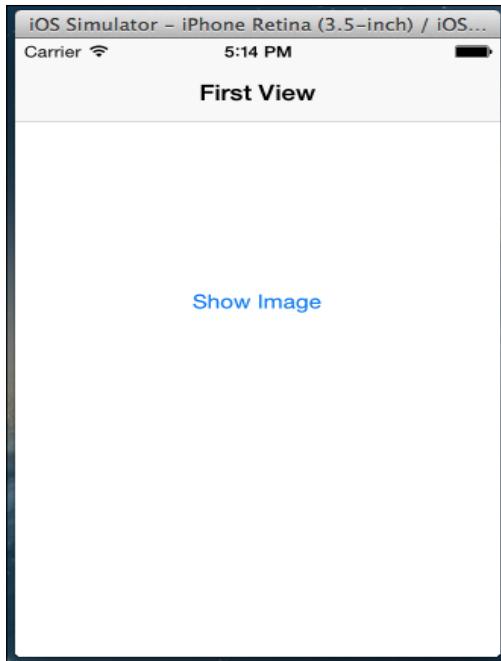


4. The linkage sign between the views is the segue sign. Select a button from the interface builder and drag it to the root view controller.
5. Find a view from the components. Drag it near the root view controller and insert an image view to the newly inserted view.
6. Copy any image from your directory to Xcode below all the classes in the left-hand panel.

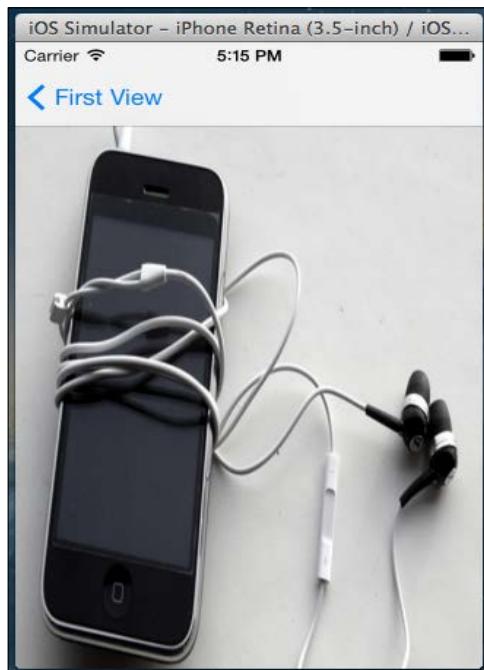
7. Go to the Inspector toolbar and give the image the same name as the image we have chosen.



8. Finally, connect the button to the new view. Right-click on the button and drag the blue line to the new view to choose **Push** from the pop-up window.
9. It's time to compile and run the program. When you see the simulator, there will be a button and navigation bar at the top of the screen. When you click on the button, a new screen will appear with your image and there is also a back button available to get the previous view. The following screenshot shows the first view:



10. Here is the second view:



Summary

In this chapter, we learned about components and how to create them programmatically. We mastered the concept of table views and how to use them. Also, we made a simple application using navigation controller. Now, we will discuss the framework in the next chapter.

3

Exploring Various Frameworks

A framework is a collection of resources. It collects a static library and its header files into a single structure that Xcode can easily include into our projects. A framework is a dynamic library. In iOS, there are two kinds of frameworks: public frameworks and private frameworks. From iOS 3.1, all private and public libraries are combined into a big cache file. There are a lot of frameworks provided by Apple for different functionalities. The extension of the framework is .framework. For every unique feature, a framework is defined – for example, for using maps a MapKit framework is available, and for location, a CoreLocation framework is available. Various frameworks are present at the <Xcode.app>Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/<iOS_SDK>/System/Library/Frameworks directory, where Xcode.app is our application path (where we saved our app).

In this chapter, we will cover the following topics:

- Framework descriptions in a tabular form
- Databases in iOS
- Social integration
- Activities to understand every topic in a better way

Generally, frameworks are bundles that contain a linkable library (.dylib) and associated resources and headers for development. Each framework contains sample code and other resources associated with it. There are a number of frameworks provided by Apple. Frameworks are listed in the following table with their descriptions:

Name	Description
Accelerate.framework	This framework handles the math, DSP, large numbers, and image processing.
Accounts.framework	This framework provides access to accounts in the Accounts database. It allows creation of accounts if none exist.
AddressBook.framework	This framework provides access to the AddressBook database.
AddressBookUI.framework	This framework contains classes for displaying the system-defined people Picker and editor interfaces.
AdSupport.framework	This framework provides access to the identifiers to serve adverts and a flag that indicates if limited tracking is on.
AssetsLibrary.framework	This framework gives access to user photos and videos.
AudioToolbox.framework	This framework provides an interface for recording, playing, and audio streaming.
AudioUnit.framework	This framework is used to load audio units and their uses.
AVFoundation.framework	This framework is used for playing and recording audio and video.
CFNetwork.framework	This framework is used to access networks through Wi-Fi or cellular networks.
CoreAudio.framework	This framework provides the datatypes used for Core Audio.
CoreBluetooth.framework	This framework provides access to Bluetooth (hardware).
CoreData.framework	This framework contains the interface for application data model.
CoreFoundation.framework	This framework provides software services and basic management of data.
CoreGraphics.framework	This framework contains the API for the Quartz engine and gives a 2D view.
CoreImage.framework	This framework is the interface used to manipulate images and video.

Name	Description
CoreLocation.framework	This framework is the interface used to determine the user's location.
CoreMedia.framework	This framework includes the low-level routines for manipulating audio and video.
CoreMIDI.framework	This framework includes the low-level routines for handling MIDI data.
CoreMotion.framework	This framework is the interface to access accelerometer and gyrometric data.
CoreTelephony.framework	This framework allows access to Carrier information and information related to the current call.
CoreText.framework	This framework contains a text layout and rendering engine.
CoreVideo.framework	This framework includes low-level routines for using audio and video. It is advised to not use it directly.
EventKit.framework	This framework interface is used to access the calendar and events.
EventKitUI.framework	This framework contains classes for displaying the calendar's interface.
ExternalAccessory.framework	This framework is an interface used to communicate with the attached hardware.
Foundation.framework	This framework contains the interface for managing Strings, Arrays, collections, and low-level datatypes.
GameController.framework	This framework is the interface to communicate with game-related hardware.
GameKit.framework	This framework is used for peer-to-peer connection and to create social games.
GLKit.framework	This framework is used for building the OpenGL ES application.
GSS.framework	This framework provides security-related services.
iAd.framework	This framework is used to display advertisements.
ImageIO.framework	This framework contains classes to read and write image data.
IOKit.framework	This framework is a low-level framework and is used to communicate with the kernel and the hardware.
JavaScriptCore	This framework contains files for evaluating JavaScript code and parsing JSON.
MapKit.framework	This framework is used for embedding a map in our application and to use reverse geocoding.

Name	Description
MediaAccessibility.framework	This framework was defined in iOS 7. It manages the presentation of closed-caption content in media files.
MediaPlayer.framework	This framework contains the interface for playing a video in full screen mode.
MediaToolbox.framework	This framework contains the interface for playing audio content.
MessageUI.framework	The framework contains the interface for composing e-mails.
MobileCoreServices.framework	This framework defines UTIs supported by the system.
MultipeerConnectivity.framework	This framework was introduced in iOS 7. It is responsible for implementing peer-to-peer networking between devices.
NewsstandKit.framework	This framework provides the interface for downloading magazines and newspapers in the background.
OpenAL.framework	This framework contains the interface for the cross-platform audio library.
OpenGL ES.framework	This framework provides the interface for the OpenGL ES library and contains classes that are used for 2D and 3D graphics.
PassKit.framework	This framework contains interfaces for creating digital passes to replace things such as tickets, membership cards, and so on.
QuickLook.framework	This provides the interface for previewing a file.
SafariServices.framework	This framework was introduced in iOS 7. It supports the creation of reading list items in Safari.
Security.framework	This framework provides the interface for managing keys, trust policies, and certificates.
Social.framework	This framework contains the interface to communicate or integrate social network services.
SpriteKit.framework	This framework was introduced in iOS 7. It facilitates the creation of sprite-based animations.
StoreKit.framework	This framework is responsible for handling financial transactions associated with the app.
SystemConfiguration.framework	This framework contains the interfaces for determining whether the network is available or not.
Twitter.framework	This framework contains the interface for sending tweets.

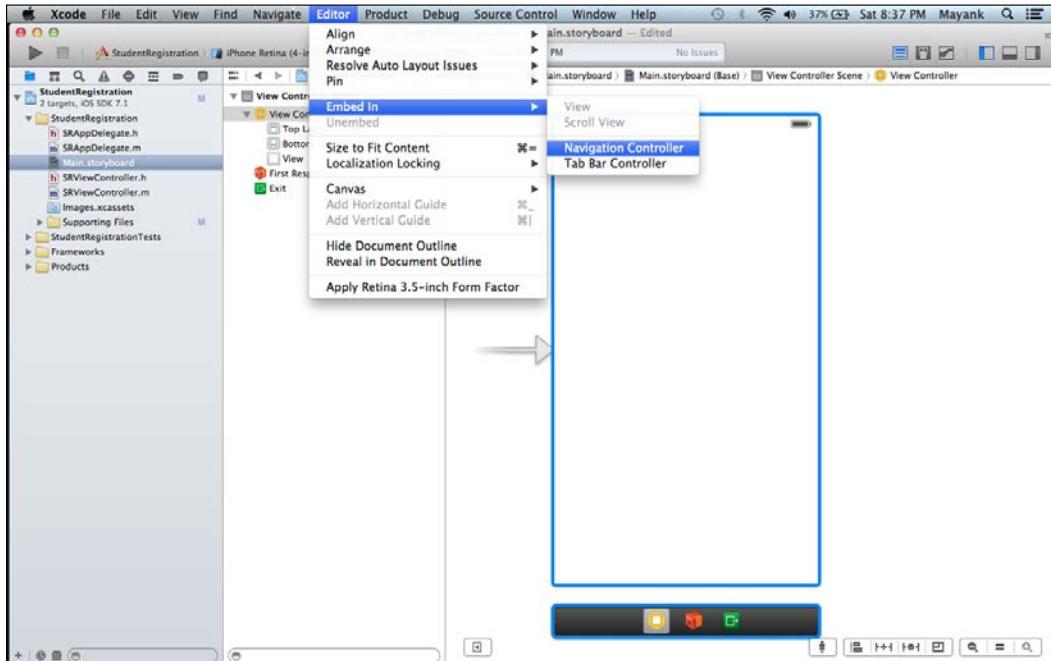
Name	Description
UIKit.framework	This framework contains the classes for iOS UI components and for the user interface layer of applications.
VideoToolbox.framework	This framework contains the interfaces used by the device. It is advised to not use it directly.

These are the various frameworks provided by Apple and we can use them as needed.

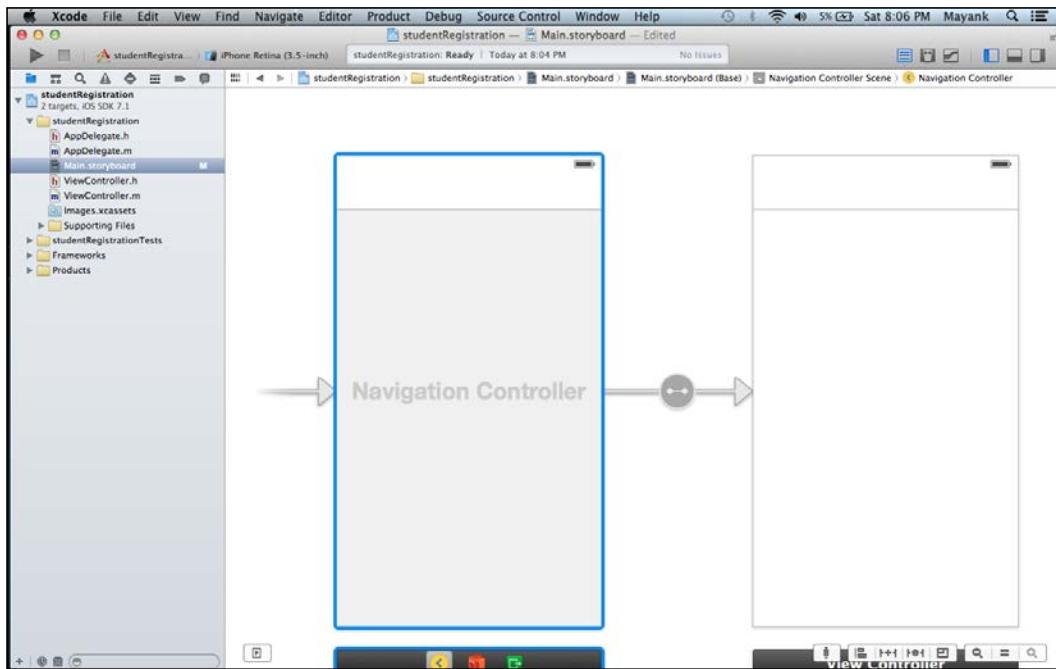
Exploring various UI components with libraries

You've already learned about UI in the previous chapter. Now we can start by exploring some UI components and the frameworks. We will make a simple Student Registration app by performing the following steps:

1. First of all, create a new single view project in Xcode.
2. Now, go to storyboard and navigate to **Editor | Embed In | Navigation Controller**, as shown in the following screenshot:

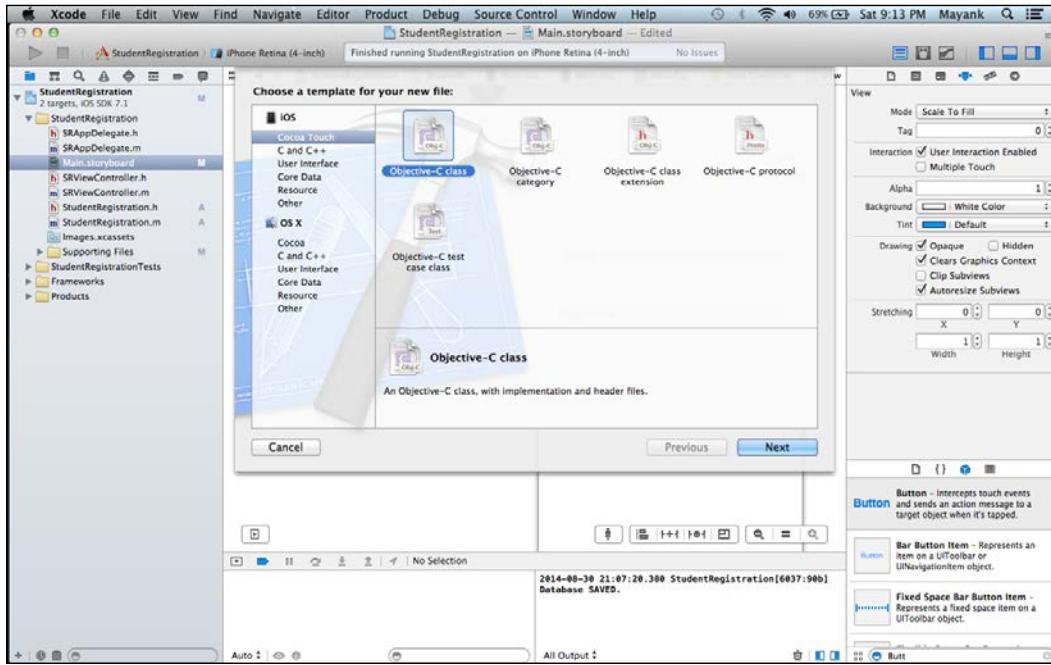


3. Now your storyboard looks like this:



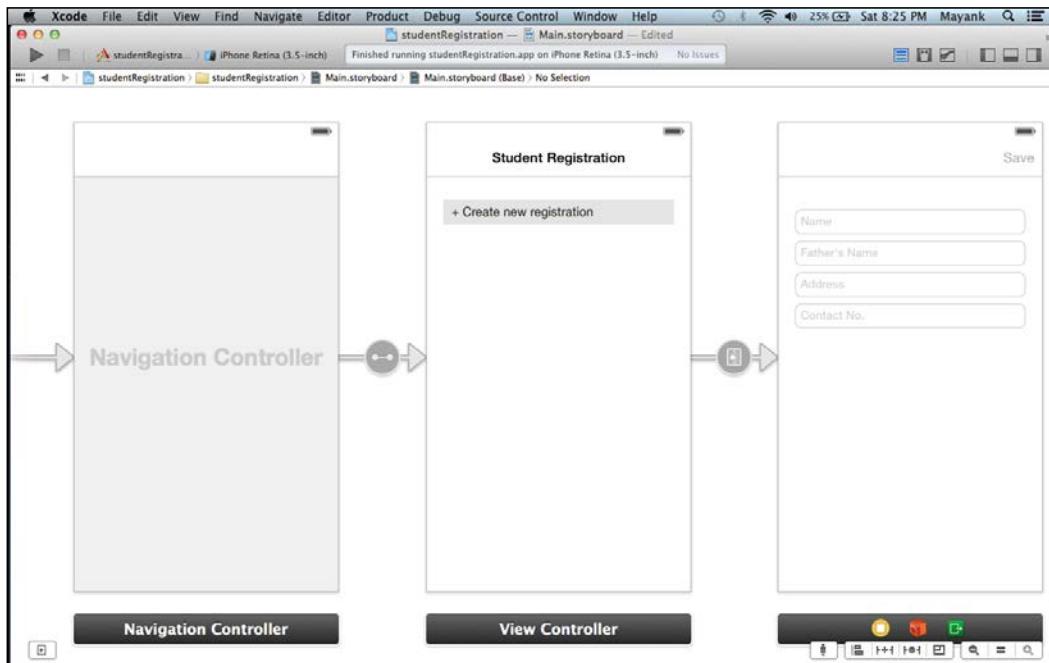
4. Select the navigation bar of the second view by clicking on it and rename it to Student Registration.
5. Add one button to the view (name it Add Student) and set the color of the button text and background of the button from the Inspector panel on the right-hand side.
6. Search for a new view controller in the interface builder and drag it beside the second view.
7. For a new view controller, we need to add new class files. Go to **File** at the top of Xcode and then navigate to **File | New | File....**

- Now, create a new **Objective-C class** file and give a suitable name to the class (for example, `StudentRegistration`), as shown in the following screenshot:



- Go to the storyboard, select our third view, and give it a name that is similar to our new Objective-C class from the inspector element.
- Right-click on the button, select the **Action** option, drag to the new view (the third view), and select **Push**.
- Now, drag four text fields in the third view as shown in the following screenshot. Select one text field and go to the inspector panel where you will find an option **Placeholder**. Here, enter text for all the text fields.
- Find the bar button from the interface builder and drag it to the bar of the third view and name it (for example, `Save`).

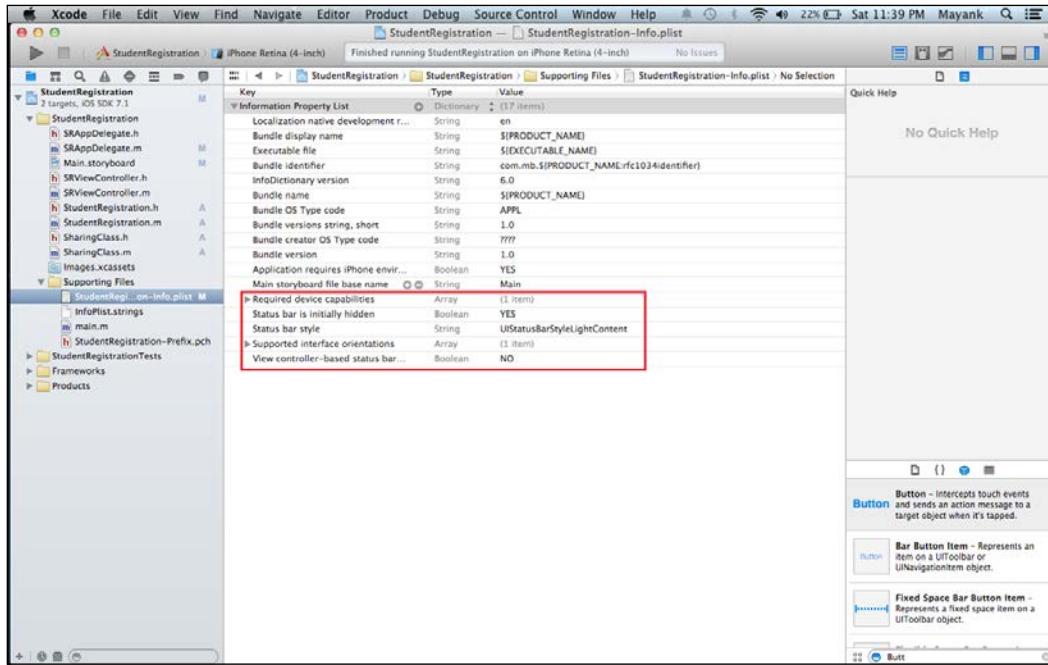
By following the preceding steps, our storyboard will look like this:



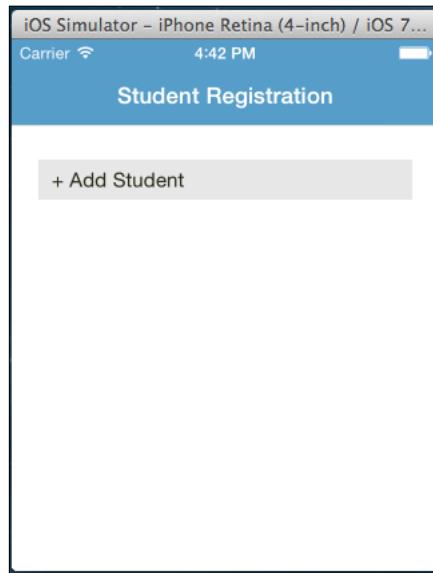
Here is our simple UI, but this time we need to stretch our UI component. We can change the color of our navigation bar for better visibility. To change our navigation bar color, we need to apply the following code. We can insert the following code in the `didFinishLaunchingWithOptions:` method of `AppDelegate.m`:

```
[[UIApplication sharedApplication] setStatusBarHidden:  
NO withAnimation:YES];  
// Change navigation bar color  
[[UINavigationBar appearance] setBarTintColor:  
[UIColor colorWithRed:41.0f/255.0f green:128.0f/255.0f  
blue:185.0f/255.0f alpha:1.0f]];  
  
[[UINavigationBar appearance] setTintColor:  
[UIColor whiteColor]];  
  
// Change font of navigation bar  
[[UINavigationBar appearance]  
setTitleTextAttributes:[NSDictionary  
dictionaryWithObjectsAndKeys:  
[UIColor whiteColor],NSForegroundColorAttributeName, nil]];
```

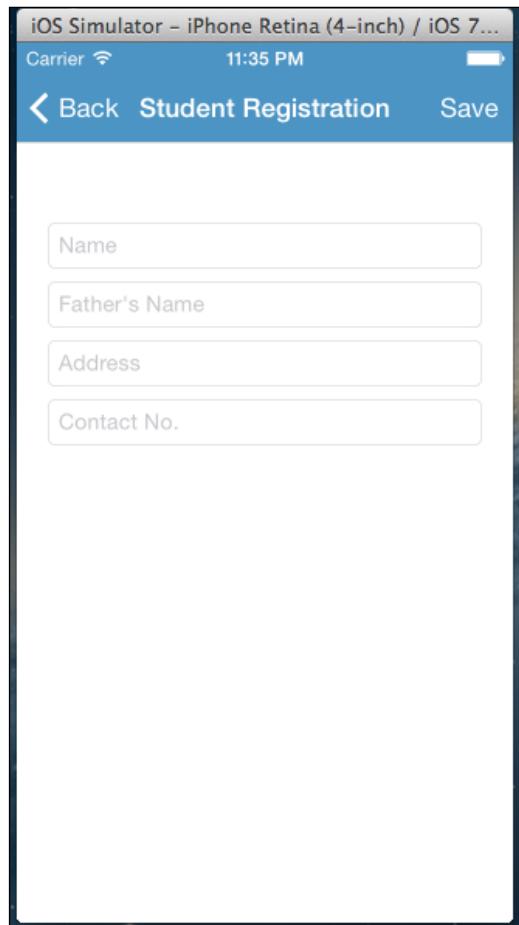
Now, we need some changes in our `StudentRegistration-Info.plist` file. Make the changes according to the following screenshot:



After running our project, the output will look like this:



Now, tap on the **+ Add Student** button; it will push our app to another view and our screen will look like this:



Database integration

Databases are a way to save our data in computer memory. A database is a set of data held in computers, and can be accessible in many ways. In iOS, there is mainly only one type of database: **SQLite**.

SQLite

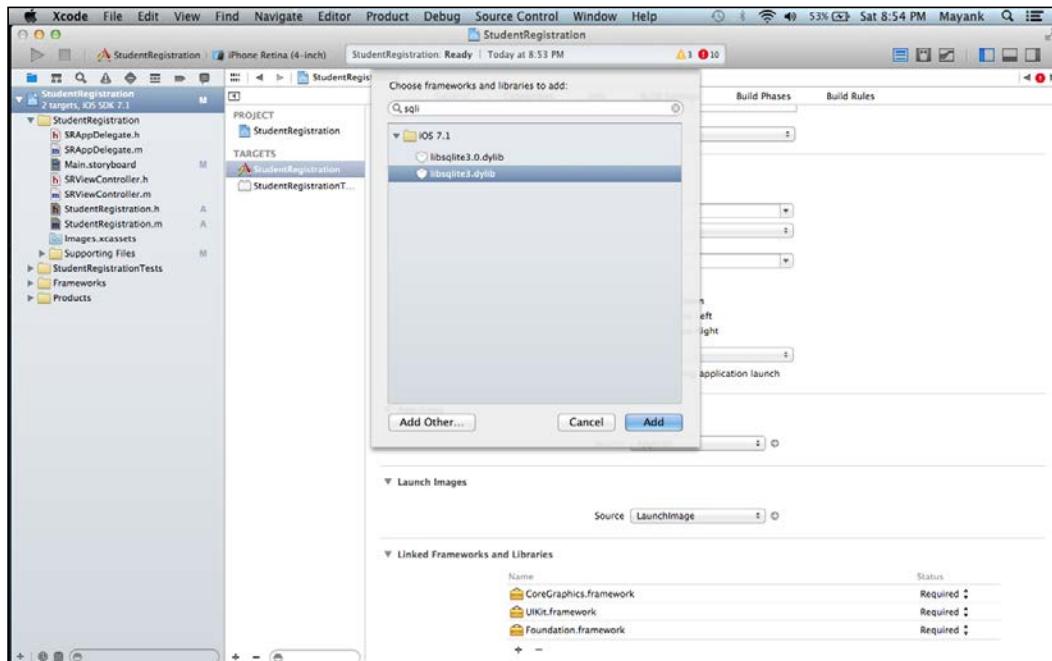
SQLite is an embedded implementation of SQL. SQLite is an in-process library that implements a self-contained, zero-configuration, server-less, and transactional SQL database engine. The source code for SQLite exists in the public domain and is free for both private and commercial purposes. SQLite is available for several programming languages such as C, C++, Java, and so on. It is also available in several operating systems such as iOS, Android, Symbian, Blackberry, and so on. SQLite is used for handling data files or creating data files in iOS. It can be easily performed by using SQL queries. SQLite works on the principle of RDBMS in which data is stored in a table and the relationship is also stored inside the table.

So we can use SQLite in our project, a library and a framework are provided by Apple: `iOSsqlite3.dylib` and `sqlite3.h` respectively. There are several functions of SQLite. The important functions are as follows:

- `sqlite3_open()`: This function creates and opens an empty database. If the database already exists, it will only open the database.
- `sqlite3_close()`: This function is used to close an opened SQLite database connection. It will free all the resources associated with the database connection.
- `sqlite3_prepare_v2()`: This function is needed to compile the SQL statement into byte code. It basically transforms a SQL statement into an executable piece of code.
- `sqlite3_step()`: This function will call a previously prepared SQL statement.
- `sqlite3_finalize()`: This function deletes a previously prepared SQL statement from memory.
- `sqlite3_exec()`: This combines the functionality of `sqlite3_prepare_v2()`, `sqlite3_step()`, and `sqlite3_finalize()` into a single function call.

Let's carry out a small activity to understand SQLite in a better way. We will extend the previous app. Open that app and proceed with the following steps:

1. Go to **Project Settings | General**. Now, scroll down and click on **+** to add framework. Search for SQLite framework and click on **Add**, as shown in the following screenshot:



2. Import the following header file for SQLite in the new class that we added in the previous steps:

```
#import <sqlite3.h>
```

3. Now, make an object for SQLite in the interface part as follows:

```
sqlite3 *_database;
```

4. Now, link all the text fields and the **Save** button to the .h file of the newly added class file.

5. Now, go to the **Save** method in the .m file and write the following code for SQLite integration:

```
- (IBAction)Save:(id)sender {  
  
    NSArray *paths=  
        NSSearchPathForDirectoriesInDomains(NSDocumentDirectory  
        NSUserDomainMask, YES);
```

```

NSString *documentsDirectory = [paths objectAtIndex:0];
NSString *databasePath =[documentsDirectory
    stringByAppendingPathComponent:@"mydbs.sqlite"] ;

// Check to see if the database file already exists
bool databaseAlreadyExists =
[ [NSFileManager defaultManager]
    fileExistsAtPath:databasePath];

// Open the database and store the handle as a data
// member
if (sqlite3_open([databasePath UTF8String], & _database)
== SQLITE_OK)
{
    // Create the database if it doesn't yet exists in the
    filesystem
    if (!databaseAlreadyExists)
    {
        // Create the SIGNUP table
        const char *sqlStatement = "CREATE TABLE IF NOT
            EXISTS REGISTRATION (NAME TEXT, FATHERNAME TEXT,
            ADDRESS TEXT, CONTACT TEXT)";
        char *error;

        if (sqlite3_exec(databaseHandle, sqlStatement,
            NULL, NULL, &error) == SQLITE_OK)
        {
            NSLog(@"%@", @"Database and tables created.");
        }
        else
        {
            NSLog(@"%@", @"Error: %s", error);
        }
    }
}

//-----SAVING DB-----

// Create insert statement for the person
NSString *insertStatement = [NSString
    stringWithFormat:@"INSERT INTO REGISTRATION (NAME,
    FATHERNAME, ADDRESS, CONTACT) VALUES ('%@',
    '%@', '%@', '%@')",
    Name.text,Fathername.text,Address.text,Contact.text];

```

```

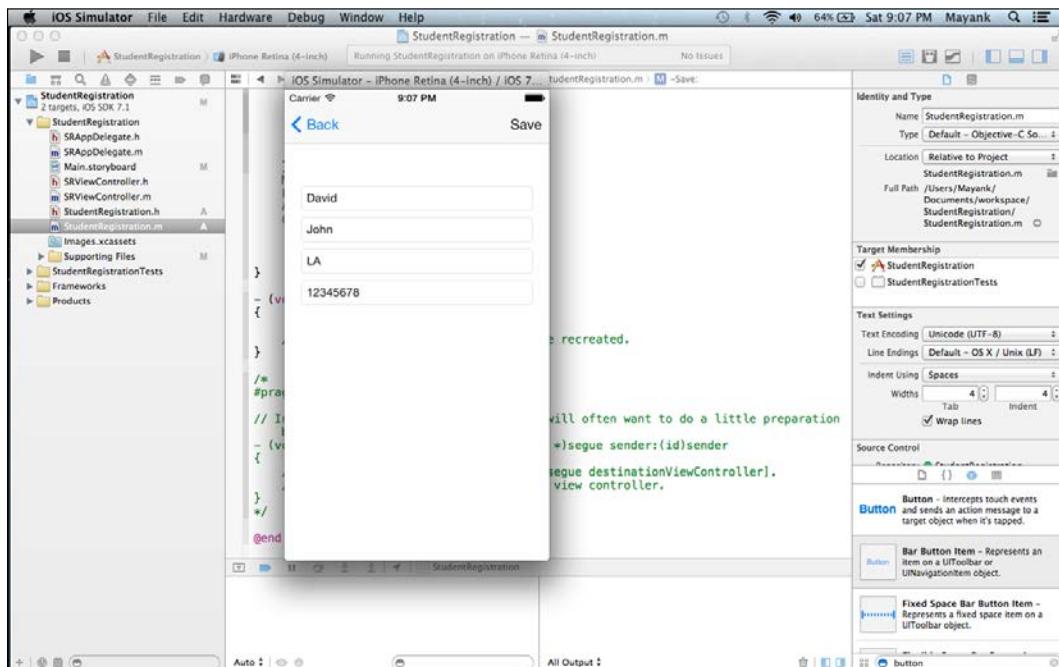
        char *error;
        if ( sqlite3_exec(databaseHandle, [insertStatement
            UTF8String], NULL, NULL, &error) == SQLITE_OK)
        {
            NSLog(@"Database SAVED.");
        }
        else
        {
            NSLog(@"Error: %s", error);
        }

        sqlite3_close(databaseHandle);
    }
    Name.text=@";
    Fathername.text=@";
    Address.text=@";
    Contact.text=@";
}

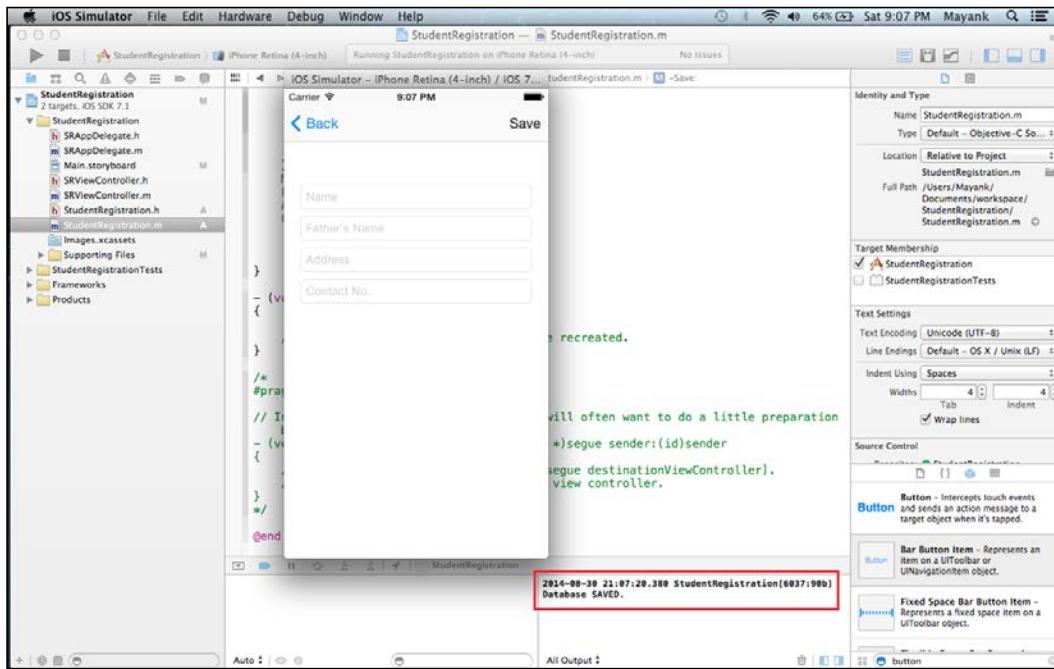
}

```

6. Compile and run your app, insert text in all the text fields, and click on **Save**, as shown in the following screenshot. In the console, the output will be **Database SAVED.**



- Clicking on the **Save** button will clear all our text fields (as shown in the following screenshot) and save our inputs to the database:



This is all there is to integrating a SQLite database with our App.

Core Data

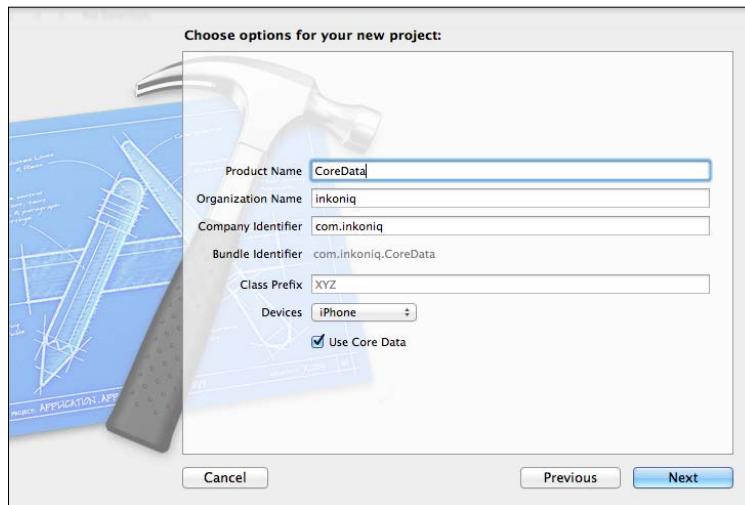
It's used in thousands of applications and by millions of people, both on iOS and OS X. Core Data is maintained by Apple and is very well documented. It's a mature framework. Core Data takes advantage of the Objective-C language and its runtime, and easily integrates with the Core Foundation framework. The result is an easy-to-use framework for managing an object graph that is easy to use and is very efficient in terms of memory usage. Core Data is a powerful framework provided by Apple to include data in applications. It is even more preferable because it doesn't use processes and can easily maintain the relation between data.

Apple defines a framework for Core Data known as `CoreData.framework`. Before using Core Data in our app, we need to add this framework in our project. Core Data is itself not a database of our application; it is a framework that manages an object graph. With Core Data, we can easily trace the objects in our apps to the table records in the database without firing any SQL query. In Core Data, we need to add three different instances to deal with the database, which are as follows:

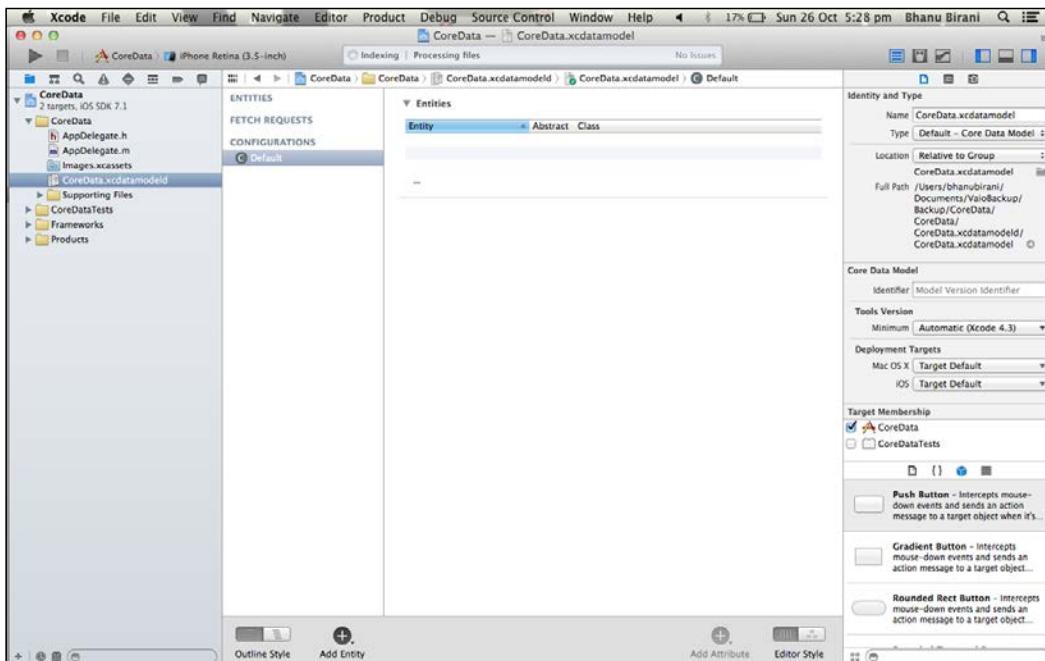
- **Persistent store coordinator:** As the name suggests, it is a coordinator that coordinates between managing object contexts and low-level files saved in our database (SQLite file). We cannot use it directly; it will only be used while setting up `NSManagedObjectContext`. Creating linkage for persistent store coordinates automatically creates linkage for SQLite files.
- **Managed object context:** We can think of it as a scratch pad (a small, fast memory for the temporary storage of data). When we fetch objects from a persistent store, we get temporary copies into the scratch pad. Then we can use and modify those objects however we like without saving those data; the persistence remains unchanged.
- **NSManagedObjectModel:** A managed object model is an instance of the `NSManagedObjectModel` class. It describes a schema (contains definitions) for objects (also called entities).

Let's understand Core Data with an activity:

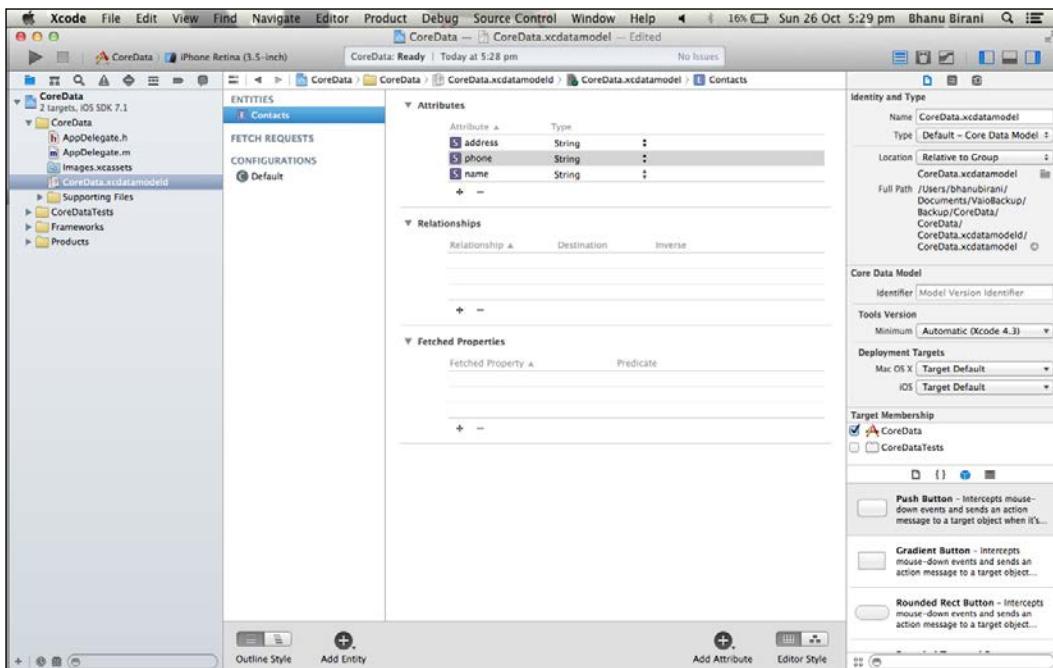
1. To create the example application project, launch Xcode and select the option to create a new project. In the new project window, select the **Empty Application** option. In the **Product Name** field, enter `CoreData`, enable the **Use Core Data** checkbox, and click on **Next** to select a location to store the project files, as shown in the following screenshot:



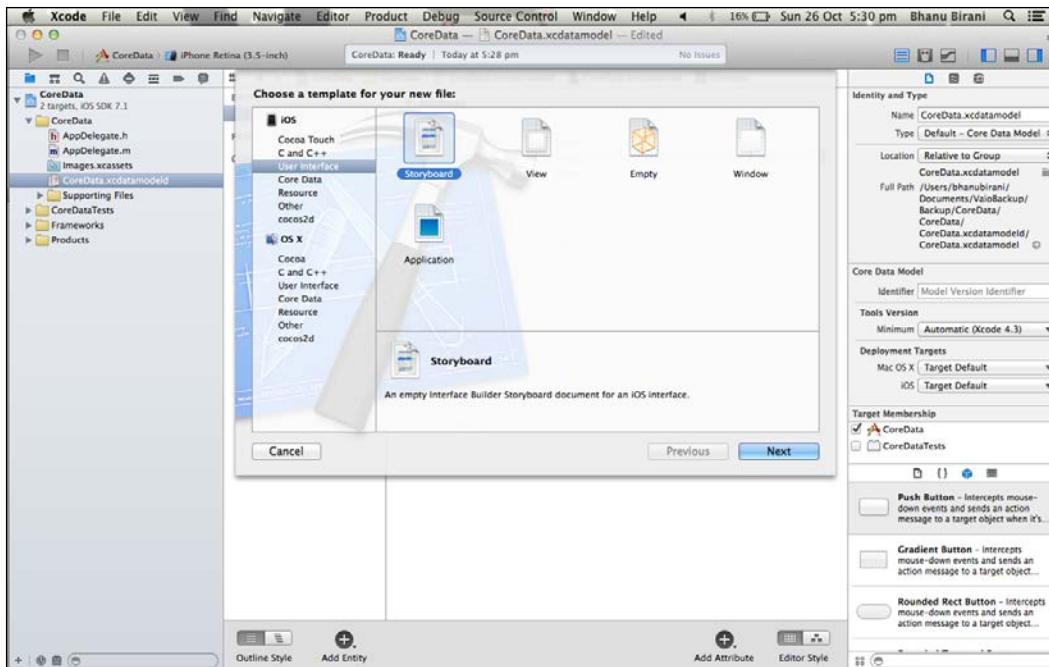
2. In addition to the usual files that are present when creating a new project, this time an additional file named `CoreData.xcdatamodeld` is also created. This is the file where the entity descriptions for our data model are going to be stored. The entity description defines the model for our data, much in the way a schema defines the model of a database table. To create the entity for the `CoreData` application, select the `CoreData.xcdatamodeld` file to load the entity editor, as shown in the following screenshot:



3. To create a new entity, click on the **Add Entity** button located in the bottom panel. Double-click on the new **Entity** item that appears beneath the **Entities** heading and change the entity name to **Contacts**. With the entity created, the next step is to add some attributes that represent the data that is to be stored. To do so, click on the **Add Attribute** button. In the **Attributes** pane, name the attribute as **name** and set the type to **String**. Repeat these steps to add two other String attributes named **address** and **phone** respectively, as shown in the following screenshot:



4. Now, we need to create our own storyboard file and view controller class. To add a storyboard file, navigate to **File | New | File...** and, in the resulting dialog, select the **User Interface** category from beneath **iOS** in the left-hand-side panel. In the main panel, select the **Storyboard** option and click on **Next**, as shown in the following screenshot:



5. Name the storyboard as Main (as shown in the following screenshot), select a location in the project for the new file, and then click on **Create**.

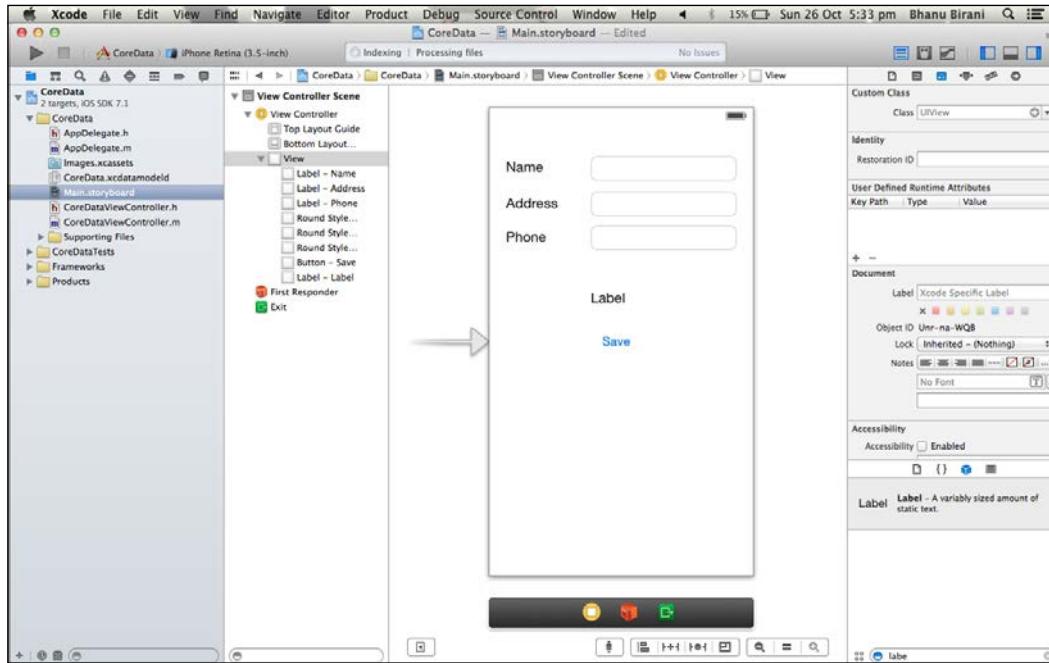


6. Next, edit the `AppDelegate.m` file and modify the `didFinishLaunchingWithOptions` method so that it simply returns YES instead of creating a window for the application (since we are now using a storyboard file for the user interface, this is no longer needed):

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

7. Now, we need a new Objective-C file. So add a new file from the menu and name it `CoreDataViewController`.

8. In storyboard, drag ViewController from the interface builder and design it as shown in the following screenshot:

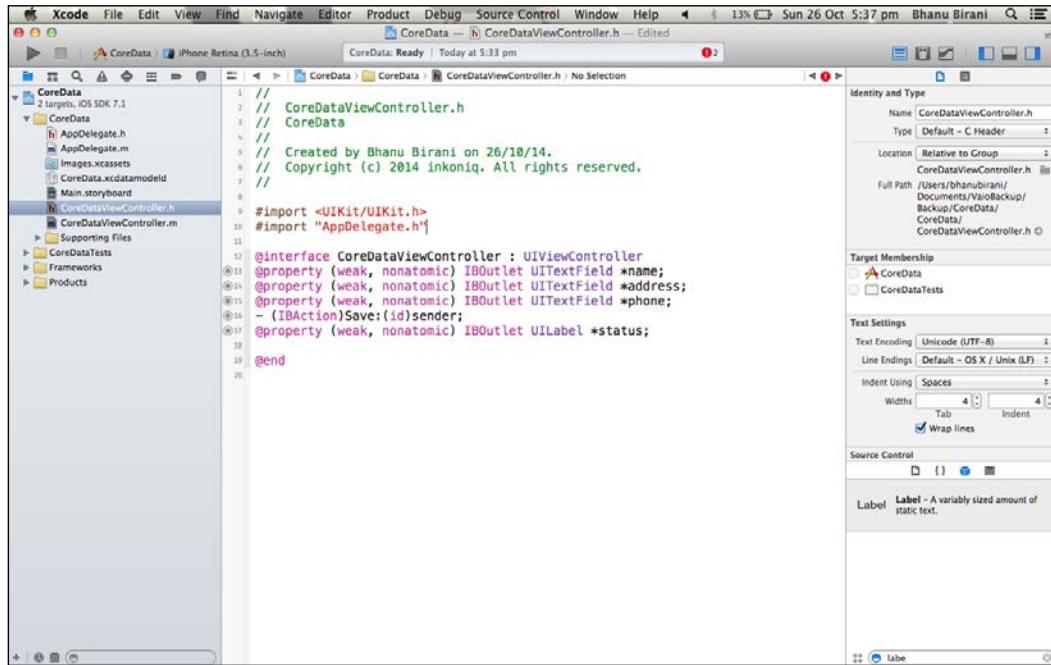


Don't forget to give a name to this view from the inspector editor on the right-hand side of Xcode.

9. Now, it is time to make connections between the storyboard and the newly added Objective-C file, (`CoreDataViewController.h`). We also need to import the `AppDelegate.h` class to our new class as follows:

```
#import <AppDelegate.h>
```

Our .h class probably looks the following screenshot:



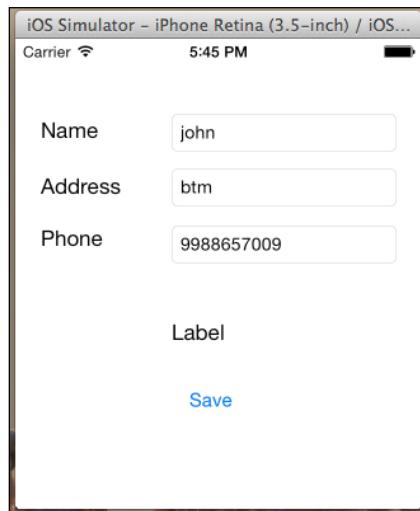
- When the user touches the **Save** button, the `Save` method is called. It is within this method, therefore, that we must implement the code to obtain the managed object context, and create and store managed objects containing the data entered by the user. Select the `CoreDataViewController.m` file, scroll down to the template `Save` method, and implement the code as follows:

```
- (IBAction)Save:(id)sender
{
    CoreDataAppDelegate *appDelegate =
    [[UIApplication sharedApplication] delegate];

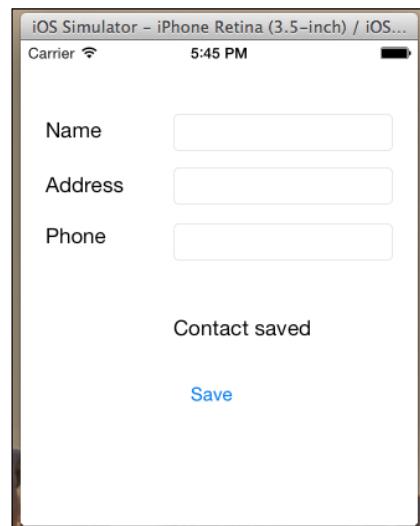
    NSManagedObjectContext *context =
    [appDelegate managedObjectContext];
    NSManagedObject *newContact;
    newContact = [NSEntityDescription
        insertNewObjectForEntityForName:@"Contacts"
        inManagedObjectContext:context];
    [newContact setValue: _name.text forKey:@"name"];
    [newContact setValue: _address.text forKey:@"address"];
    [newContact setValue: _phone.text forKey:@"phone"];
    _name.text = @"";
    _address.text = @"";
    _phone.text = @"";
}
```

```
NSError *error;
[context save:&error];
_status.text = @"Contact saved";
}
```

11. The final step is to build and run the application. Click on the **Run** button located in the toolbar of the main Xcode project window and enter the example text in to the text fields, as shown in the following screenshot:



12. After tapping on the **Save** button, our data is saved in the database and our label is changed into **Contact saved**, as shown in the following screenshot:



Social integration in our application

Integration of social sites in an app is very common nowadays. In iOS 6, a new framework is introduced known as **Social.framework**. The social framework lets us integrate social networking services such as Facebook and Twitter in our application. We don't have to download any SDK or use any API; Social framework handles everything. One class is important for integrating social networking services: **SLComposeViewController**. The **SLComposeViewController** class presents a standard view for users to compose tweet or Facebook posts. This class also allows users to share location without any additional code.

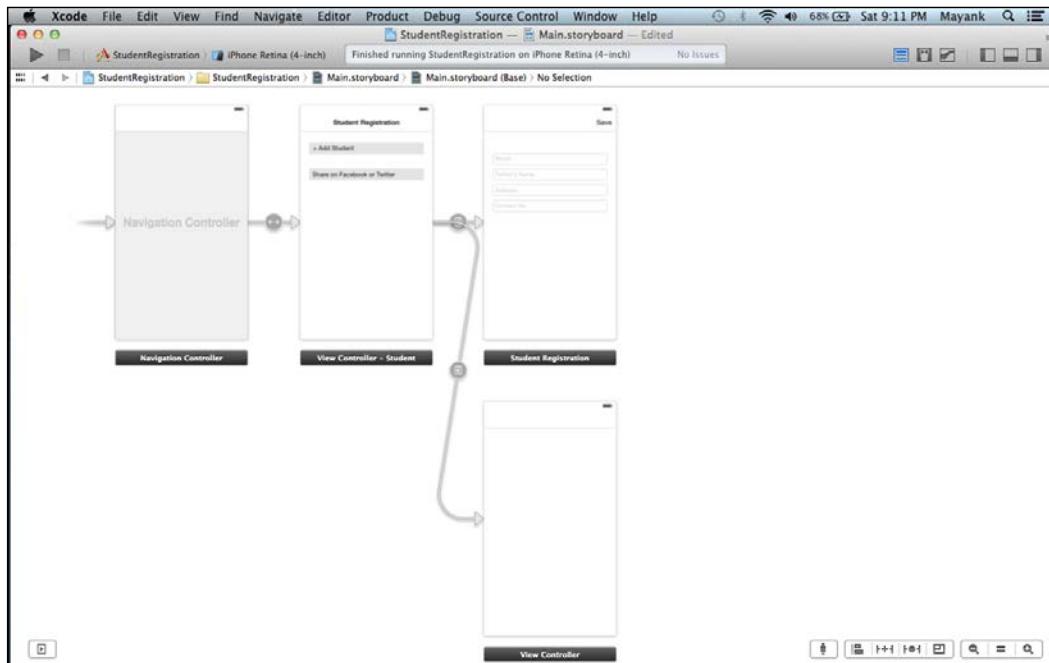
The work of the **SLComposeViewController** class is to get the Twitter or Facebook screen on our device to enable sharing.

Now, in order to handle the API and connection, another class is available in Social framework: **SLRequest**. The **SLRequest** class allows iOS applications to interact directly with social network APIs through HTTP-based requests.

Let's carry out one simple activity to understand social integration in a better way:

1. Again, continue with the previous project. Open that project and add one new Objective-C class as we did earlier and name it.
2. Drag a new view controller near or below the third view controller and add two buttons named Tweet and Facebook respectively.
3. Add one more button in the second view below the **Add Student** button and name it (for example, Share on Facebook or Twitter).
4. Now, give the new view (the fourth view) the same name as the new class in the inspector element.

5. Click on the **Share on Facebook or Twitter** button and choose the action and link it to the fourth (new) view. Storyboard will look like this:



6. Link both the buttons (**Tweet** and **Facebook**) to the new class.

7. Add a new framework, `Social.framework`, to our code.

8. Import the header to the new class `.h` file:

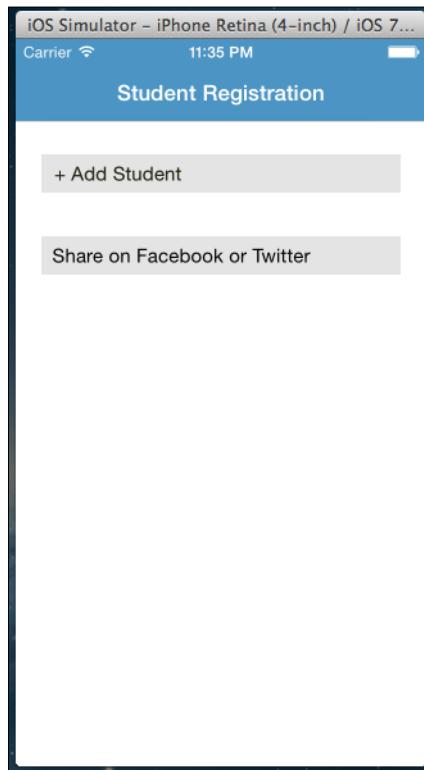
```
#import<Social/Social.h>
```

9. Add the following code in the new class `.m` file:

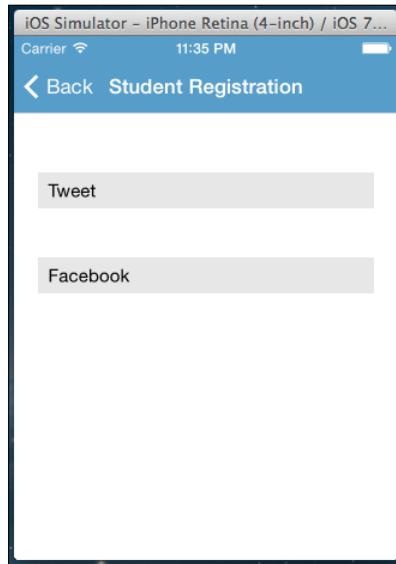
```
- (IBAction)Tweet:(id)sender {  
  
    SLComposeViewController *tweetSheet =  
        [SLComposeViewController  
        composeViewControllerForServiceType:  
        SLServiceTypeTwitter];  
    [tweetSheet setInitialText:@" Share on Twitter -_- "];  
    [self presentViewController:tweetSheet  
    animated:YES completion:nil];  
  
}
```

```
- (IBAction)PostOnFB:(id)sender {  
  
    SLComposeViewController *controller =  
    [SLComposeViewController  
        composeViewControllerForServiceType:  
        SLSERVICETypeFacebook];  
  
    [controller setInitialText:@"Post to your wall -_- "];  
    [self presentViewController:controller animated:YES  
        completion:nil];  
}
```

10. Compile and run the project. The output will look like the following screenshot:



11. Click on the **Share on Facebook or Twitter** button. The output will be as follows:



12. Now, click on the **Tweet** button. The output will look like this:



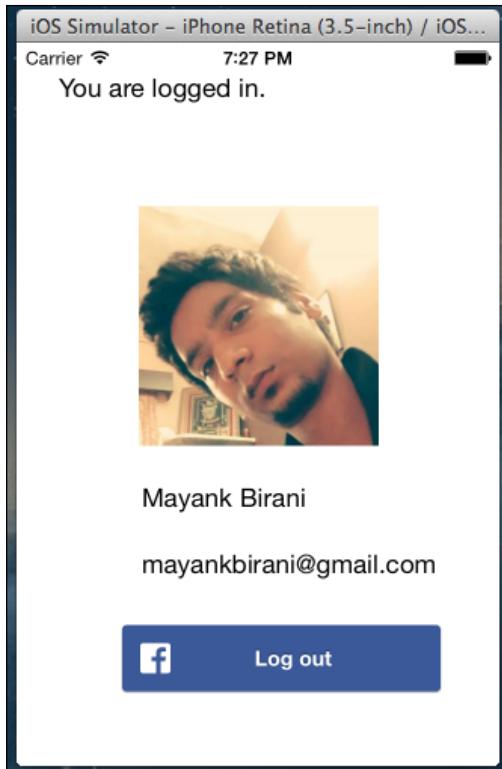
This output appears because we didn't manage the settings in the simulator. When you set your account in **Settings**, it will be able to post our tweet on Twitter; or when we run it on the actual device, it will work properly.

13. Now, click on the **Facebook** button, the output will be as follows:



This activity is only for sharing something on our post. For creating a login we need Facebook SDK. Logging in with Facebook not only allows you to attach a social characteristic into your app, but it can also be used as a login system instead of creating a custom one. To create a Facebook login, we need to download the Facebook SDK from <https://developers.facebook.com/docs/ios>. There are a lot of classes available in this SDK. Most tasks are handled by those classes. Developers don't have to bother about everything. They are only required to add the login view to the view controller.

This activity is a little complex and the method is also long, so you can go through the link: <http://www.appcoda.com/ios-programming-facebook-login-sdk/> and get appropriate examples of Facebook login. Follow the steps given in the link and you will get an output similar to the following screenshot:



Summary

In this chapter, you learned some more about the UI and how to integrate frameworks in your project. You also made a simple app, Student Registration, by exploring each section. And, at the end of the chapter, you created a fully-loaded app with a UI and its functionalities. You also learned about databases with social media integration.

In next chapter, you will learn about new frameworks and APIs and various new concepts.

4

APIs Introduced in iOS 7

API stands for **application programming interface**. An API is a set of commands, functions, sections of code, and protocols that programmers can use in their applications. They are predefined functions available for programmers to use instead of writing them from scratch. While APIs make a programmer's task easier, an API is a software-to-software interface, not a user interface. With APIs, applications communicate with each other without any user intervention. Some popular APIs are Google Map APIs, Twitter APIs, YouTube APIs, and so on.

This chapter will cover the following topics:

- Using AirDrop to remotely send/receive files
- iOS's first native game engine
- Text kit to manage your typography
- Sample projects

There are a lot of major updates done by Apple for developers to incorporate into their app. The user interface has been completely redesigned. iOS 7 introduces a new animation system to create 2D and 2.5D games. Multitasking enhancements, peer-to-peer connectivity, and many other important features have been added.

Using AirDrop

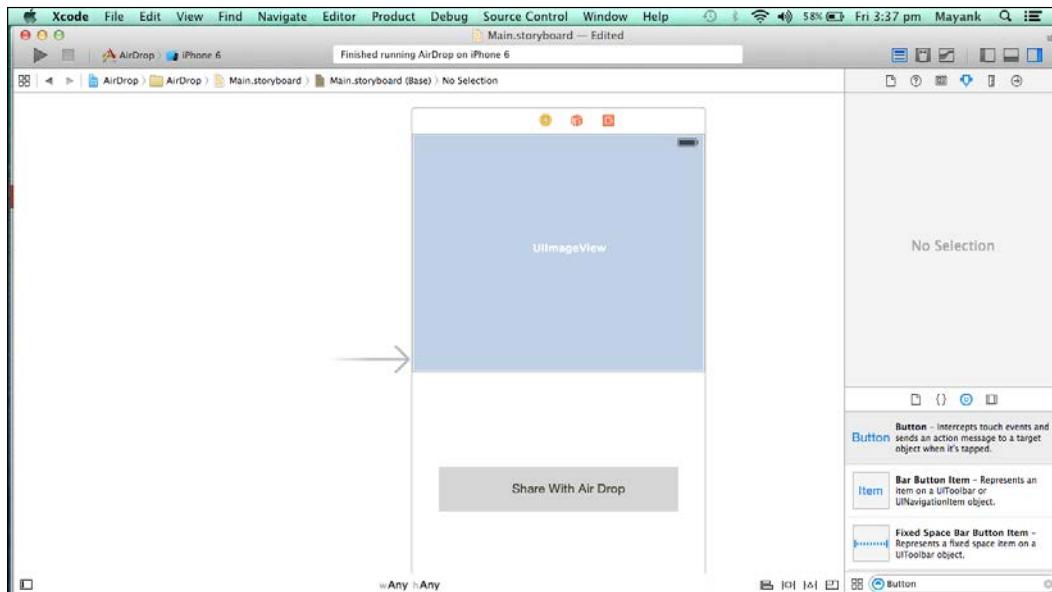
Apple introduced a new feature in iOS 7 called **AirDrop**. AirDrop is used to easily share pictures, contacts, notes, and more with other nearby iOS devices. It uses Bluetooth to detect nearby devices. When a connection is established via Bluetooth, it'll create an ad hoc Wi-Fi network to link the two devices together. There is a class called `UIActivityViewController` available to integrate AirDrop in our apps. We just need to tell this class which objects we want to share, and it handles the rest. The `UIActivityViewController` class is a standard view controller class that provides several standard services, such as copying items to the clipboard, sharing content on social media sites, sending items via messages, and so on. In iOS 7 SDK, this class is served with the built-in AirDrop feature:

```
UIImage *Image1 = [UIImage imageNamed:@"Image.png"] ;

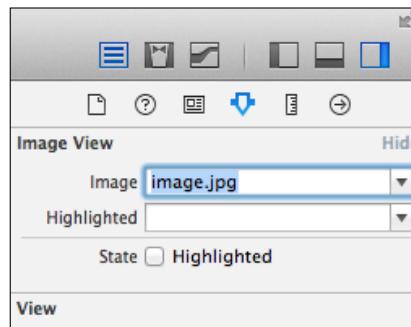
UIActivityViewController *activityVC = [[UIActivityViewController
alloc]
initWithActivityItems:[NSArray arrayWithObjects:@"Share Image",
Image1,
nil] applicationActivities:nil];
```

Let's understand this feature with a small activity:

1. Open Xcode and make a new project.
2. Drag the image view and button into the storyboard from the interface builder. Our storyboard will look like the following screenshot:



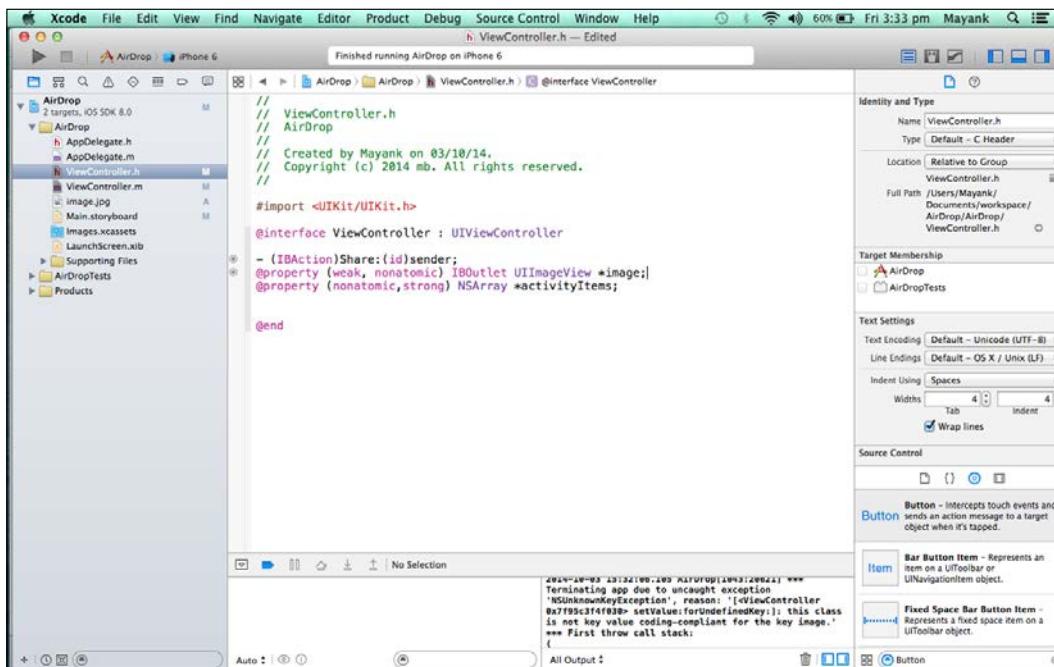
3. Drag any image in Xcode below the class files that you want to use.
4. Select the image view from the storyboard and move to the Attribute Inspector. Then, in the **Image** textbox, give it the same name as that of the image we dragged in Xcode, as shown in the following screenshot:



5. Now, link the image view and the button to `viewController.h` and create one array in which we will store the content that we want to transfer from AirDrop.

```
- (IBAction)share:(id)sender;
@property (weak, nonatomic) IBOutlet UIImageView *image;
@property (nonatomic,strong) NSArray *activityItems;
```

The following screenshot will show the content that we want to transfer:



6. Move to `viewController.m` and add the following code to the `viewDidLoad` method:

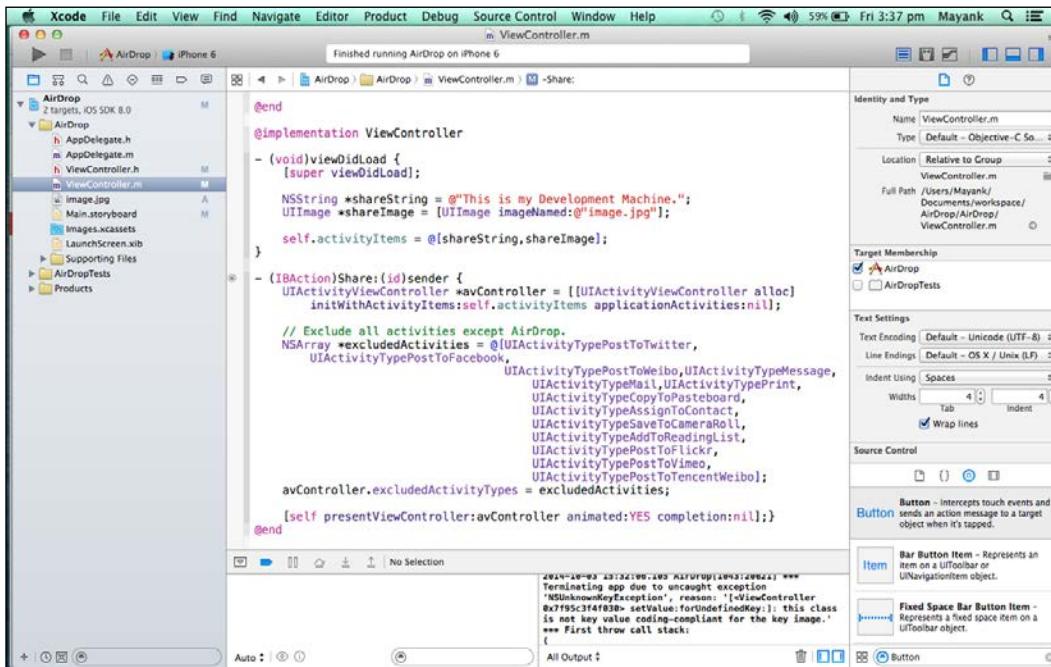
```
NSString *shareString = @"This is my Development Machine.";  
UIImage *shareImage = [UIImage imageNamed:@"image.jpg"];  
self.activityItems = @[shareString,shareImage];
```

In the preceding code, we created one string that we want to share and created one `UIImage` object that stores our image.

Then, we added both of them to our defined array. Now, let's add some code in our button event as follows:

```
UIActivityViewController *avController =  
    [[UIActivityViewController alloc]  
     initWithActivityItems:self.activityItems  
     applicationActivities:nil];  
  
NSArray *excludedActivities = @[ UIActivityTypePostToTwitter,  
                                 UIActivityTypePostToFacebook,  
                                 UIActivityTypePostToWeibo,  
                                 UIActivityTypeMessage,  
                                 UIActivityTypeMail,  
                                 UIActivityTypePrint,  
                                 UIActivityTypePrint,  
                                 UIActivityTypeCopyToPasteboard,  
                                 UIActivityTypeAssignToContact,  
                                 UIActivityTypeSaveToCameraRoll,  
                                 UIActivityTypeAddToReadingList,  
                                 UIActivityTypePostToFlickr,  
                                 UIActivityTypePostToVimeo,  
                                 UIActivityTypePostToTencentWeibo];  
  
avController.excludedActivityTypes = excludedActivities;  
  
[self presentViewController:avController animated:YES  
                  completion:nil];  
}
```

In the preceding code, we created `UIActivityViewController` with our `activityItems` array. With the `excludedActivityTypes` property, we excluded all the activities that are not needed, leaving AirDrop as the only sharing option. Finally, we presented the activity view controller. The following screenshot illustrates the preceding code snippet:



The screenshot shows the Xcode interface with the following details:

- Project Navigator:** Shows the project structure for "AirDrop" with targets "iPhone 6" and "AirDrop".
- Editor:** Displays the code for `ViewController.m`. The code initializes `UIActivityViewController` with a share string and image, and excludes other sharing options except for AirDrop.
- Utilities:** Shows the `Identity and Type` panel for `ViewController.m`, indicating it's a Default - Objective-C Source File located relative to the group. It also shows target membership for "AirDrop" and "AirDropTests".
- Text Editor:** Shows the error output from the build process, indicating a terminating app due to an uncaught exception.

```

ViewController.m

@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString *shareString = @""This is my Development Machine."";
    UIImage *shareImage = [UIImage imageNamed:@"image.jpg"];
    self.activityItems = @[shareString,shareImage];
}
- (IBAction)Share:(id)sender {
    UIActivityViewController *avController = [[UIActivityViewController alloc]
        initWithActivityItems:self.activityItems applicationActivities:nil];
    // Exclude all activities except AirDrop.
    NSArray *excludedActivities = @[[UIActivityTypePostToTwitter,
        UIActivityTypePostToFacebook,
        UIActivityTypePostToWeibo,UIActivityTypeMessage,
        UIActivityTypeMail,UIActivityTypePrint,
        UIActivityTypeAssignToContact,
        UIActivityTypeSaveToCameraRoll,
        UIActivityTypeAddToReadingList,
        UIActivityTypePostToFlickr,
        UIActivityTypePostToVimeo,
        UIActivityTypePostToTencentWeibo];
    avController.excludedActivityTypes = excludedActivities;
    [self presentViewController:avController animated:YES completion:nil];
}

```

7. It is now time to compile and run our program. After execution, our simulator will look like the following screenshot:



8. Tap on the **Share With Air Drop** button. Our simulator will now look like this:



Here, no options are available because we have excluded all the options except the AirDrop option, and the simulator does not support AirDrop. When we put this code on the real device, it will show us the devices that are available for sharing and the devices that have AirDrop.

SpriteKit

Apple has launched its first game engine, SpriteKit, that allows us to create games for iOS without being dependent on third-party game libraries. It is very powerful and inclined towards the traditional iOS framework approach when it comes to its usage. It's also very easy to adopt and learn. In addition to this, it supports lots of features such as physics simulations, a texture atlas, gravity, restitution, and game center support. Moreover, it comes with very rich developer documentation at the **Apple Development Center**. It's very useful and well written. You might need to understand the anatomy of game development first to get started in SpriteKit. So, there are two basic and most important terms here: one is Scenes, and the other one is Sprites. Scenes can be considered as the levels in the games. So, in any game, the score layer, the **HUD (Heads-Up Display)** layer, and the gameplay layer can act as different scenes. However, any object in the scene, such as the player or enemy, can be considered as a sprite.

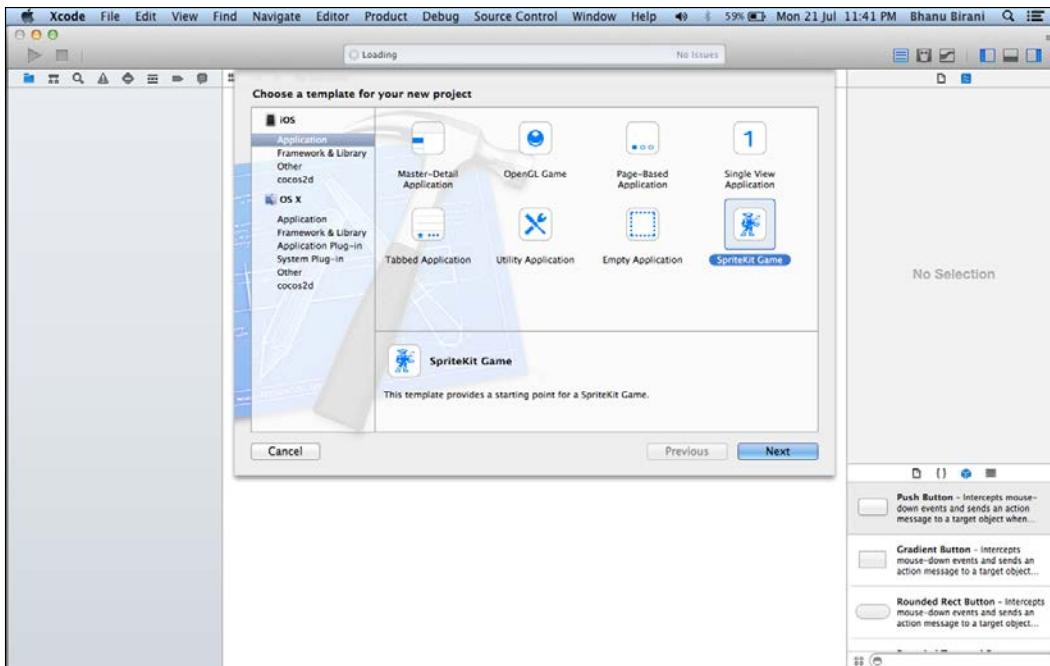
The iOS native game framework

Apple has introduced its own native 2D game framework called SpriteKit. SpriteKit is a great 2D game engine that offers support for sprites, animations, filters, and masking. Most importantly, it also offers support for the physics engine to provide a real-world simulation for the game.

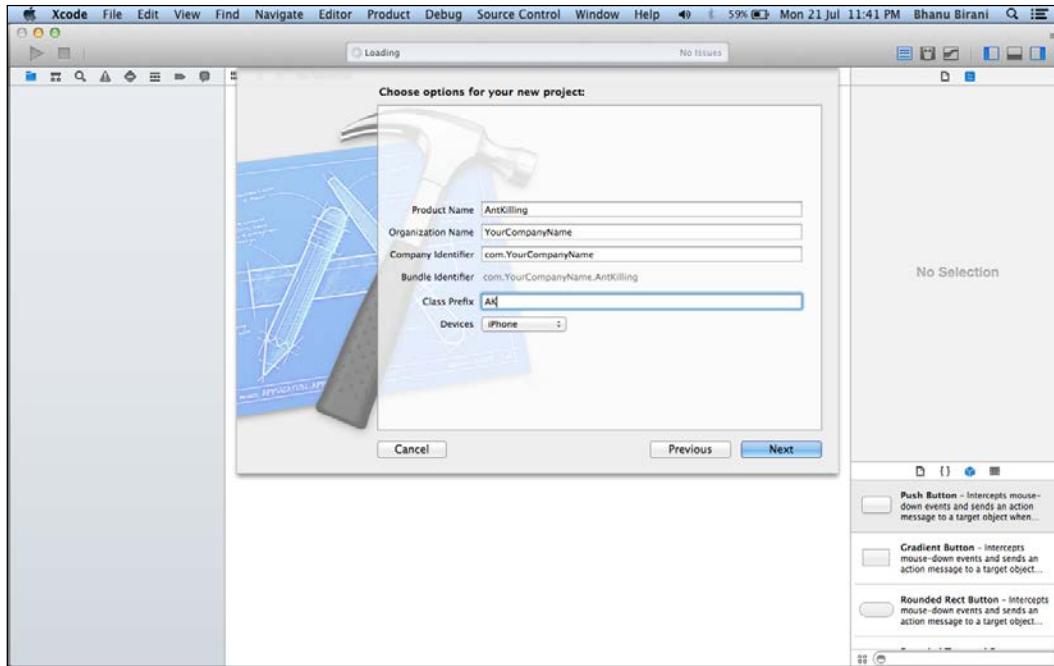
Apple provides a sample game called *Adventure Game* to get started with SpriteKit. You can check out the sample SpriteKit game project at <https://developer.apple.com/library/ios/documentation/GraphicsAnimation/Conceptual/CodeExplainedAdventure/AdventureArchitecture/AdventureArchitecture>. This sample project provides a glimpse into the capability of this framework. However, the project is complicated to understand; for learning purposes, you just want to create something simple to understand and learn. To have a deeper understanding of SpriteKit-based games, we will build a bunch of mini games in this book. To understand the basics of SpriteKit game programming, we will build a mini *AntKilling* game in this chapter.

Let's start building the *AntKilling* game by performing the following steps:

1. Start your Xcode. Navigate to **File | New | Project**. Then, in the prompt window, navigate to **iOS | Application | SpriteKit Game** and click on **Next**, as shown in the following screenshot:



2. Fill in all the project details in the prompt window and enter **AntKilling** as **Product Name**. Also provide the name of your organization and select the device name as **iPhone** and **Class Prefix** as **AK**. Click on **Next** as shown in the following screenshot:



3. Select a location on your drive to save the project and then click on **Create**.

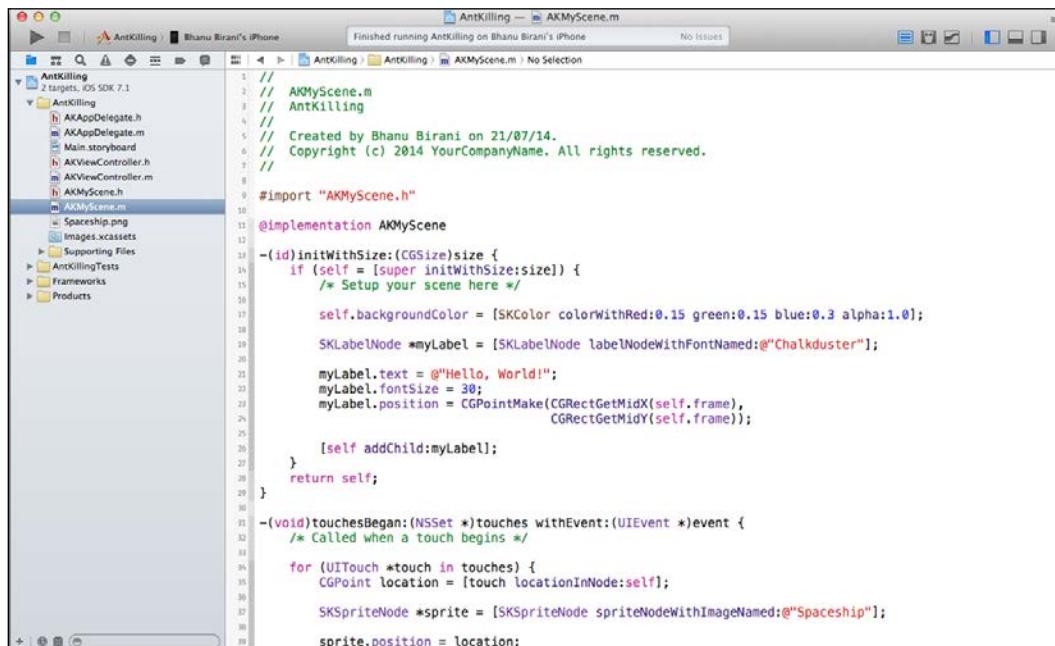
4. Then, build the sample project to check its output. Once you build and run the project with the Play button, you will see the following screen on your device:



As you can see, the sample SpriteKit project plays a label with a background color. SpriteKit works on the concept of scenes, which can be understood as the levels or screens of the game. There can be multiple scenes working at the same time; for example, there can be a gameplay scene, HUD scene, and the score scene running at the same time in the game.

Now, we can look at more detailed arrangements for the starter project by performing the following steps:

1. In the main directory, you already have one scene created by default; this scene is called `AKMyScene`. Now, click on `AKMyScene.m` to explore the code for adding the label on the screen. You should see something similar to following screenshot:

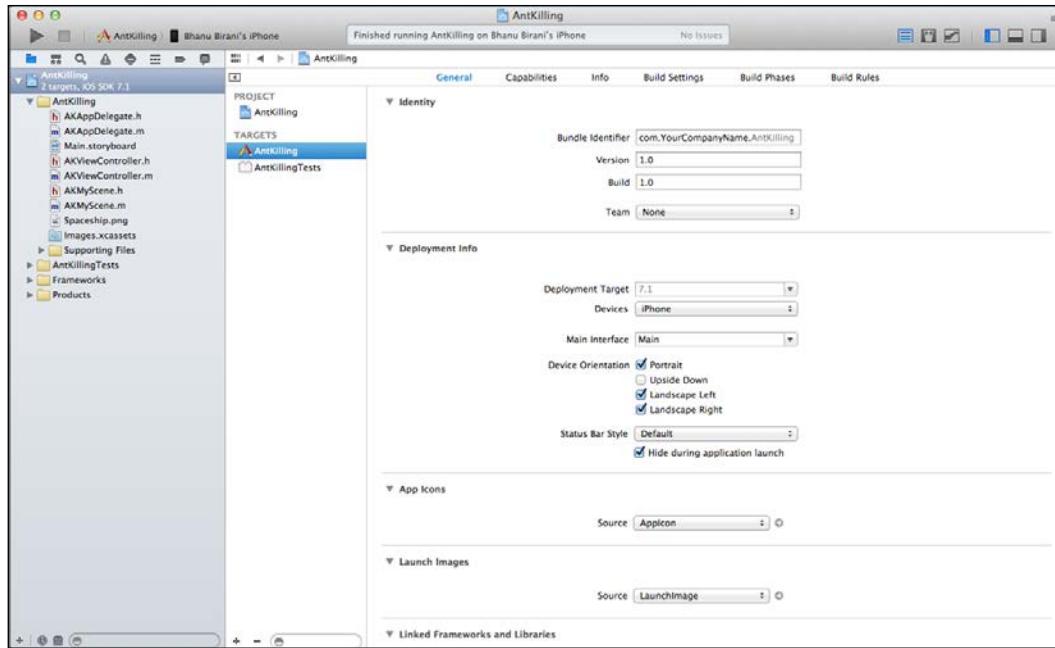


```
AntKilling — AKMyScene.m
Bhanu Birani's iPhone
Finished running AntKilling on Bhanu Birani's iPhone
No Issues

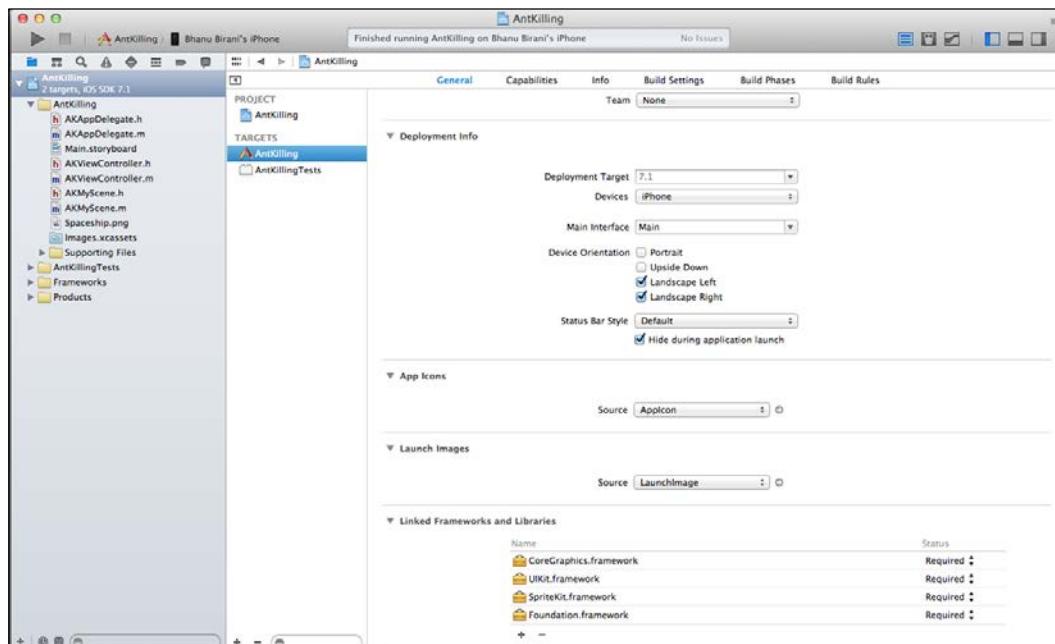
1 // AKMyScene.m
2 // AntKilling
3 // Created by Bhanu Birani on 21/07/14.
4 // Copyright (c) 2014 YourCompanyName. All rights reserved.
5
6 #import "AKMyScene.h"
7
8 @implementation AKMyScene
9
10 -(id)initWithSize:(CGSize)size {
11     if (self = [super initWithSize:size]) {
12         /* Setup your scene here */
13
14         self.backgroundColor = [SKColor colorWithRed:0.15 green:0.15 blue:0.3 alpha:1.0];
15
16         SKLabelNode *myLabel = [SKLabelNode labelNodeWithFontNamed:@"Chalkduster"];
17
18         myLabel.text = @"Hello, World!";
19         myLabel.fontSize = 30;
20         myLabel.position = CGPointMake(CGRectGetMidX(self.frame),
21                                         CGRectGetMidY(self.frame));
22
23         [self addChild:myLabel];
24     }
25
26     return self;
27 }
28
29 -(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
30     /* Called when a touch begins */
31
32     for (UITouch *touch in touches) {
33         CGPoint location = [touch locationInNode:self];
34
35         SKSpriteNode *sprite = [SKSpriteNode spriteNodeWithImageNamed:@"Spaceship"];
36
37         sprite.position = location;
38
39 }
```

Now, we have to update this file with our code to create our *AntKilling* game. We have to fulfill a few prerequisites to get started with the code; for example, we have to lock the orientation to landscape as we want a landscape-orientation game.

2. To change the orientation of the game, open the **AntKilling** project settings and navigate to **TARGETS | General**. You will see something similar to the following screenshot:



- Now, in the **General** tab, uncheck **Portrait** under the **Device Orientation** option so that the final settings look similar to the following screenshot:



4. Now, build and run the project. You will be able to see the app started in the landscape orientation.



5. Now, it's time to update `AKMyScene` to hold our ant sprites. Just download and open all the resources you got for this chapter and add them to your Xcode project.
6. While adding the resources to the Xcode project, make sure that the selected target is **AntKilling** and that the **Copy items into destination group folder** is checked, if needed.
7. Now, delete all the existing code from `AKMyScene.m` and make it look similar to following screenshot:

```
#import "AKMyScene.h"

@interface AKMyScene : NSObject

@property (nonatomic) SKSpriteNode *ant;

@end

@implementation AKMyScene

-(id)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {
        /* Setup your scene here */

        NSLog(@"Size: %@", NSStringFromCGSize(size));
        self.backgroundColor = [SKColor colorWithRed:1.0 green:1.0 blue:1.0 alpha:1.0];

        self.ant = [SKSpriteNode spriteNodeWithImageNamed:@"ant.jpg"];
        self.ant.position = CGPointMake(self.size.width/2, self.size.height/2);
        [self addChild:self.ant];
    }
    return self;
}

@end
```

Now, here is the explanation of what we did so far:

1. First, we created a private interface to declare the private variables:

```
@interface AKMyScene ()  
@property (nonatomic) SKSpriteNode *ant;  
@end
```

2. Then, in the `init` method, we printed a log to print the size of the screen:

```
 NSLog(@"%@", NSStringFromCGSize(size));
```

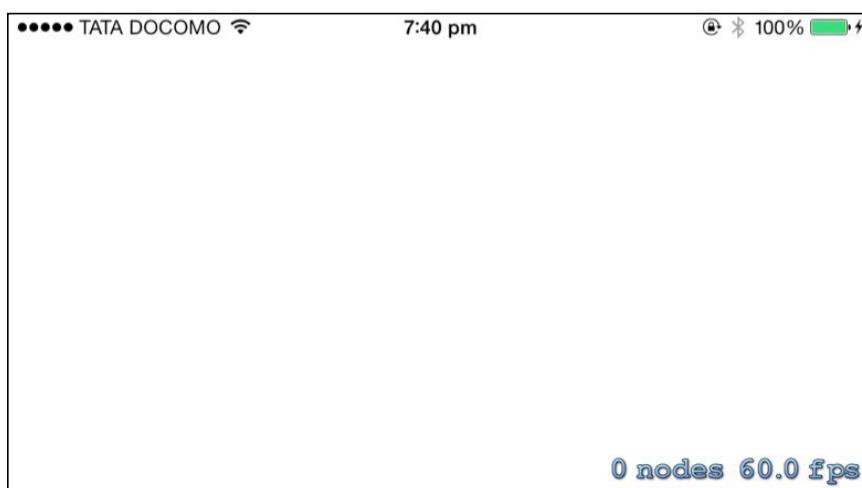
3. We changed the screen background color to white using the following line of code:

```
self.backgroundColor = [SKColor colorWithRed:1.0 green:1.0  
blue:1.0  
alpha:1.0];
```

4. In the following line of code, we created a sprite object using the `spriteNodeWithImageNamed` method and passed the image name to it. Then, we positioned the sprite object to (100, 100) of the screen, which is in the bottom-left corner of the screen. Then, finally, we added it as a child method:

```
self.ant = [SKSpriteNode spriteNodeWithImageNamed:@"ant.  
jpg"];  
self.ant.position = CGPointMake(100, 100);  
[self addChild:self.ant];
```

8. Now, build and run your application. You will see something similar to the following screenshot:



Now, as you can see, the screen color has changed to white, but there is no ant on the screen. This means there is something wrong with the code. So now, let's check our logs, which should print the following:

```
2014-07-22 19:13:27.019 AntKilling[1437:60b] Size: {320, 568}
```

So, we found out that the scene size is wrong; it should print 568 as the width and 320 as the height, and it is printing the opposite.

9. To debug this, navigate to the `viewDidLoad` method of your `AKViewController.m`. This will be something similar to the one shown in the following screenshot:

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

    // Configure the view.
    SKView * skView = (SKView *)self.view;
    skView.showsFPS = YES;
    skView.showsNodeCount = YES;

    // Create and configure the scene.
    SKScene * scene = [AKMyScene sceneWithSize:skView.bounds.size];
    scene.scaleMode = SKSceneScaleModeAspectFill;

    // Present the scene.
    [skView presentScene:scene];
}
```

So, from this method, we can see that our scene absorbs the size from the bounds of the view, and this `viewDidLoad` method is invoked even before the view has been added to the view hierarchy. Thus, it has not responded to the layout changes. As a result of the inconsistent view bounds, our scene is getting started with the wrong bounds.

10. To solve this issue, we have to move the scene startup code in the `viewWillLayoutSubviews` method.

11. After removing the code from the `viewDidLoad` method and pasting it to `viewWillLayoutSubviews`, the code file will look similar to the one shown in the following screenshot:

```
#import "AKViewController.h"
#import "AKMyScene.h"

@implementation AKViewController

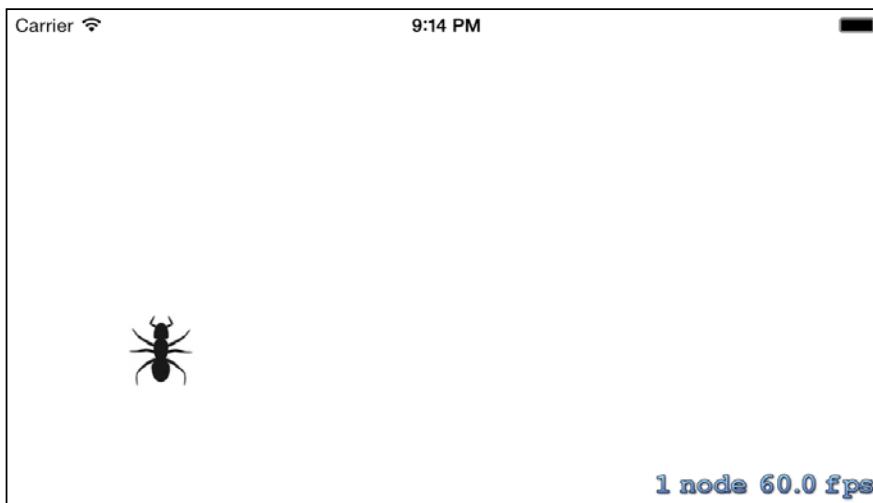
- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (void)viewWillLayoutSubviews
{
    // Configure the view.
    SKView * skView = (SKView *)self.view;
    skView.showsFPS = YES;
    skView.showsNodeCount = YES;

    // Create and configure the scene.
    SKScene * scene = [AKMyScene sceneWithSize:skView.bounds.size];
    scene.scaleMode = SKSceneScaleModeAspectFill;

    // Present the scene.
    [skView presentScene:scene];
}
```

12. Now, again, build and run the app. You will see the following output:



Congrats! You have fixed the issue; now, your ant has appeared on the screen at your given location. If you observe closely, you can see that the status bar is coming on the top of the game, and this is not looking great.

13. To remove the status bar from the screen, open your `AntKilling-Info.plist` file and add the `UIViewControllerBasedStatusBarAppearance` attribute and the value as `NO`. Your `.plist` file should be similar to following screenshot:

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
Localization native development r...	String	en
Bundle display name	String	<code> \${PRODUCT_NAME}</code>
Executable file	String	<code> \${EXECUTABLE_NAME}</code>
Bundle identifier	String	<code> com.YourCompanyName.\${PRODUCT_NAME}Identifier</code>
InfoDictionary version	String	6.0
Bundle name	String	<code> \${PRODUCT_NAME}</code>
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone envir...	Boolean	YES
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
Status bar is initially hidden	Boolean	YES
View controller-based status...  	Boolean	NO  
► Supported interface orientations	Array	(2 items)

14. Build and run your project again. You should be able to see the game without the status bar now, as shown in the following screenshot:



This looks perfect now; our ant is residing on the screen as expected. So now, our next objective is to animate the ant when we tap on it.

15. To accomplish this, we need to add the following code in the `AKMyScene.m` file just below our `initWithSize` method:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    CGPoint positionInScene = [touch locationInNode:self];
    SKSpriteNode *touchedNode = (SKSpriteNode *)[self
        nodeAtPoint:positionInScene];
    if (touchedNode == self.ant) {
        SKAction *sequence = [SKAction sequence:@[[SKAction
            rotateByAngle:degreeToRadian(-3.0f) duration:0.2],
            [SKAction rotateByAngle:0.0
            duration:0.1],
            [SKAction rotateByAngle:
            degreeToRadian(3.0f) duration:0.2]]];
        [touchedNode runAction:[SKAction
            repeatActionForever:sequence]];
    }
}
float degreeToRadian(float degree) {
    return degree / 180.0f * M_PI;
}
```

The final code file will look similar to the one shown in the following screenshot:

```
-(id)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {
        /* Setup your scene here */

        NSLog(@"Size: %@", NSStringFromCGSize(size));
        self.backgroundColor = [SKColor colorWithRed:1.0 green:1.0 blue:1.0 alpha:1.0];

        self.ant = [SKSpriteNode spriteNodeWithImageNamed:@"ant.jpg"];
        self.ant.position = CGPointMake(100, 100);
        [self addChild:self.ant];
    }
    return self;
}

-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    CGPoint positionInScene = [touch locationInNode:self];
    SKSpriteNode *touchedNode = (SKSpriteNode *)[self nodeAtPoint:positionInScene];
    if (touchedNode == self.ant) {
        SKAction *sequence = [SKAction sequence:@[[SKAction rotateByAngle:degreeToRadian(-3.0f) duration:
        0.2],
            [SKAction rotateByAngle:0.0 duration:0.1],
            [SKAction rotateByAngle:degreeToRadian(3.0f) duration:0.2
            ]]];
        [touchedNode runAction:[SKAction repeatActionForever:sequence]];
    }
}
float degreeToRadian(float degree) {
    return degree / 180.0f * M_PI;
}
@end
```

Let's go through it line-by-line to understand what we have done so far:

1. To begin with, we added the - (void) touchesBegan: (NSSet *) touches withEvent: (UIEvent *) event method to the grab all the touches on the scene.
2. Now, in the function the first line allowed us to grab the UITouch
`*touch = [touches anyObject]; touch.`
3. In the next line, we grabbed the touch and converted it to the
`CGPoint positionInScene = [touch locationInNode:self];`
location.
4. Using the following line, we fetched the sprite that was touched:
`SKSpriteNode *touchedNode = (SKSpriteNode *)[self
nodeAtPoint:positionInScene];`
5. Once you have the sprite object, compare and check whether the selected object is the ant bug. If it's the ant bug, then animate the object by adding the following line of code:
`SKAction *sequence = [SKAction sequence:@[[SKAction
rotateByAngle:degreeToRadian(-3.0f) duration:0.2],
[SKAction
rotateByAngle:0.0 duration:0.1],
[SKAction
rotateByAngle:degreeToRadian(3.0f) duration:0.2]];
[touchedNode runAction:[SKAction
repeatActionForever:sequence]];`
16. Now, this code will animate the selected sprite. Build and run the project, and you will see the ant animating when we tap on it.

You will soon notice that, on tapping on the ant, it starts animating, but there is no way to stop this. So, let's add a way to stop this animation once you click anywhere on the scene. Go to the - (void) touchesBegan: (NSSet *) touches withEvent: (UIEvent *) event method and update it to the following code:

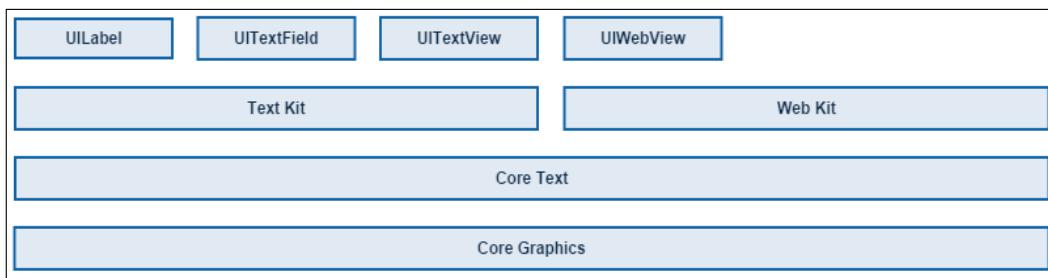
```
- (void) touchesBegan: (NSSet *) touches withEvent: (UIEvent *) event
{
    UITouch *touch = [touches anyObject];
    CGPoint positionInScene = [touch locationInNode:self];
    SKSpriteNode *touchedNode = (SKSpriteNode *)[self
nodeAtPoint:positionInScene];
    if (touchedNode == self.ant) {
        SKAction *sequence = [SKAction sequence:@[[SKAction
rotateByAngle:degreeToRadian(-3.0f) duration:0.2],
```

```
rotateByAngle:0.0 duration:0.1],  
[SKAction rotateByAn  
gle:degreeToRadian(3.0f) duration:0.2]];  
  
    [touchedNode runAction:[SKAction  
repeatActionForever:sequence]];  
} else {  
    [self.ant removeAllActions];  
}  
}  
}
```

If you observe closely, you can see that we have added an `if/else` condition to check whether the ant animates when we tap on it and whether all actions stop when we tap anywhere outside the screen. To stop all the actions on the sprite, we can use the `removeAllActions` method on the sprite.

Text Kit

The **UIKit** framework includes several classes whose purpose is to display text in a user's app, such as **UITextView**, **UITextField**, **UILabel**, and **UIWebView**. Text views, created from the **UITextView** class, are meant to display different types of text on screen. **UITextView** is a powerful layout engine called **Text Kit**. Text Kit is built on top of **Core Text**, so it provides the same speed and power as that of Core Text. **UITextView** is fully integrated with Text Kit; it provides editing and display capabilities that enable users to input text, specify formatting attributes, and view the results. The other Text Kit classes provide text storage and layout capabilities. The following diagram shows the position of Text Kit among other iOS text and graphic frameworks:



We know that Text Kit is a collection of many classes and functions. However, there are three primary classes of Text Kit:

- `NSTextStorage` (text storage class)
- `NSLayoutManager` (layout manager class)
- `NSTextContainer` (text container class)

NSTextStorage

The `NSTextStorage` class is responsible for storing all text attribute-related information, such as font, size, or paragraph information. The `NSTextStorage` class is a subclass of the `NSMutableAttributedString` class, and that's why it is responsible for keeping all text attributes. Besides this, its role also lies in making sure that all the edited text attribute data will remain consistent throughout all the management and editing operations that might be performed.

NSLayoutManager

The `NSLayoutManager` class, as its name implies, manages the way in which the text data stored in a `NSTextStorage` object will be displayed in a view. Its job is to handle and support any view object that the text can be displayed in and perform any required conversions of Unicode characters to glyphs so that each character properly appears on screen. An object of this class is notified by `NSTextStorage` about any modifications made to the text and its attributes, so every change immediately gets reflected in the corresponding view.

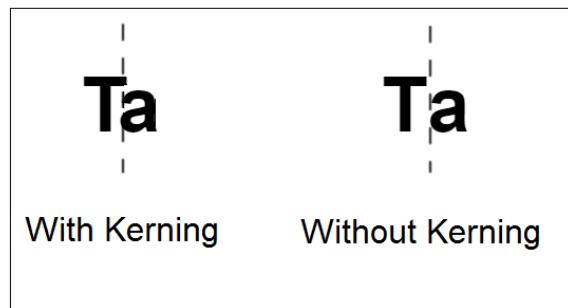
NSTextContainer

The `NSTextContainer` class actually specifies the view where the text will appear, and it handles information regarding this view (such as its frame or shape). However, a quite important characteristic of this class is its ability to keep an array of Bezier paths that define areas that should be excluded from the allowed region where the text will appear. This gives Text Kit the unique possibility of letting text flow around images or other non-text objects and allowing developers to display text in an impressive or demanding manner.

Text Kit is a set of classes and protocols in the `UIKit` framework that provide high-quality typographical services that enable apps to store, layout, and display text with all kinds of typesetting: kerning, ligatures, line breaking, and justification.

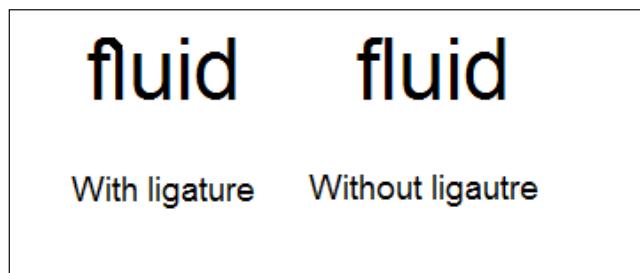
Kerning

All characters have different and irregular shapes, and these shapes must be placed exactly adjacent to each other. Text kit layout takes this into account; for example, a capital letter, *T*, has a lot of free space under its "wings" and moves the following lowercase letters closer. This results in significantly improved legibility of text, especially in longer pieces of writing. The following screenshot illustrates kerning:



Ligatures

This is an artistic feature of Text Kit. Some characters look nice when they are combined, for example when *f* is combined with *l*. These combined symbols are called glyphs.



Line breaking

To create lines from a string of glyphs, the layout engine must perform line breaking by finding a point at which to end one line and begin the next. In the text system, you can specify line breaking at either word or glyph boundaries. In Roman text, a word broken between glyphs requires the insertion of a hyphen glyph at the breakpoint.

Justification

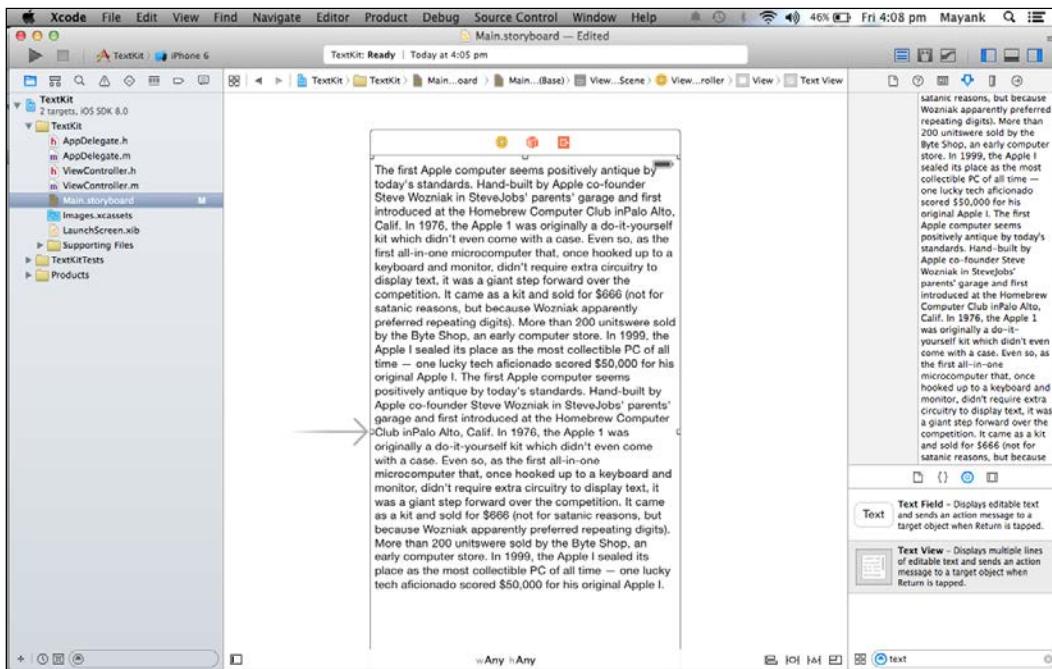
Lines of text can also be justified; for horizontal text, the lines are aligned on both right and left margins by varying interword and interglyph spacing, as shown in the following screenshot. The system performs alignment and justification, if requested, after the text stream has been broken into lines, hyphens have been added, and other glyph substitutions have been made.

Text Kit is part of the *UIKit* framework, and it consists of a collection of classes that enable developers to manage text and all of its attributes, as well as to display it in various ways, using new great methods and with a little effort. Indeed, prior to Text Kit and iOS 7, advanced text manipulation was really hard to be performed. In the need of modifying text details, such as font or layout attributes, one had to deal with *Core Text*, a powerful framework, yet hard to work with. Further than that, only *UIWebView* views were able to display formatted text. Things became better in iOS 6 with *attributed strings*, where *UITextView* views could also display rich text, but yet getting

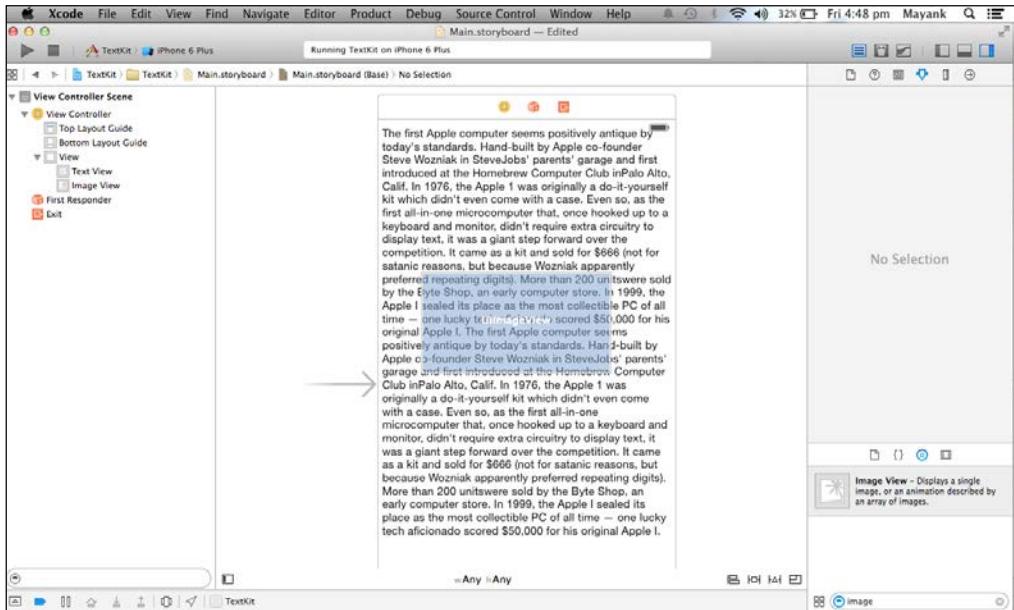
Justified Text

Let's move to exclusion paths again. We know that Text Kit contains many classes, and Text Container is one of them. One great feature of it is that it can store an array of `UIBezierPath` and force text to flow around these paths; as they are excluded from the text draw region, they are called **exclusion paths**. Let's understand how they work by performing the following steps:

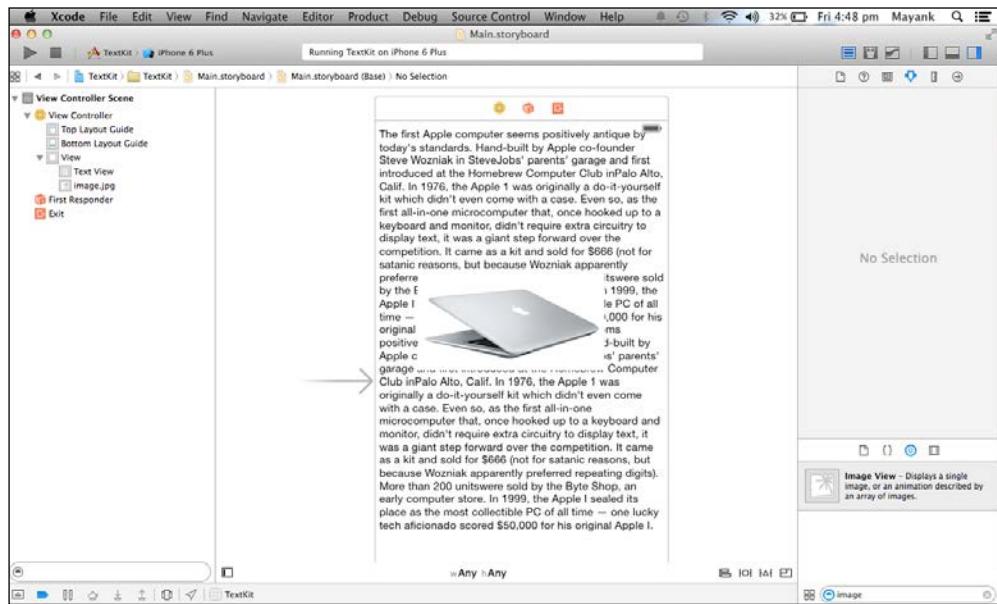
1. Open Xcode and create a new project. Go to the storyboard, drag a **Text Field** to view, and edit the text as you want (as shown in the following screenshot):



2. Drag an image view on the text view, as follows:



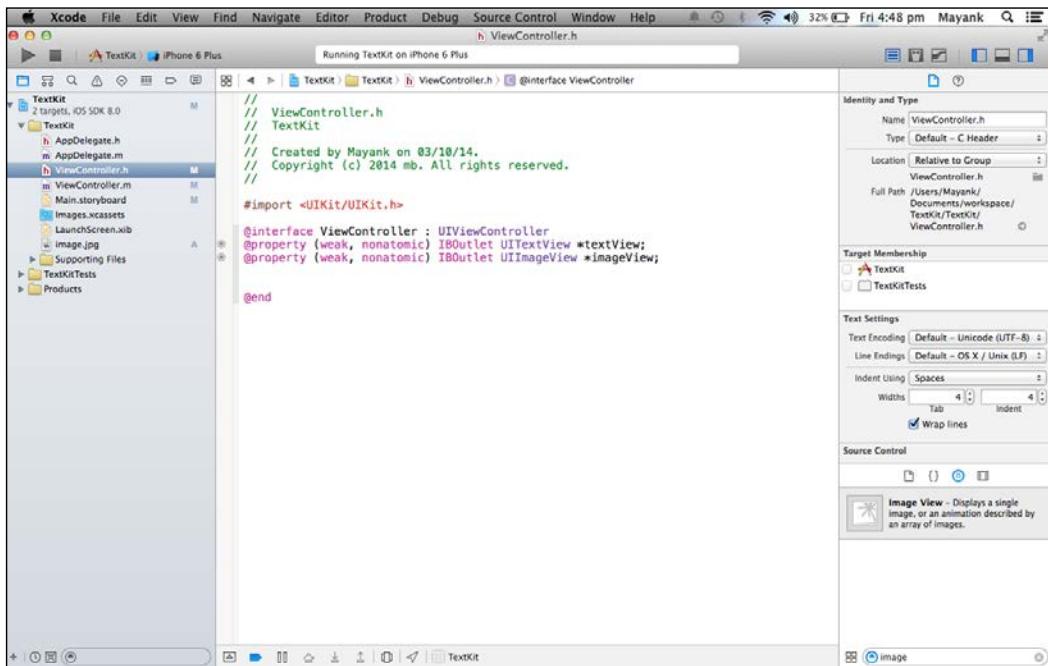
3. Add an image on the image view, as we did earlier. Drag any image in Xcode, as shown in the following screenshot, and give the same name to the image view via the Attribute Inspector as that of the dragged image:



4. Compile and run the code; our simulator will look like the following screenshot. It's cutting our text part below the image; this is not the output that we expected.



5. To implement the exclusive path text, we need to add some code. First of all, link the text field and image view to the viewController.h file, as shown in the following screenshot:



6. In the viewController.m file, write the following code in the viewDidLoad method:

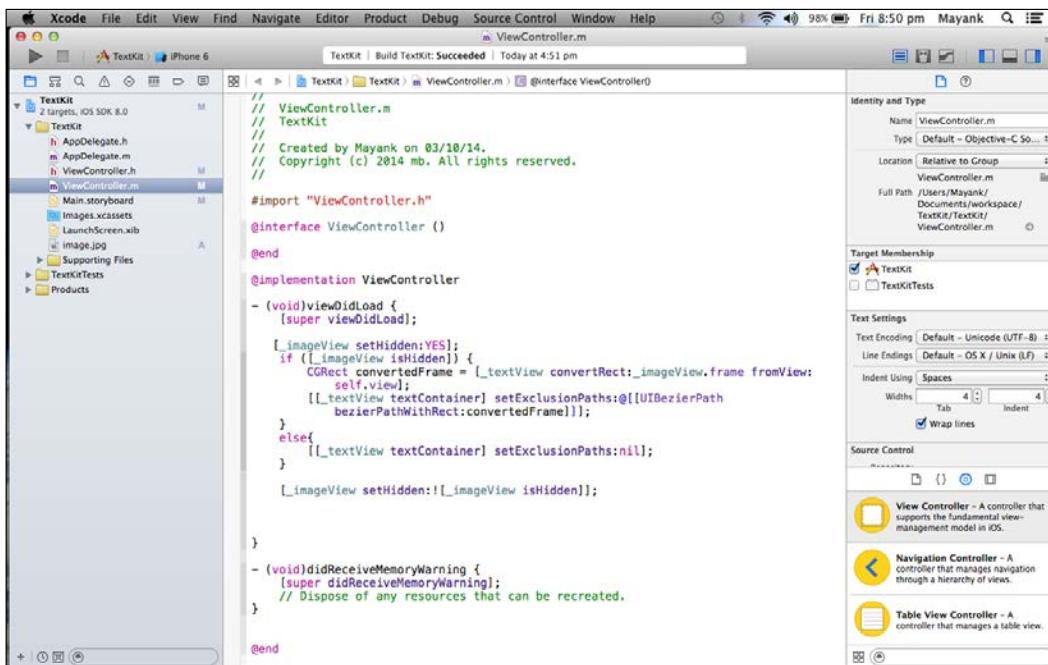
```
[_imageView setHidden:YES];
if ([_imageView isHidden]) {
    CGRect convertedFrame =
        [_textView convertRect:_imageView.frame fromView:self.view];
    [[_textView textContainer] setExclusionPaths:
     @[[UIBezierPath bezierPathWithRect:convertedFrame]]];
}
else{
    [[_textView textContainer] setExclusionPaths:nil];
}
[_imageView setHidden:![_imageView isHidden]];
```

In the preceding code, we first checked whether the image view is hidden or not (`_propertyName` is also a way to access any property other than `self.property`). If it's hidden (the initial state), then we have to set the exclusion path to the text container object of the text view so that the text flows around the image view and, of course, to make the image view visible:

```
CGRect convertedFrame = [_textView convertRect:_imageView.frame
fromView:self.view];
```

This line is used to convert the image view coordinates from the `self.view` view to the text view, so both the image view and the floating text exist in the same area. Next, using the `bezierPathWithRect:` class method of the `UIBezierPath` class, we created the Bezier path by the frame specified by its parameter; in our case, this is the frame (converted to the text view's coordinates) of the image view. We added this Bezier path to `NSArray`, and finally, we set the exclusion path. Note that, in the `else` case, we set the exclusion path to `nil`, because we don't want our text to flow when the image view is not there.

Our `viewController.m` file will look like the one shown in the following screenshot:



7. Run the code. Now, our text will not be cut because of the image. Look at the following screenshot; it's not cutting our text, and our text continues after the image in each line. Text gives space to the image through the `UIBezierPath` class and makes a different class for the image view. So, now, you can try it for round-shaped images or other images of any shape.



Summary

In this chapter, we learned a lot of interesting things, including how to create a game, add an image between text, and share it with AirDrop. All these APIs are introduced in iOS 7. After this chapter, try to extend all the activities performed here. This will clarify your concepts. In the next chapter, we will learn iOS 8 APIs, including PhotoKit, manual camera control, and the Handoff concept.

5

Frameworks Introduced with iOS 8

In this chapter, we will cover the following topics:

- Working with PhotoKit
- Handoff for resuming activities seamlessly among all devices

Working with PhotoKit

One new feature in iOS 8 has been revealed; it is named **PhotoKit**, also known as Photos Framework. It's a new extension that lets developers work more effectively with photos and videos stored in the device.

PhotoKit consists of the following two new frameworks:

- **Photos Framework:** This will allow developers to retrieve and edit photos and videos. It is responsible for handling changes made from external apps and also provides the tools to build a complete app, such as the Photos app that is available in iOS by default.
- **Photos UI:** This is responsible for providing the ability to create editing extensions—in other words, to edit photos with our custom app directly from the iOS Camera roll.

Photos Framework

Photos Framework provides tools to access, add, edit, and remove the model objects (images, videos, albums, and moments).

These model instances have a key characteristic – that is, they're read-only. So, for example, we can edit an image, and it won't modify the original content; however, it will create a new one.

The different model objects are as follows:

- **Assets:** They refer to images and videos, and are represented by the `PHAsset` class. It gives the ability to specify the type of media content (photo or video), the date of creation, location, and whether it's a favorite or not.
- **Asset collections:** They're also called moments, and they refer to ordered collections of assets, such as albums and smart albums. These objects are represented by `PHAssetCollection`, and its properties are title, type, and the start and end dates.
- **Collection lists:** They're an ordered collection of collections, and they usually represent a folder or a moment year. The class that manages them, `PHCollectionList`, stores the type, title, and the start and end dates of the list.

Photos Framework also introduces **Transient collections**, which reference a group of assets that are the result of a search or user selection, and can be used to interchange with common collections. One more thing to take into account when using this framework is that we will need to make use of class methods such as the following ones to work with assets:

```
[PHAsset fetchAssetsWithMediaType: options:];
[PHAssetCollection fetchMomentsWithOptions:];
[PHAssetCollection transientAssetCollectionWithAssets:title:];
```

Photos UI

Photos UI is a very interesting framework that allows us to create photo-editing extensions in our apps that will be available in the built-in Photos app.

This means that, when editing an image in the Camera roll, we can choose which app we want to use for this purpose; the result of this editing will be available for other apps and devices through iCloud and won't modify the original as it's read-only. To achieve this, we will need to create an app extension target that will provide a view controller that, in turn, will adopt the `PHContentEditingController` protocol. This is pretty simple, as the new version of Xcode provides a template to create photo-editing extensions; we just need to focus on the implementation of the following protocol methods:

```
[startContentEditingWithInput:]
[finishContentEditingWithCompletionHandler:]
[canHandleAdjustmentData:]
[cancelContentEditing]
```

We can use Photos Framework to work with the photo and video assets managed by the Photos app, including the **iCloud Photo Library**. Use this framework to retrieve assets for display and playback, edit their image or video content, or work with collections of assets, such as albums, moments, and iCloud shared albums.

Features of PhotoKit

The features of PhotoKit are as follows:

- **Fetching entities and requesting changes:** Instances of the Photos Framework model classes (`PHAsset`, `PHAssetCollection`, and `PHCollectionList`) represent the entities on which a user works within the Photos app. These entities are assets (images or videos), collections of assets (such as albums or moments), and lists of collections (such as album folders or moment clusters). These objects, also called photo entities, are read-only, immutable, and contain only metadata such as the asset's media type and its creation date.

We work with assets and collections by fetching the photo entities we're interested in and then using these objects to fetch the data we need to work with. To make changes to photo entities, we create change request objects and explicitly commit them to the shared `PHPhotoLibrary` object. This architecture makes it easy, safe, and efficient to work with the same assets from multiple threads or multiple apps and app extensions.

- **Change observing:** Use the shared `PHPhotoLibrary` object to register a change handler for the photo entities you fetch. PhotoKit tells your app whenever another app or device changes the content or metadata of an asset or the list of assets in a collection. The `PHChange` objects provide information on the object state before and after each change, with semantics that make it easy to update a collection view or a similar interface.
- **Support for Photos app features:** Use the `PHCollectionList` class to find assets corresponding to the moments hierarchy in the Photos app. Use the `PHAsset` class to identify burst photos, panoramic photos, and high-frame-rate videos. When the iCloud Photo Library is enabled, assets and collections in Photos Framework reflect the content available across all the devices on the same iCloud account.
- **Asset and thumbnail loading and caching:** Use the `PHImageManager` class to request images of assets in a specified size or AV Foundation objects to work with video assets. Photos Framework automatically downloads or generates images according to your specification, caching them for quick reuse. For faster performance with a large numbers of assets—for example, when populating a collection view with thumbnails—the `PHCachingImageManager` subclass adds bulk preloading.

- **Asset content editing:** The `PHAsset` and `PHAssetChangeRequest` classes define the methods to request photo or video content to edit and to commit your edits to the photo library. To support continuity of editing between different apps and extensions, Photos keeps the current and previous versions of each asset, along with the `PHAdjustmentData` object that describes the last edit. If your app supports adjustment data from a previous edit, you can allow the user to revert or alter the edit.

There are a lot of classes for PhotoKit; we will discuss them in this table:

Classes	Description
<code>PHAdjustmentData</code>	When a user edits an asset, Photos saves this object along with the modified image or video data
<code>PHAssetChangeRequest</code>	You can create and use this object within a photo library and change the block to create, delete, or modify <code>PHAsset</code> objects
<code>PHAssetCollectionChangeRequest</code>	You can create and use this object within a photo library and change the block to create, delete, or modify <code>PHAssetCollection</code> objects
<code>PHChange</code>	Photos provides this object to notify your app of any changes to the assets and collections managed by the Photos app
<code>PHCollectionListChangeRequest</code>	You can create and use this object within a photo library and change the block to create, delete, or modify <code>PHCollectionList</code> objects
<code>PHContentEditingInput</code>	This object describes an asset to be used for editing
<code>PHContentEditingInputRequestOptions</code>	You can use this object to specify options when requesting to edit the image or video content of a <code>PHAsset</code> object
<code>PHContentEditingOutput</code>	This object represents the results of editing the photo or video content of a Photos asset
<code>PHFetchOptions</code>	You can use this object to specify options when using class methods on the <code>PHAsset</code> , <code>PHCollection</code> , <code>PHAssetCollection</code> , and <code>PHCollectionList</code> classes to retrieve photo entities
<code>PHFetchResult</code>	This object is a container for an ordered list of photo entity objects

Classes	Description
<code>PHFetchResultChangeDetails</code>	This object provides detailed information about the differences between two fetch results—the one that you previously obtained and the updated one that would be the result if you performed the same fetch again
<code>PHImageManager</code>	This is a shared object, and it provides methods to load images or video data associated with a <code>PHAsset</code> object
<code>PHCachingImageManager</code>	This object fetches or generates image data for photo or video assets.
<code>PHImageRequestOptions</code>	You can use this object to specify options when requesting image representations of photo assets from a <code>PHImageManager</code> object
<code>PHObject</code>	This class is the abstract base class for photo-entity objects
<code>PHAsset</code>	This object represents an image or video file that appears in the Photos app, including the iCloud Photos content
<code>PHCollection</code>	This class is an abstract class that defines the behavior shared between the Photos collection classes
<code>PHAssetCollection</code>	This object represents a collection of photos or video assets
<code>PHCollectionList</code>	This object represents a group of asset collections
<code>PHObjectPlaceholder</code>	This object is a read-only proxy that represents an object yet to be created
<code>PHObjectChangeDetails</code>	This object provides detailed information about the differences between two states of a photo entity—one that you previously obtained and the updated state that would be the result if you fetched this entity again
<code>PHPhotoLibrary</code>	This is a shared object that represents the user's Photos library—the entire set of assets and collections managed by the Photos app, including the objects stored in the local device and (if enabled) in iCloud Photos
<code>PHVideoRequestOptions</code>	You can use this object to specify options when requesting video assets from a <code>PHImageManager</code> object

PhotoKit makes it easy to query model data through a variety of fetch methods. For example, to retrieve all images, you can call `PFAsset.Fetch`, passing the `PHAssetMediaType.Image` media type:

```
PHFetchResult fetchResults = PHAsset.FetchAssets  
    (PHAssetMediaType.Image, null);
```

The `PHFetchResult` instance will then contain all the `PFAsset` instances that represent images. To get the images themselves, you can use `PHImageManager` (or the caching version, `PHCachingImageManager`) to make a request for the image by calling `requestImageForAsset`. For example, the following code retrieves an image for each asset in `PHFetchResult` to display in a collection view cell. This example is in the Swift language, so we need to understand Swift. To begin with Swift in a better way, you can go to <http://www.raywenderlich.com/tutorials>.

```
public override UICollectionViewCell GetCell  
    (UICollectionView collectionView, NSIndexPath indexPath)  
{  
    var imageCell = (ImageCell)collectionView.DequeueReusableCell  
        (cellId, indexPath);  
    imageMgr.RequestImageForAsset ((PHAsset)fetchResults  
        [(uint)indexPath.Item],  
        thumbnailSize,  
        PHImageContentMode.AspectFill, new PHImageRequestOptions (),  
        (img, info) => {  
            imageCell.ImageView.Image = img;  
        });  
    return imageCell;  
}
```

This is how you handle querying and reading data. You can also write changes back to the library. Since multiple interested applications are able to interact with the system's photo library, you can register an observer to be notified of any changes using `PhotoLibraryObserver`. Then, when changes come in, your application can update accordingly. For example, here's a simple implementation for reloading the collection view:

```
class PhotoLibraryObserver : PHPhotoLibraryChangeObserver  
{  
    readonly PhotosViewController controller;  
    public PhotoLibraryObserver (PhotosViewController controller)  
    {  
        this.controller = controller;  
    }  
    public override void PhotoLibraryDidChange  
        (PHChange changeInstance)
```

```

    {
        DispatchQueue.MainQueue.DispatchAsync (() => {
            var changes = changeInstance.GetFetchResultChangeDetails
                (controller.fetchResults);
            controller.fetchResults = changes.FetchResultAfterChanges;
            controller.CollectionView.ReloadData ();
        });
    }
}

```

To actually write changes back from your application, you can create a change request. Each of the model classes has an associated change request class. For example, to change `PHAsset`, you can create `PHAssetChangeRequest`. The steps to perform changes that are written back to the photo library and sent to observers, like the preceding code, are as follows:

1. Perform the editing operation.
2. Save the filtered image data to a `PHContentEditingOutput` instance.
3. Make a change request to publish the changes from the editing output.

Here's an example that writes back a change to an image that applies a core image noir filter. You can also refer to <http://blog.xamarin.com/build-great-photo-experiences-in-ios-8-with-photokit/> to understand in more detail:

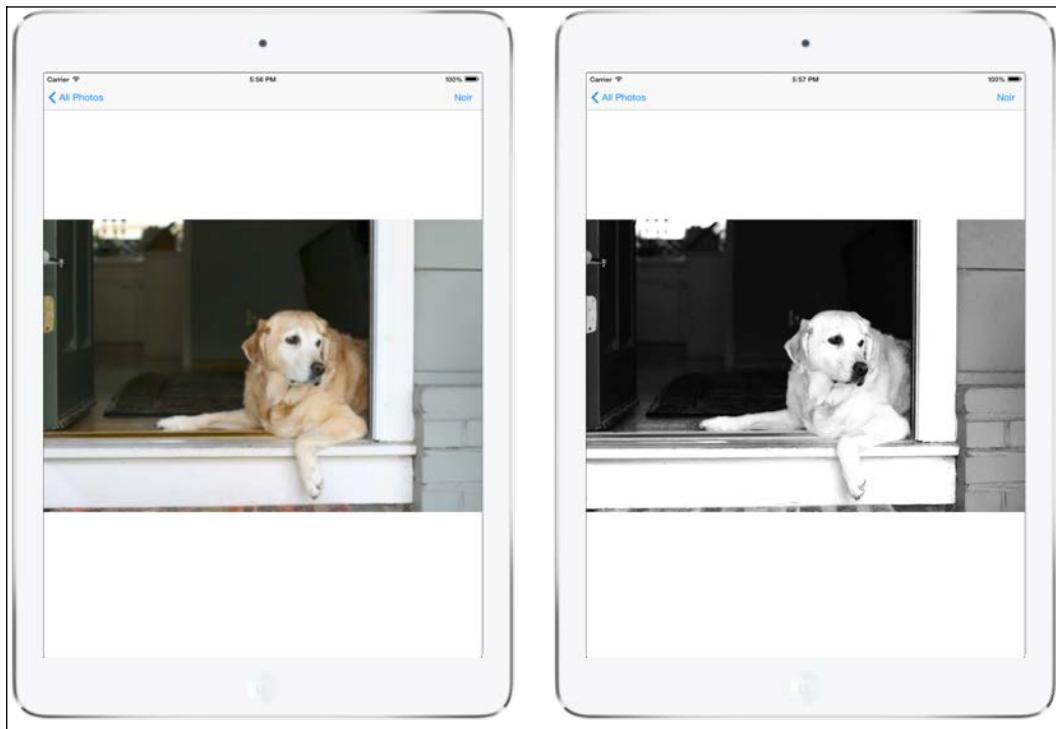
```

void ApplyNoirFilter (object sender, EventArgs e)
{
    Asset.RequestContentEditingInput (new
        PHContentEditingInputRequestOptions (), (input, options) => {
        //
        // perform the editing operation, which applies a noir filter
        // in this case
        var image = CIImage.FromUrl (input.FullSizeImageUrl);
        image = image.CreateWithOrientation
            ((CIImageOrientation)input.FullSizeImageOrientation);
        var noir = new CIPhotoEffectNoir {
            Image = image
        };
        var ciContext = CIContext.FromOptions (null);
        var output = noir.OutputImage;
        var uiImage = UIImage.FromImage (ciContext.CreateCGImage
            (output, output.Extent));
        imageView.Image = uiImage;
        //
        // save the filtered image data to a PHContentEditingOutput
        // instance
    });
}

```

```
var editingOutput = new PHContentEditingOutput(input);
var adjustmentData = new PHAdjustmentData();
var data = UIImage.AsJPEG();
NSError error;
data.Save(editingOutput.RenderedContentUrl, false, out error);
editingOutput.AdjustmentData = adjustmentData;
//
// make a change request to publish the changes from the
// editing output
PHPhotoLibrary.GetSharedPhotoLibrary.PerformChanges (
    () => {
        PHAssetChangeRequest request =
            PHAssetChangeRequest.ChangeRequest(Asset);
        request.ContentEditingOutput = editingOutput;
    },
    (ok, err) => Console.WriteLine ("photo updated successfully:
        {0}", ok));
});
}
```

When the user selects the button, the filter is applied, as shown in the following screenshot:



Thanks to `PHPhotoLibraryChangeObserver`, the change is reflected in the collection view when the user navigates back to the photo library.

You can download the sample project from <https://github.com/mikebluestein/PhotoKitDemo>.

Handoff for seamlessly resuming activities

Handoff is a feature in OS X and iOS that extends the user experience of continuity across devices. Handoff enables users to begin an activity on one device and then switch to another device and resume the same activity there. For example, a user who is browsing a long article in Safari moves to an iOS device that's signed in to the same Apple ID; the same webpage now automatically opens in Safari on iOS, with the same scroll position as on the original device. Handoff makes this experience as seamless as possible.

To participate in Handoff, an app adopts a small API in foundation. Each ongoing activity in an app is represented by a user activity object that contains the data needed to resume an activity on another device. When the user chooses to resume this activity, the object is sent to the resuming device. Each user activity object has a delegate object that is invoked to refresh the activity state at opportune times, such as just before the user activity object is sent between devices.

If continuing an activity requires more data than is easily transferred by the user activity object, the resuming app has the option to open a stream to the originating app. Document-based apps automatically support activity continuation for users working with iCloud-based documents. Apple apps use public APIs to implement Handoff for iOS 8 and OS X v10.10. A third-party developer can use the same APIs to implement Handoff in apps that share the developer's team ID. Such apps must either be distributed through the App Store or signed by the registered developer.

Compatibility with Handoff

First, let's see what is compatible with Handoff:

- iOS devices with a Lightning connector along with 2012 or later Mac models support Handoff. Both have radio chips that support both Bluetooth Low Energy and Wi-Fi Direct.
- If your device is compatible, you can enable Handoff in the **General** system preference on your Mac and in the **General** pane under the **Settings** app in iOS; in both cases, look for an option that includes the **Handoff** word.

- All the devices have to be signed in to the same iCloud account. Handoff doesn't work with other users (this is what AirDrop is for).
- Finally, make sure that both Bluetooth and Wi-Fi are turned on for all devices.

So far, Apple has announced that Handoff will work with the following apps:

- Mail
- Safari
- Pages
- Numbers
- Keynote
- Maps
- Messages
- Reminders
- Calendar
- Contacts

With them, we can start composing or reading an e-mail or website; editing a document, spreadsheet, or keynote; finding a location; typing a text; picking a reminder; entering an appointment; or looking up an address on your Mac and continuing or finishing it on your iPhone (iPad, or vice versa).

App framework support

UIKit and **AppKit** provide support for Handoff in the document, responder, and app delegate classes. Although there are minor behavioral differences between the platforms, the basic mechanism that enables the apps to save and restore user activities is the same, and the APIs are the same.

Handoff interactions

Handing off a user activity involves the following three steps:

1. Create a user activity object for each activity in which the user is engaged.
2. Update the user activity object regularly with information on what the user is doing.
3. Continue the user activity on a different device when the user requests it.

Implementing Handoff directly

Adopting Handoff in your app requires you to write the code that use APIs in UIKit and AppKit to create a user activity object, update the state of the object to track the activity, and continue the activity on another device.

Creating the user activity object

Every user activity that can be handed off to a continuing device is represented by a user activity object instantiated from the `NSUserActivity` class. This class creates a user activity object for each user activity it supports. The types of those user activities depend on the app. For example, Safari lets the user to continue with the browser on the same site.

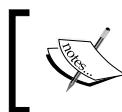
The following code creates the instance of `NSUserActivity`. The `myactivity.userInfo` object stores the current URL with the scroll position. The `becomeCurrent` object contains the current state of our project, and it is updating every second. Creating an object for the current state is necessary; otherwise, other devices won't understand from where to start.

```
NSUserActivity* myActivity = [[NSUserActivity alloc]
    initWithActivityType: @"com.myCompany.myBrowser.browsing"];
myActivity.userInfo = @{@" ... "};
myActivity.title = @"Browsing";
[myActivity becomeCurrent];
```

After terminating or finishing the app, the user activity object will release automatically. Then the object is removed from all the devices.

Specifying an activity type

The activity type identifier is a short string that appears in your app's `Info.plist` property list file in its `NSUserActivityTypes` array, which lists all the activity types that your app supports. The same string is passed when you create the activity, where the activity object is created with the activity type of `com.myCompany.myBrowser.browsing`, a reverse-DNS-style notation intended to avoid collisions. When the user chooses to continue the activity, the activity type (along with the app's team ID) determines which app to launch on the receiving device to continue the activity.



You can specify the activity type of an `NSUserActivity` object when you create the instance. You cannot change the activity type of the object after it is created.

For example, a Reminders-style app serializes the reminder list the user is looking at. When the user clicks on a new reminder list, the app tracks that activity in `NSUserActivityDelegate`. The following code shows a possible implementation of a method that gets called whenever the user switches to a different reminder list. This app appends an activity name to the app's bundle identifier to create the activity type to use when it creates its `NSUserActivity` object.

```
// UIResponder and NSResponder have a userActivity property
NSUserActivity *currentActivity = [self userActivity];

// Build an activity type using the app's bundle identifier
NSString *bundleName = [[NSBundle mainBundle]
    bundleIdentifier];
NSString *myActivityType = [bundleName
    stringByAppendingString:@".selected-list"];

if(![[currentActivity activityType]
    isEqualToString:myActivityType]) {
    [currentActivity invalidate];

    currentActivity = [[NSUserActivity alloc]
        initWithActivityType:myActivityType];
    [currentActivity setDelegate:self];
    [currentActivity setNeedsSave:YES];

    [self setUserActivity:currentActivity];
} else {

    // Already tracking user activity of this type
    [currentActivity setNeedsSave:YES];
}
```

The preceding code uses the `setNeedsSave:` accessor method to mark the user activity object when it needs to be updated. This enables the system to coalesce updates and perform them lazily.

Populating the activity object's user info dictionary

The activity object has a user info dictionary that contains whatever data is needed to hand off the activity to the continuing app. The user info dictionary can contain `NSArray`, `NSDate`, `NSDictionary`, `NSNull`, `NSNumber`, `NSSet`, `NSString`, and `NSURL` objects. The system modifies the `NSURL` objects that use the `file:` scheme and point at iCloud documents to point to the same items in the corresponding container on the receiving device:

```
NSUserActivity* myActivity = [[NSUserActivity alloc]
    initWithActivityType: @"com.myCompany.myReader.
reading"] ;

// Initialize userInfo
NSURL* webpageURL = [NSURL URLWithString:@"http://www.myCompany.com"] ;
myActivity.userInfo = @{
    @"docName" : currentDoc,
    @"pageNumber" : self.pageNumber,
    @"scrollPosition" : self.scrollPosition
} ;
```

Adopting Handoff in responders

You can associate responder objects (inheriting from `NSResponder` on OS X or `UIResponder` on iOS) with a given user activity if you set the activity as the responder's `userActivity` property. The system automatically saves the `NSUserActivity` object at appropriate times, calling the responder's `updateUserActivityState:` override to add current data to the user activity object using the activity object's `userInfoEntriesFromDictionary:` method:

```
- (void)updateUserActivityState:(NSUserActivity *)userActivity {
    ...
    [userActivity setTitle: self.activityTitle];
    [userActivity addUserInfoEntriesFromDictionary:
        self.activityUserInfo];
}
```

Continuing an activity

Handoff automatically advertises user activities that are available to be continued on iOS and OS X devices that are in physical proximity to the originating device and signed in to the same iCloud account as the originating device. When the user chooses to continue a given activity, Handoff launches the appropriate app and sends the app delegate messages that determine how the activity is resumed using the `AppDelegate`.

Implement the `application:willContinueUserActivityWithType:` method to let the user know that the activity will continue shortly. Use the `application:continueUserActivity:restorationHandler:` method to configure the app to continue the activity. The system calls this method when the activity object, including the activity state data in its `userInfo` dictionary, is available to the continuing app.

 For URLs transferred in the `userInfo` dictionary of an `NSUserActivity` object, we must call `startAccessingSecurityScopedResource`, and it must return YES before we can access the URL. Call `stopAccessingSecurityScopedResource` when you are done using the file.

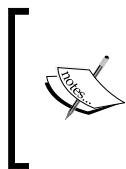
Exceptions to this requirement are URLs of `UIDocument` documents and those of `NSDocument` that are automatically created for specifying `NSUbiquitousDocument` `UserActivityType` apps and return NO from the `:continueUserActivity:restorationHandler: application` (or leave it unimplemented). See *Adopting Handoff in Document-Based Apps* at https://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/Handoff/AdoptingHandoff/AdoptingHandoff.html#/apple_ref/doc/uid/TP40014338-CH2-SW17.

Additional configuration of your app to continue the activity can optionally be performed by objects you give to the restoration handler block that is passed in with the `application:continueUserActivity:restorationHandler:` message. The following code shows a simple implementation of this method:

```
- (BOOL) application:(NSApplication *)application
              continueUserActivity:(NSUserActivity *)userActivity
                restorationHandler:(void (^)(NSArray *))restorationHandler {
    BOOL handled = NO;
    // Extract the payload
    NSString *type = [userActivity activityType];
    NSString *title = [userActivity title];
    NSDictionary *userInfo = [userActivity userInfo];
    // Assume the app delegate has a text field to display the
    // activity information
    [appDelegateTextField setStringValue: [NSString
        stringWithFormat:
            @"User activity is of type %@, has title %@, and user info
            %@",
            type, title, userInfo]];
    restorationHandler(self.windowControllers);
    handled = YES;
    return handled;
}
```

In this case, the app delegate has an array of `NSWindowController` objects, `windowControllers`. These window controllers know how to configure all of the app's windows to resume the activity. After you pass this array to the `restorationHandler` block, Handoff sends each of those objects a `restoreUserActivityState:` message, passing in the resuming activity's `NSUserActivity` object. The window controllers inherit the `restoreUserActivityState:` method from `NSResponder`; each controller object overrides this method to configure its window, using the information in the activity object's `userInfo` dictionary.

To support graceful failure, the app delegate should implement the `application:didFailToContinueUserActivityWithType:error:` method. If you don't implement this method, the app framework, nonetheless, displays diagnostic information contained in the passed-in `NSError` object.



The `UIApplicationDelegate` methods for Handoff described in this section are not called when either of the application delegate methods, `application:willFinishLaunchingWithOptions:` or `application:didFinishLaunchingWithOptions:`, returns NO.

Native App-to-Web Browser Handoff

When using a native app on the originating device, the user might want to continue the activity on a corresponding native app. If there is a web page that corresponds to the activity, it can still be handed off. For example, video library apps enable users to browse movies available for viewing, and mail apps enable users to read and compose e-mails. In many cases, users can perform the same activity through a web page interface. In this case, the native app knows the URL for the web interface, possibly including syntax that designates a particular video being browsed or message being read. So, when the native app creates the `NSUserActivity` object, it sets the `webpageURL` property. If the receiving device doesn't have an app that supports the user activity's `activityType` property, it can resume the activity in the default web browser of the continuing platform.

A web browser on OS X that wants to continue an activity in this way should claim the `NSUserActivityTypeBrowsingWeb` activity type (by entering this string in its `NSUserActivityTypes` array in the app's `Info.plist` property list file). This ensures that, if the user selects any other browser as their default browser, it receives the activity object instead of Safari.

Web Browser-to-Native App Handoff

In the opposite case, if the user is using a web browser on the originating device and the receiving device is an iOS device with a native app that claims the domain portion of the `webpageURL` property, then iOS launches the native app and sends it an `NSUserActivity` object with an `activityType` value of `NSUserActivityTypeBrowsingWeb`. The `webpageURL` property contains the URL that the user was visiting, while the `userInfo` dictionary is empty.

The native app on the receiving device must adapt to this behavior by claiming a domain in the `com.apple.developer.associated-domains` entitlement. The value of this entitlement has the `<service>:<fully qualified domain name>` format, for example, `activitycontinuation:example.com`. In this case, the service must be `activitycontinuation`. Add the value for the `com.apple.developer.associated-domains` entitlement in Xcode in the **Associated Domains** section under the **Capabilities** tab of the **Target** settings.

If this domain matches the `webpageURL` property, Handoff downloads a list of approved app IDs from the domain. Domain-approved apps are authorized to continue the activity. On your website, you can list the approved apps in a signed JSON file named `apple-app-site-association`; for example, the web address becomes `https://example.com/apple-app-site-association` (you must use an actual device rather than the simulator to test download the JSON file).

The JSON file contains a dictionary that specifies a list of app identifiers in the `<team identifier>. <bundle identifier>` format in the **General** tab of the **Target** settings—for example, `YWBN8XTPBJ.com.example.myApp`. The following code shows an example of such a JSON file formatted for reading:

```
{  
    "activitycontinuation": {  
        "apps": [ "YWBN8XTPBJ.com.example.myApp",  
                 "YWBN8XTPBJ.com.example.myOtherApp" ]  
    }  
}
```

To sign the JSON file (so that it is returned from the server with the correct content-type of `application/pkcs7-mime`), put the content in a text file and sign it. You can perform this task with terminal commands such as those shown in the following code, by removing the whitespace from the text for ease of manipulation. Use the `openssl` command with the certificate and key for an identity issued by a certificate authority trusted by iOS (which is listed at <http://support.apple.com/kb/ht5012>). It need not be the same identity that hosts the web credentials (`https://example.com` in the example code), but it must be a valid TLS certificate for the domain name in question:

```
echo '{"activitycontinuation":{"apps":["YWBN8XTPBJ.com.example.myApp",  
"YWBN8XTPBJ.com.example.myOtherApp"]}}' > json.txt
```

```
cat json.txt | openssl smime -sign -inkey example.com.key  
    -signer example.com.pem  
    -certfile intermediate.pem  
    -noattr -nodetach  
    -outform DER > apple-app-site-  
        association
```

The output of the `openssl` command is the signed JSON file that you put on your website at the `apple-app-site-association` URL—in this example, `https://example.com/apple-app-site-association`.

An app can set the `webpageURL` property to any web URL, but it only receives activity objects whose `webpageURL` domain is in the `com.apple.developer.associated-domains` entitlement. Also, the scheme of the `webpageURL` must be `http` or `https`. Any other scheme throws an exception.

Using continuation streams

If resuming an activity requires more data than can be efficiently transferred by the initial Handoff payload, a continuing app can call back to the originating app's activity object to open streams between the apps and transfer more data. In this case, the originating app sets its `NSUserActivity` object's Boolean property, `supportsContinuationStreams`, to YES, sets the user activity delegate, and then calls `becomeCurrent`, as shown in the following code:

```
NSUserActivity* activity = [[NSUserActivity alloc] init];  
activity.title = @"Editing Mail";  
activity.supportsContinuationStreams = YES;  
activity.delegate = self;  
[activity becomeCurrent];
```

On the continuing device, after users indicate that they want to resume the activity, the system launches the appropriate app and begins sending messages to the app delegate. The app delegate can then request streams back to the originating app by sending the `getContinuationStreamsWithCompletionHandler` message to its user activity object, as shown in the override implementation in the following code:

```
- (BOOL)application:(UIApplication *)application  
continueUserActivity: (NSUserActivity *)userActivity  
restorationHandler: (void(^)(NSArray  
*restorableObjects))restorationHandler  
{  
    [userActivity getContinuationStreamsWithCompletionHandler:^(  
        NSInputStream *inputStream,
```

```
NSOutputStream *outputStream, NSError *error) {  
  
    // Do something with the streams  
  
}];  
  
return YES;  
}
```

On the originating device, the user activity delegate receives the streams in a callback to its `userActivity:didReceiveInputStream:outputStream` method, which it implements to provide the data needed to continue the user activity on the resuming device using the streams.

`NSInputStream` provides read-only access to stream data, and `NSOutputStream` provides write-only access. Therefore, data written to the output stream on the originating side is read from the input stream on the continuing side; and vice versa. Streams are meant to be used in a request-and-response fashion, that is, the continuing side uses the streams to request more continuation data from the originating side, which then uses the streams to provide the requested data.

Continuation streams are an optional feature of Handoff; most user activities do not need them for successful continuation. Even when streams are needed, in most cases there should be minimal back-and-forth between the apps. A simple request from the continuing app accompanied by a response from the originating app should be enough for most continuation events. You can download the sample project from <https://github.com/dokterdok/Continuity-Activation-Tool>.

Summary

In this chapter, we discussed the new iOS 8 APIs and little code snippets of Swift. You learned about the PhotoKit framework and Handoff with some code snippets. In the next chapter, we will discuss iCloud and security services in iOS, together with their implementations.

6

Using iCloud and Security Services

In this chapter, you will learn how to store your data on cloud, that is, on iCloud and security services for iOS, through which we can secure our data, passwords, and so on.

In this chapter, we will cover the following topics:

- Working with iCloud
- Saving data using the keychain process
- iOS Touch ID authentication

Working with iCloud

Basically, **iCloud** is a service that helps users synchronize their data across devices. The main purpose is to let users easily store their data, whether it's a file or document, so that they can access it on any of their iOS devices. While you can use other cloud services to save files or data, the core idea behind iCloud is to eliminate explicit or wired connection between devices. Apple does not want users to think of the cloud servers and the syncing. Everything simply works seamlessly.

The same design philosophy also applies to developers. When we adopt iCloud, we do not need to know how to interact with the cloud server or upload data to iCloud. The iOS handles all the heavy lifting. Our focus is on the content such as managing the change of data or developing a connection between a cloud and a device.

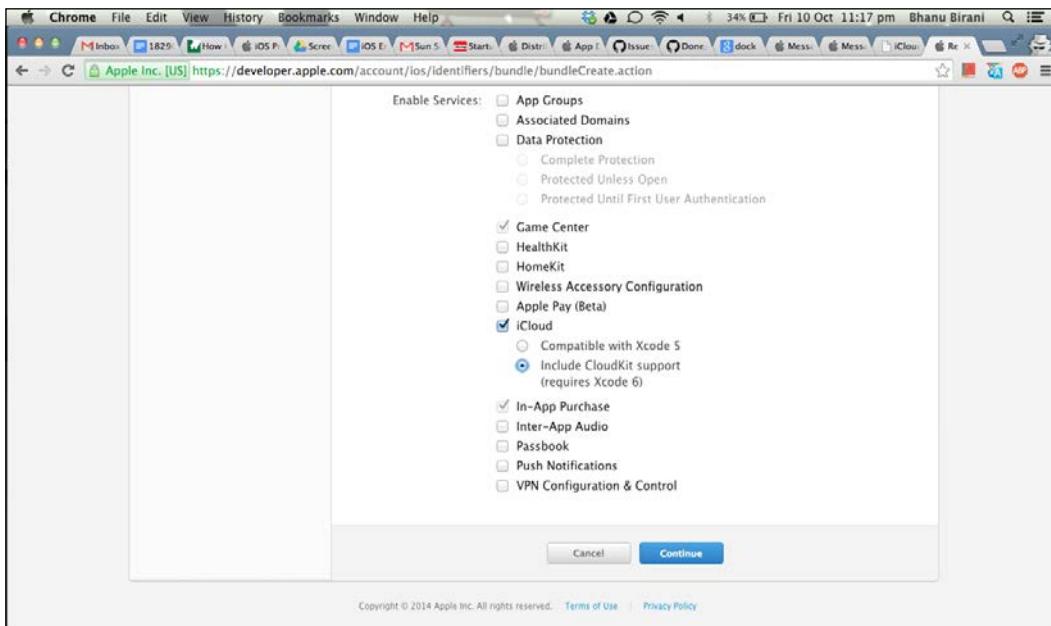
iCloud offers three kinds of storage:

- **Key-value storage:** This is used to store content such as settings, preferences, and app states.

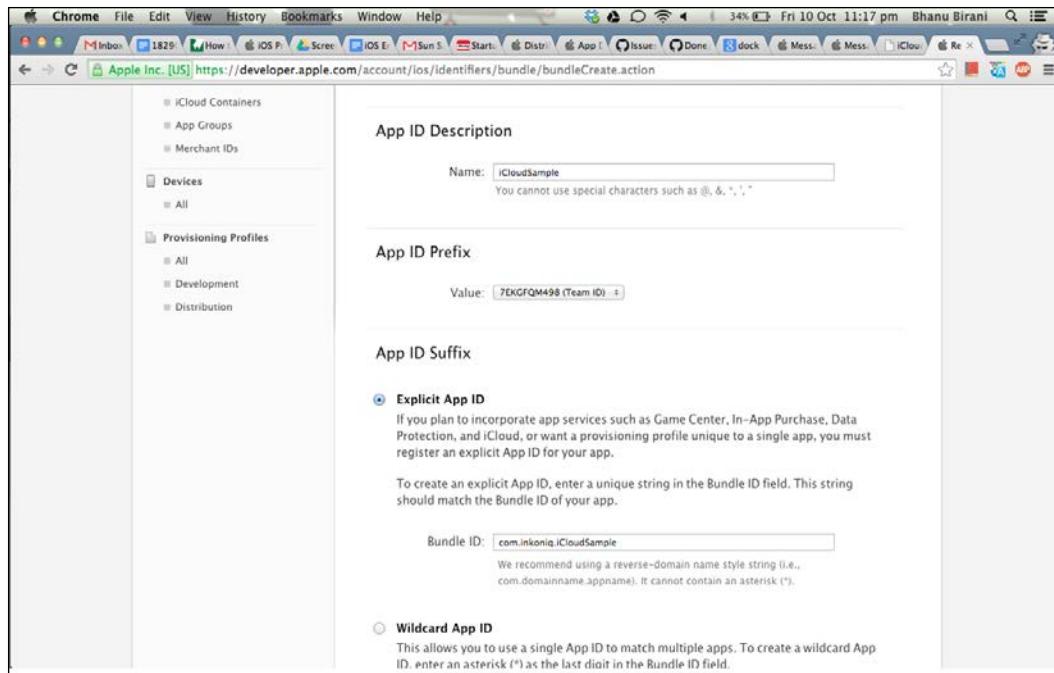
- **Document storage:** This is used to store file type content such as WordPress documents, drawings, and complex app states.
- **Core data storage:** This is used for multi-device database solutions for structured content. iCloud core data storage is built on document storage and employs the same iCloud APIs.

Let's understand how iCloud actually works. To use iCloud, we need an iOS developer account. Assuming that we have an iOS developer account, proceed with the following steps:

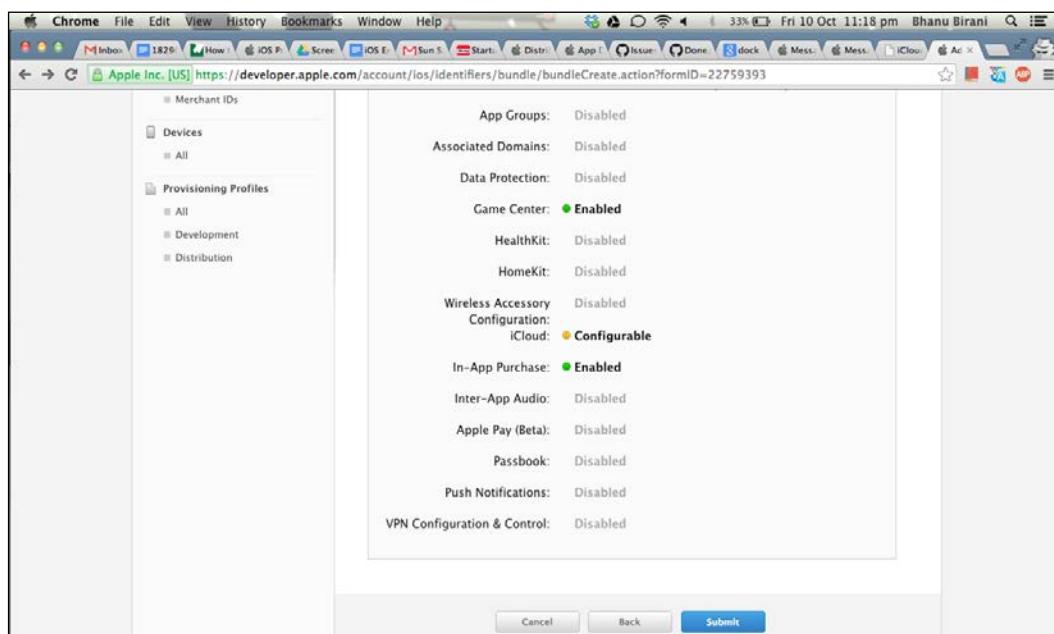
1. Go to <https://idmsa.apple.com/IDMSWebAuth/login?&appIdKey=891bd3417a7776362562d2197f89480a8547b108fd934911bcbea0110d07f757&pah=%2F%2Faccount%2Findex.action>. We will first create the app ID with the iCloud feature available.
2. Sign in to the iOS Provisioning Portal, select the app IDs, and then create a new app ID.
3. Enable the iCloud service for your app by selecting the **iCloud** option. Select the Xcode that you are using (if you are using Xcode 5, then select Xcode 5) and click on **Continue**, as shown in the following screenshot:



4. Now, give the name of our project in the **Name** text field, write the bundle ID in the **Bundle ID** text field, and click on **Continue**, as shown in the following screenshot:

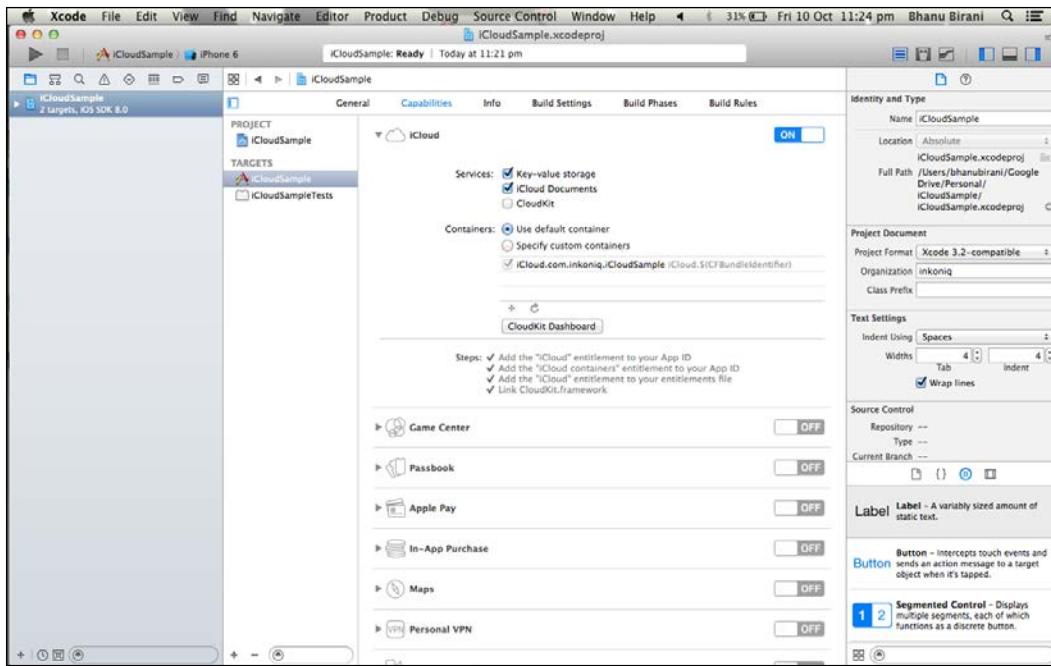


- Now, it will show the enabled services for our app. Make sure **iCloud** is enabled or configurable, as shown in the following screenshot:

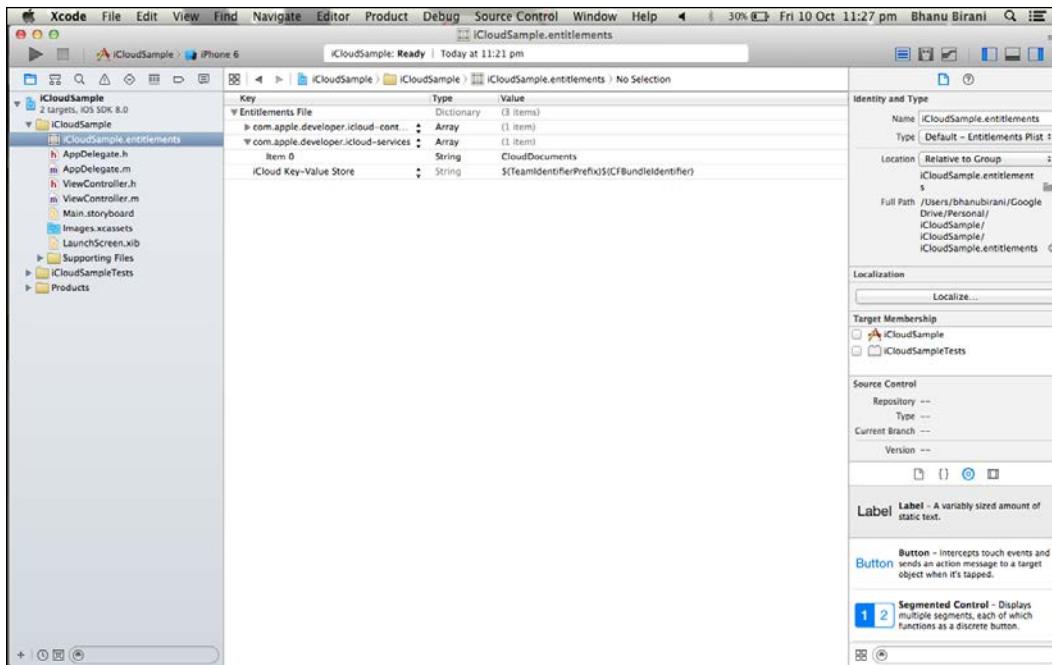


Using iCloud and Security Services

6. Then, go to **Xcode** and build a new single-view application. After that, go to **Capabilities** in **Xcode** and turn on **iCloud**; this will include iCloud entitlements, iCloud containers, and the link to the Cloud framework, as shown in the following screenshot:

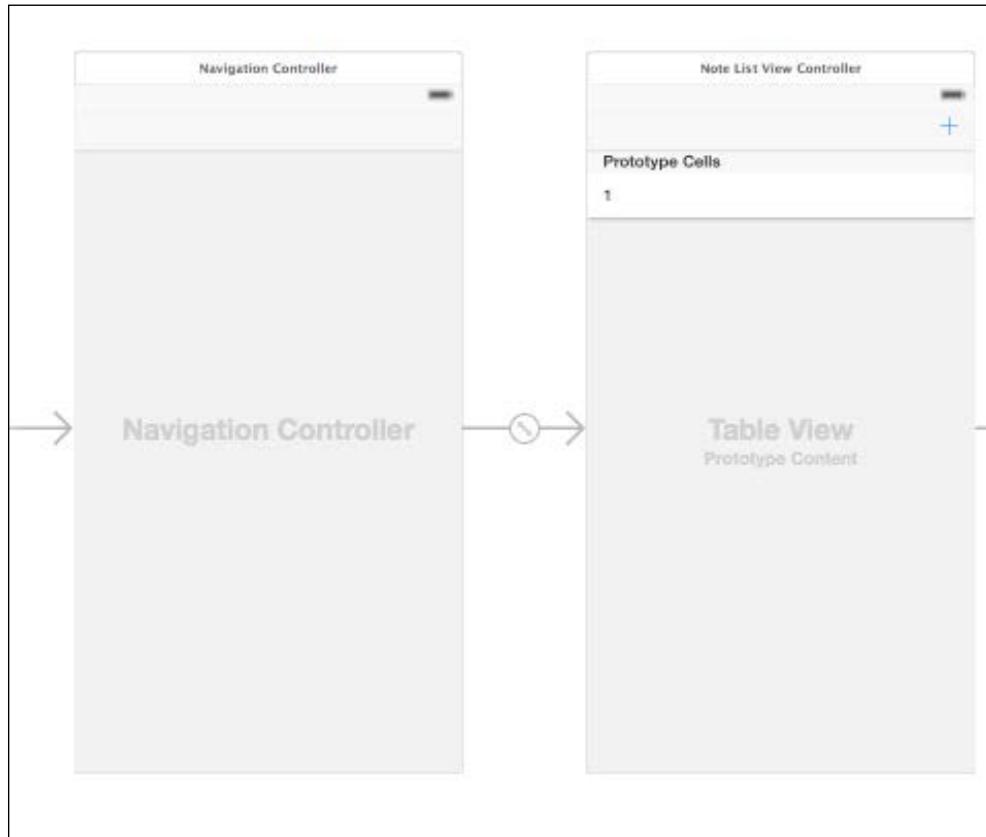


7. Make sure that the entitlements are created automatically. The following screenshot shows the entitlements created:



8. Delete the `ViewController` class and existing view in the storyboard.
9. Now, we will empty the storyboard and create a new objective-C class, a subclass of `tableViewController`, and name it (for example, `NoteListViewController`).
10. Drag one table view controller to the storyboard and embed it in Navigation Controller. Now, give a name to this table view controller from the Attribute Inspector; it should be the same as the class name. Do the same thing one more time for a new class and name it (for example, `AddNoteViewController`).

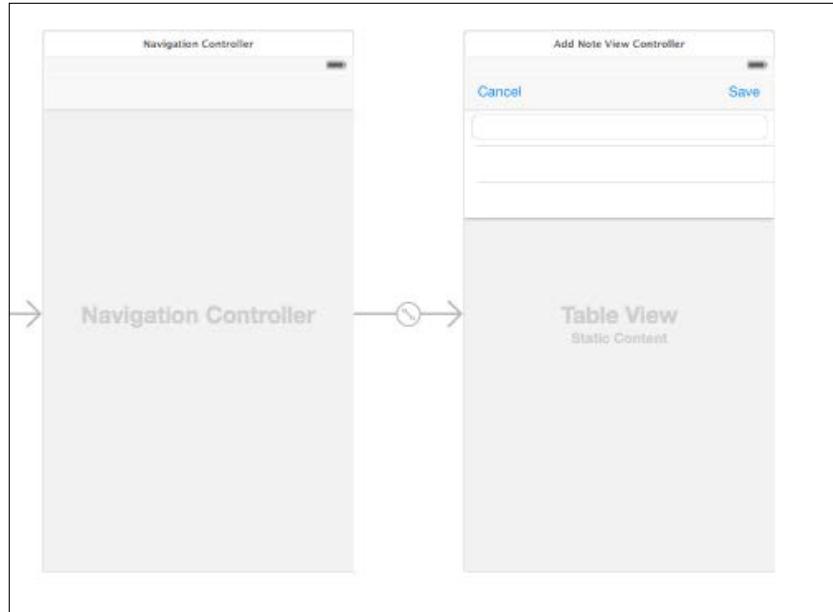
11. Now, we have two table views in the storyboard. For `NoteListViewController`, drag a button to the top-right part of the navigation bar and set the identifier as `add`. This will automatically change the button to a `+` button, as shown in the following screenshot. Next, select the prototype cell and change its style to **Basic**.



Navigation Controller and Note List View Controller

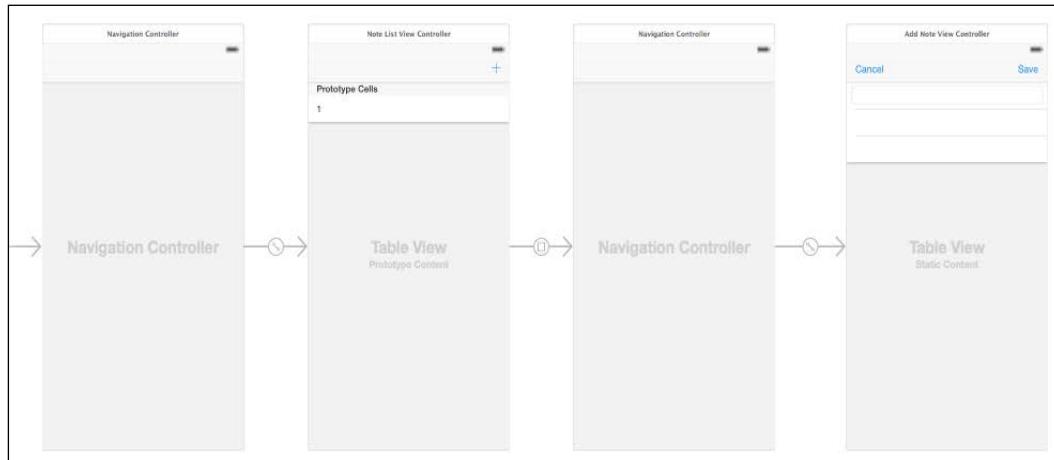
12. For `AddNoteViewController`, drag the bar buttons into the navigation bar. Name one as `Cancel` and the other as `Save`.

13. Then, select the table view and set the content to **Static Cells**. Finally, add one static row to the table view and add a text field, as shown here:



Navigation Controller and Note List View Controller

14. AddNoteViewController will be shown when the user taps on the + button. So, press and hold the *Ctrl* key, click on the + button, and drag it to **Add Note View Controller**. Select **Modal** as the Segue action. Our storyboard should now look like this:



The table view showing Add Note View Controller

15. Finally, link up the text field to AddNoteViewController and create two action methods: Cancel and Save. Our code in AddNoteViewController.h should like this:

```
#import <UIKit/UIKit.h>
@interface AddNoteViewController : UITableViewController
- (IBAction)cancel:(id)sender;
- (IBAction)save:(id)sender;
@property (weak, nonatomic) IBOutlet UITextField
    *noteTextField;
```

16. We need to implement the NoteListViewController class so that we can display the notes available on the cloud in the table view. Open NoteListViewController.h and add this property to it:

```
@property (strong, nonatomic) NSMutableArray *notes;
```

17. The purpose of the property is to save notes locally. We'll display the notes in the table view when needed. Next, we need to make a lazy instantiation of this property. Lazy instantiation is a good technique if we have an object that only needs to be configured once and has some configuration involved that we don't want to clutter in our init method. Add the following code in NoteListViewController.m:

```
- (NSArray *)notes
{
    if(!_notes){
        _notes = [NSMutableArray array];
        _notes = [[[NSUbiquitousKeyValueStore defaultStore]
            objectForKey:@"AVAILABLE_NOTES"] mutableCopy];
    }
    return _notes;
}
```

18. In viewDidLoad, we need to create the notification method as follows:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem =
        self.editButtonItem;
    // Observer to catch changes from iCloud
    NSUbiquitousKeyValueStore *store =
    [[NSUbiquitousKeyValueStore defaultStore] addObserver:self
        selector:@selector(storeDidChange:)
        name: NSUbiquitousKeyValueStore
        DidChangeExternallyNotification
        object:store];
    [[NSUbiquitousKeyValueStore defaultStore] synchronize];
    // Observer to catch the local changes
```

```
[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(didAddNewNote:)
    name:@"New Note"
    object:nil];
}
```

iCloud data synchronization is achieved using the `NSUbiquitousKeyValueStore` class. `NSUbiquitousKeyValueStore` is a subclass of `NSObject`, and it is available in iOS 5.0 and later. An `NSNotificationCenter` object provides a mechanism to broadcast information or messages within a program. It is also a subclass of `NSObject`, and it is available in iOS 2.0 and later.

Our `viewDidLoad` method will probably look like this:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    // Observer to catch changes from iCloud
    NSUbiquitousKeyValueStore *store = [NSUbiquitousKeyValueStore defaultStore];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(storeDidChange:)
        name:
        NSUbiquitousKeyValueStoreDidChangeExternallyNotification
        object:store];

    [[NSUbiquitousKeyValueStore defaultStore] synchronize];

    // Observer to catch the local changes
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(didAddNewNote:)
        name:@"New Note"
        object:nil];
}
```

19. Now, we have to implement methods that are executed when the preceding notifications are called. The `didAddNewNote` method will be invoked when users save a new note:

```
#pragma mark - Observer New Note
- (void) didAddNewNote: (NSNotification *) notification{
    NSDictionary *userInfo = [notification userInfo];
    NSString *noteStr = [userInfo valueForKey:@"Note"];
    [self.notes addObject:noteStr];
    // Update data on the iCloud
    [[NSUbiquitousKeyValueStore defaultStore]
        setArray:self.notes forKey:@"AVAILABLE_NOTES"];
    // Reload the table view to show changes
    [self.tableView reloadData];
}
```

```
#pragma mark - Observer

- (void)storeDidChange:(NSNotification *)notification{
    // Retrieve the changes from iCloud
    _notes = [[[NSUbiquitousKeyValueStore defaultStore]
        arrayForKey:@"AVAILABLE_NOTES"] mutableCopy];
    // Reload the table view to show changes
    [self.tableView reloadData];
}
```

20. We need to display the notes in table view. We have already retrieved the notes saved in iCloud; the rest of the implementation is to display the notes in the table view. In `NoteListViewController.m`, add the following code:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSectionInSection:(NSInteger)section
{
    return [_notes count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier
        forIndexPath:indexPath];
    NSString *note = [_notes
        objectAtIndex:indexPath.row];
    [cell.textLabel setText:note];
    return cell;
}
```

21. We have now come to the last part of the process: implementing `AddNoteViewController` to add notes to the cloud. Go to the implementation file of `AddNoteViewController` and add the following code:

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

```
// Notify the previous view to save the changes
locally
[[NSNotificationCenter defaultCenter]
postNotificationName:@"New Note" object:self
userInfo:[NSDictionary dictionaryWithObject:
self.noteTextField.text forKey:@"Note"]];
[self dismissViewControllerAnimated:YES
completion:nil];
}
```

22. To test the note app, we have to compile and deploy it onto an actual device. If you are using a simulator that supports iOS 7.1 or later, then you will be able to use the simulator for testing. Make sure that you enable iCloud on both devices. Launch the app, add a note on one device, and you'll see the note appear on the other device.

Keychain Services

Keychain Services is a programming interface that enables developers to add, find, modify, and remove keychain items. In iOS and OS X, a keychain is an encrypted container that stores passwords and other private data that need to be secured. In iOS, each application has its own keychain to which it has access. This ensures that our data is secured by the third party and other users.

Keychain provides a small space on which we can only store specific data such as passwords, account numbers, private numbers, and so on. With this article, I hope to convince you of the value of using the keychain in iOS and OS X instead of, for example, the application's user-defaults database, which stores its data in plain text without any form of security. Saving our data in keychain is better than the default database, because keychain is far more secure and robust.

In iOS, an application can use the keychain through the Keychain Services API. This API provides a number of functionalities to manipulate the data stored in the application's keychain. The APIs are as follows:

- `SecItemAdd`: This API is used to add data in keychain
- `SecItemCopyMatching`: This API is used to find the existing data in keychain
- `SecItemDelete`: This API is used to remove the data from the application's keychain
- `SecItemUpdate`: This API is used to update the data in the application's keychain

The Keychain Services API is a C-based API, but this doesn't prevent us from using it. Each of the preceding functions accepts a dictionary (`CFDictionaryRef`).

Encryption and decryption

Most of us know about two types of encryption: symmetric and asymmetric encryption. Symmetric encryption, on the one hand, uses one shared key to encrypt and decrypt data. Asymmetric encryption, on the other hand, uses one key to encrypt data and another separate, but related, key to decrypt data.

In iOS, to encrypt and decrypt data, a Security framework is available. This process takes place under the hood, so we won't be directly interacting with this framework. We'll use symmetric encryption in our example application.

The Security framework offers a number of other services, such as Randomize services to generate cryptographically-secure random numbers; Certificate, Key and Trust services to manage certificates; public and private keys; and trust policies. The Security framework is a low-level framework available in both iOS and OS X with C-based APIs.

iOS keychain concepts and structure

The keychain is a secure and encrypted way to store our precious data. It is important that your app, and all subsequent versions of it, are signed by the same mobile-provision profile. If they aren't, you will have many troubles later on.

A keychain is a unit of sensitive data stored in your app. Keychain items are accompanied by one or more attributes. The attributes describe the keychain item, and which attributes we can use depends on the item class of the keychain item. The item class refers to the type of data we are going to store. This can be a username/password combination, a certificate, a generic password, and so on.

Understanding the application flow

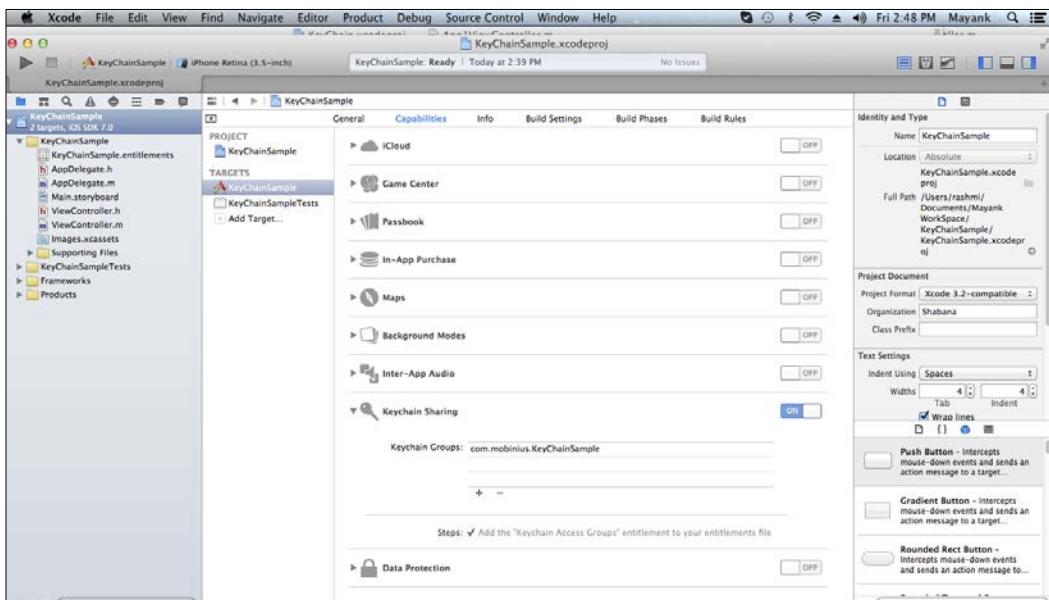
Before we start building the application, we need to know about the application flow, which is explained here:

1. When the user launches the application, it presents the user with a view to sign in.
2. If it hasn't created an account yet, its credentials are added to the keychain and signed in. If it has an account but enters an incorrect password, an error message is shown.

3. Once it has signed in, it has access to the data collected with the application.
The data is securely stored by the application.
4. Whenever it takes data with the text field, this data is encrypted and stored in the application's Documents directory.
5. Whenever it switches to another application or the device gets locked, it automatically signs out.

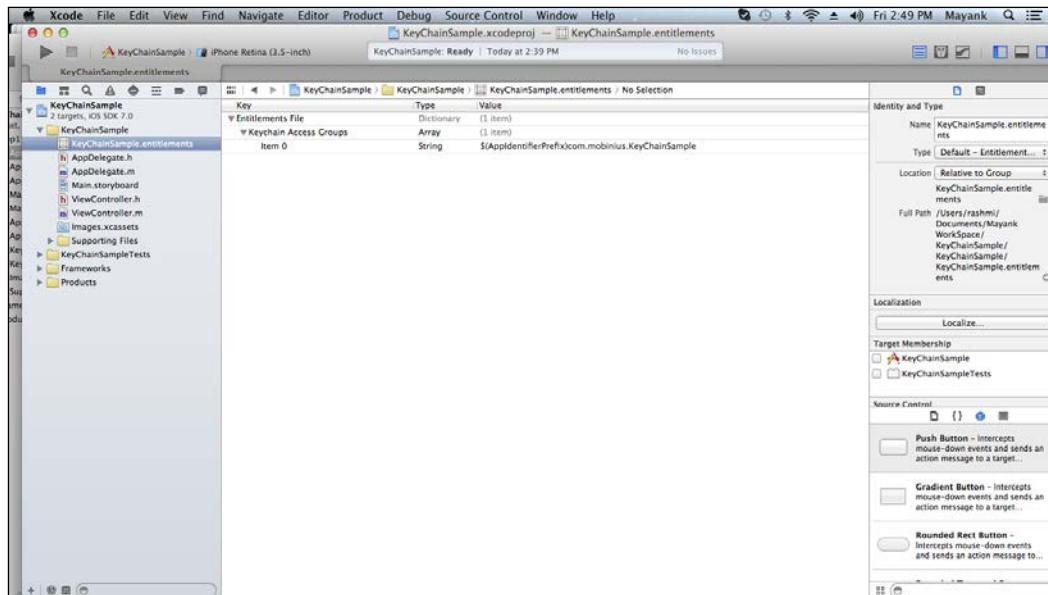
Let's start an activity on the Keychain. Just follow these steps to accomplish the activity:

1. Open Xcode and make a single-view application. Then, go to **KeyChainSample | Capabilities** and turn on the **Keychain Sharing** option, as shown in the following screenshot:

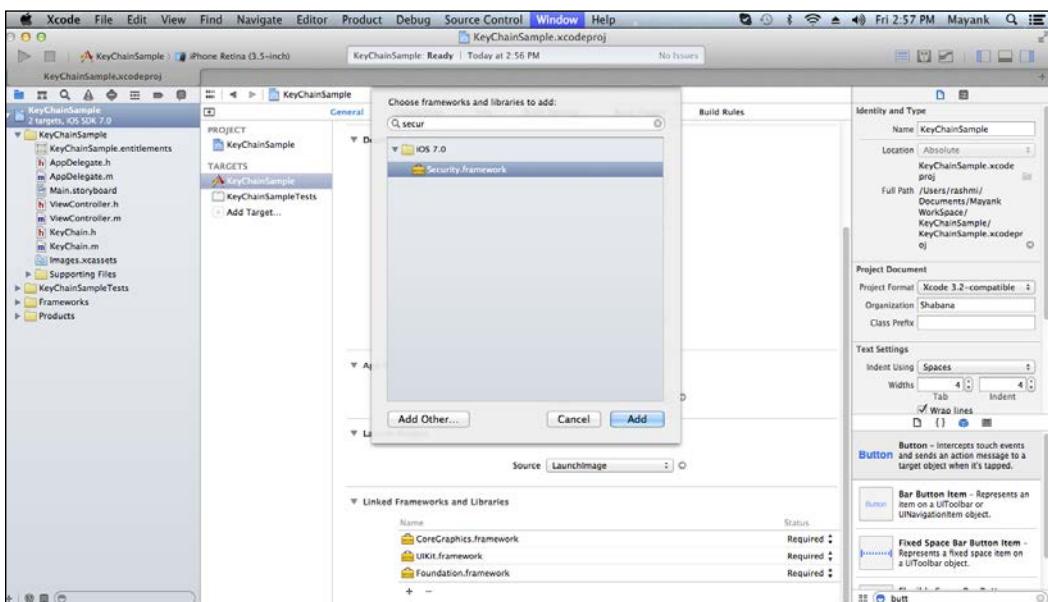


Using iCloud and Security Services

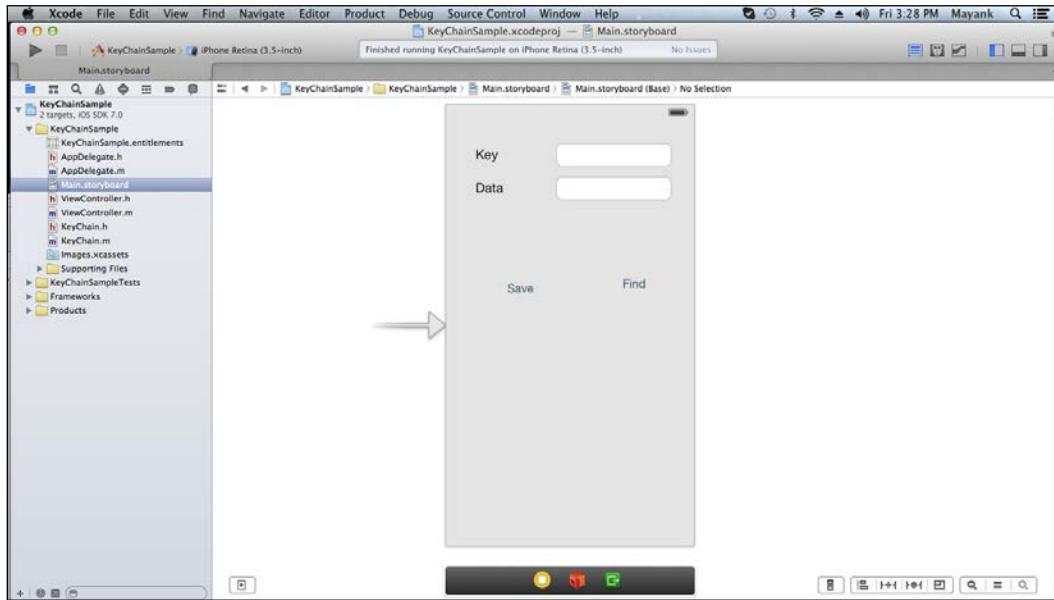
2. After turning on the **Keychain Sharing** option, it will automatically create **Entitlements File** for us, as shown in the following screenshot:



3. Don't forget to add **Security framework** (shown in the following screenshot); without it, the keychain will not work:



4. Now, make a user interface in the storyboard like this:



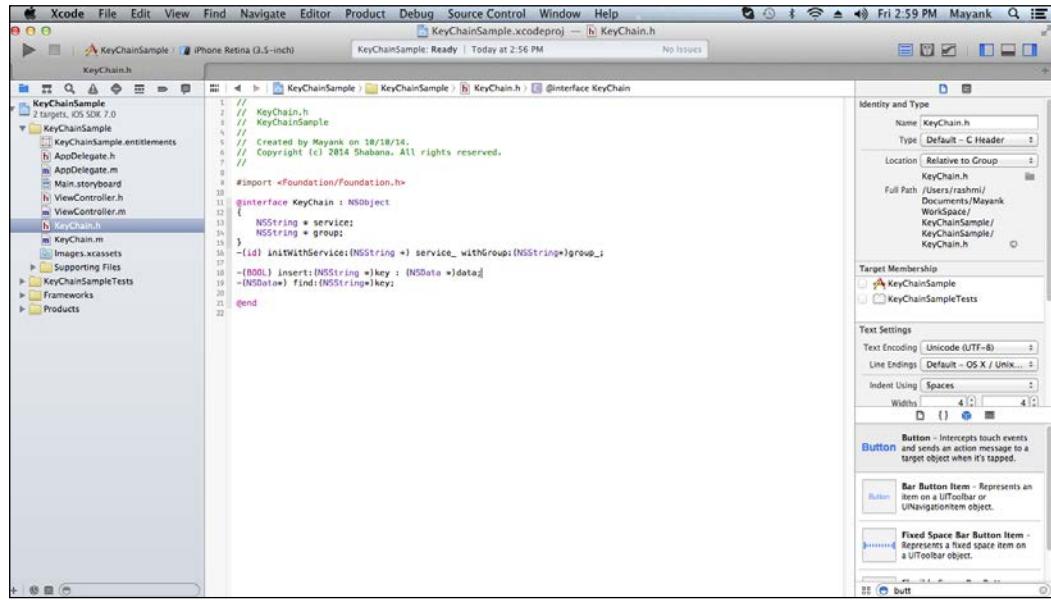
5. Add a new Objective-C file to the project subclass of `NSObject`. Then, write the following code in the interface file of the new file. Also, import `security.h` on the top of the file:

```
#import <security.h>
@interface KeyChain : NSObject
{
    NSString * service;
    NSString * group;
}
-(id) initWithService:(NSString *) service_
    withGroup:(NSString*)group_;

-(BOOL) insert:(NSString *)key : (NSData *)data;
-(NSData*) find:(NSString*)key;
```

Using iCloud and Security Services

Our interface file will look like this:



6. Write the following code in the implementation file of the new file that we added:

```
@implementation KeyChain
- (id) initWithService:(NSString *) service_
    withGroup:(NSString*)group_
{
    self = [super init];
    if(self)
    {
        service = [NSString stringWithString:service_];
        if(group_)
            group = [NSString stringWithString:group_];
    }
    return self;
}
- (NSMutableDictionary*) prepareDict:(NSString *) key
{
    NSMutableDictionary *dict =
        [[NSMutableDictionary alloc] init];
    [dict setObject:(__bridge id)kSecClassGenericPassword
        forKey:(__bridge id)kSecClass];
    NSData *encodedKey = [key
        dataUsingEncoding:NSUTF8StringEncoding];
```

```

[dict setObject:encodedKey
    forKey:(__bridge id)kSecAttrGeneric];
[dict setObject:encodedKey
    forKey:(__bridge id)kSecAttrAccount];
[dict setObject:service
    forKey:(__bridge id)kSecAttrService];
[dict setObject:(__bridge id)
    kSecAttrAccessibleAlwaysThisDeviceOnly
    forKey:(__bridge id)kSecAttrAccessible];
if(group != nil)
    [dict setObject:group
        forKey:(__bridge id)kSecAttrAccessGroup];
return dict;

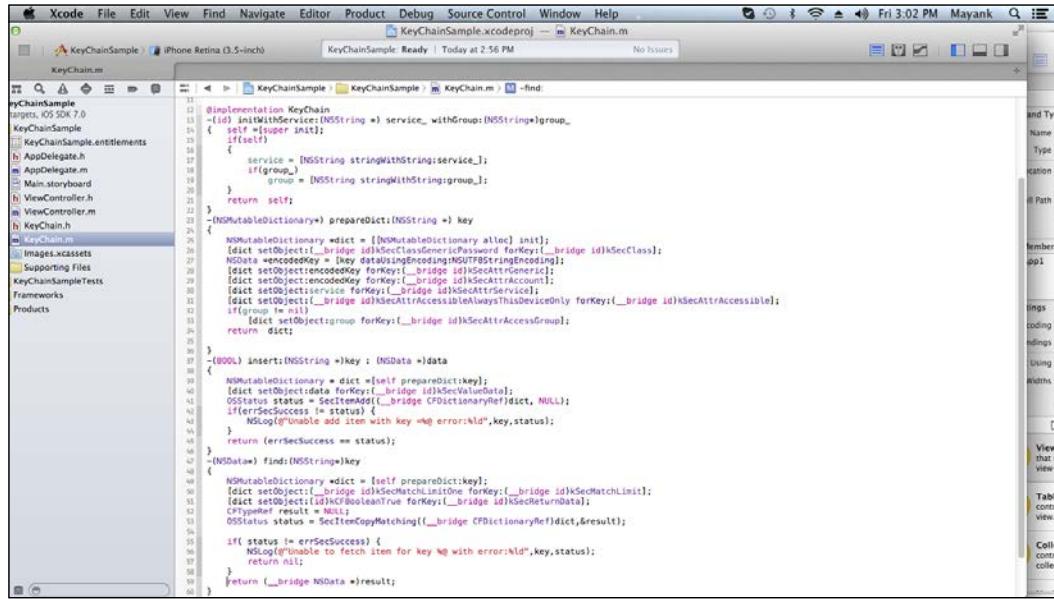
}
-(BOOL) insert:(NSString *)key : (NSData *)data
{
    NSMutableDictionary * dict =[self prepareDict:key];
    [dict setObject:data
        forKey:(__bridge id)kSecValueData];
    OSStatus status = SecItemAdd
        ((__bridge CFDictionaryRef)dict, NULL);
    if(errSecSuccess != status) {
        NSLog(@"%@",@"Unable add item with key %@ with
            error:%ld",key,status);
    }
    return (errSecSuccess == status);
}
-(NSData*) find:(NSString*)key
{
    NSMutableDictionary *dict = [self prepareDict:key];
    [dict setObject:(__bridge id)kSecMatchLimitOne
        forKey:(__bridge id)kSecMatchLimit];
    [dict setObject:(id)kCFBooleanTrue
        forKey:(__bridge id)kSecReturnData];
    CFTypeRef result = NULL;
    OSStatus status = SecItemCopyMatching
        ((__bridge CFDictionaryRef)dict,&result);

    if( status != errSecSuccess) {
        NSLog(@"%@",@"Unable to fetch item for key %@ with
            error:%ld",key,status);
        return nil;
    }
    return (__bridge NSData *)result;
}

```

Using iCloud and Security Services

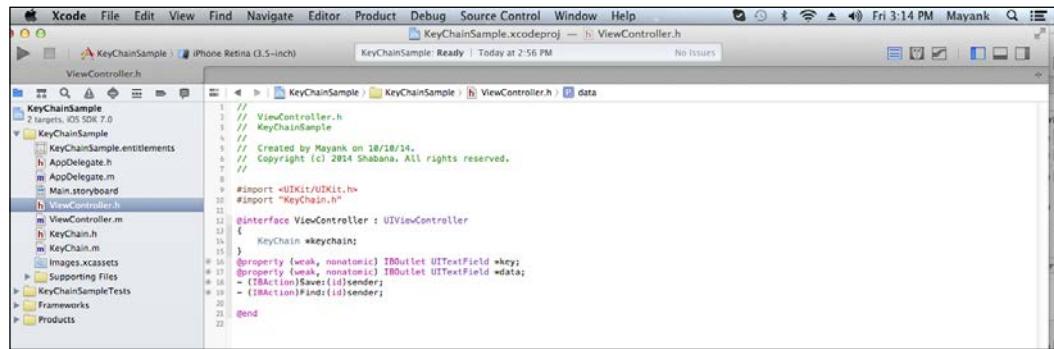
The `_bridge` object is used to transfer a pointer between Objective-C and the core foundation, with no transfer of ownership. Our interface file will probably look like this:



The screenshot shows the Xcode interface with the file `KeyChain.m` open. The code implements a `KeyChain` class with methods for initializing with service and group, preparing a dictionary for a key, inserting data for a key, and finding data for a key. It uses `SecItemCopyMatching` and `SecItemAdd` functions from the Security framework.

```
1 // 
2 // KeyChain.h
3 // KeyChainSample
4 // 
5 // Created by Mayank on 18/10/14.
6 // Copyright (c) 2014 Shabana. All rights reserved.
7 // 
8 #import <UIKit/UIKit.h>
9 #import "KeyChain.h"
10 @interface KeyChain : NSObject
11 {
12     _bridge *bridge;
13 }
14 
15 - (id) initWithService:(NSString *)service withGroup:(NSString *)group;
16 
17 - (id) initWithService:(NSString *)service;
18 
19 - (id) initWithGroup:(NSString *)group;
20 
21 - (id) init;
22 
23 - (id) init;
24 
25 - (NSMutableDictionary *) prepareDict:(NSString *)key;
26 
27 - (NSMutableDictionary *) prepareDict:(NSString *)key;
28 
29 - (NSMutableDictionary *) prepareDict:(NSString *)key;
30 
31 - (NSMutableDictionary *) prepareDict:(NSString *)key;
32 
33 - (NSMutableDictionary *) prepareDict:(NSString *)key;
34 
35 - (NSMutableDictionary *) prepareDict:(NSString *)key;
36 
37 - (NSMutableDictionary *) prepareDict:(NSString *)key;
38 
39 - (NSMutableDictionary *) prepareDict:(NSString *)key;
40 
41 - (NSMutableDictionary *) prepareDict:(NSString *)key;
42 
43 - (NSMutableDictionary *) prepareDict:(NSString *)key;
44 
45 - (NSMutableDictionary *) prepareDict:(NSString *)key;
46 
47 - (NSMutableDictionary *) prepareDict:(NSString *)key;
48 
49 - (NSMutableDictionary *) prepareDict:(NSString *)key;
50 
51 - (NSMutableDictionary *) prepareDict:(NSString *)key;
52 
53 - (NSMutableDictionary *) prepareDict:(NSString *)key;
54 
55 - (NSMutableDictionary *) prepareDict:(NSString *)key;
56 
57 - (NSMutableDictionary *) prepareDict:(NSString *)key;
58 
59 - (NSMutableDictionary *) prepareDict:(NSString *)key;
60 
61 - (NSMutableDictionary *) prepareDict:(NSString *)key;
62 
63 - (NSMutableDictionary *) prepareDict:(NSString *)key;
64 
65 - (NSMutableDictionary *) prepareDict:(NSString *)key;
66 
67 - (NSMutableDictionary *) prepareDict:(NSString *)key;
68 
69 - (NSMutableDictionary *) prepareDict:(NSString *)key;
70 
71 - (NSMutableDictionary *) prepareDict:(NSString *)key;
72 
73 - (NSMutableDictionary *) prepareDict:(NSString *)key;
74 
75 - (NSMutableDictionary *) prepareDict:(NSString *)key;
76 
77 - (NSMutableDictionary *) prepareDict:(NSString *)key;
78 
79 - (NSMutableDictionary *) prepareDict:(NSString *)key;
80 
81 - (NSMutableDictionary *) prepareDict:(NSString *)key;
82 
83 - (NSMutableDictionary *) prepareDict:(NSString *)key;
84 
85 - (NSMutableDictionary *) prepareDict:(NSString *)key;
86 
87 - (NSMutableDictionary *) prepareDict:(NSString *)key;
88 
89 - (NSMutableDictionary *) prepareDict:(NSString *)key;
90 
91 - (NSMutableDictionary *) prepareDict:(NSString *)key;
92 
93 - (NSMutableDictionary *) prepareDict:(NSString *)key;
94 
95 - (NSMutableDictionary *) prepareDict:(NSString *)key;
96 
97 - (NSMutableDictionary *) prepareDict:(NSString *)key;
98 
99 - (NSMutableDictionary *) prepareDict:(NSString *)key;
100 
101 - (BOOL) insert:(NSData *)data forKey:(NSString *)key;
102 
103 - (NSData *) find:(NSString *)key;
104 
105 - (NSData *) find:(NSString *)key;
106 
107 - (NSData *) find:(NSString *)key;
108 
109 - (NSData *) find:(NSString *)key;
110 
111 - (NSData *) find:(NSString *)key;
112 
113 - (NSData *) find:(NSString *)key;
114 
115 - (NSData *) find:(NSString *)key;
116 
117 - (NSData *) find:(NSString *)key;
118 
119 - (NSData *) find:(NSString *)key;
120 
121 - (NSData *) find:(NSString *)key;
122 
123 - (NSData *) find:(NSString *)key;
124 
125 - (NSData *) find:(NSString *)key;
126 
127 - (NSData *) find:(NSString *)key;
128 
129 - (NSData *) find:(NSString *)key;
130 
131 - (NSData *) find:(NSString *)key;
132 
133 - (NSData *) find:(NSString *)key;
134 
135 - (NSData *) find:(NSString *)key;
136 
137 - (NSData *) find:(NSString *)key;
138 
139 - (NSData *) find:(NSString *)key;
140 
141 - (NSData *) find:(NSString *)key;
142 
143 - (NSData *) find:(NSString *)key;
144 
145 - (NSData *) find:(NSString *)key;
146 
147 - (NSData *) find:(NSString *)key;
148 
149 - (NSData *) find:(NSString *)key;
150 
151 - (NSData *) find:(NSString *)key;
152 
153 - (NSData *) find:(NSString *)key;
154 
155 - (NSData *) find:(NSString *)key;
156 
157 - (NSData *) find:(NSString *)key;
158 
159 - (NSData *) find:(NSString *)key;
160 
161 - (NSData *) find:(NSString *)key;
162 
163 - (NSData *) find:(NSString *)key;
164 
165 - (NSData *) find:(NSString *)key;
166 
167 - (NSData *) find:(NSString *)key;
168 
169 - (NSData *) find:(NSString *)key;
170 
171 - (NSData *) find:(NSString *)key;
172 
173 - (NSData *) find:(NSString *)key;
174 
175 - (NSData *) find:(NSString *)key;
176 
177 - (NSData *) find:(NSString *)key;
178 
179 - (NSData *) find:(NSString *)key;
180 
181 - (NSData *) find:(NSString *)key;
182 
183 - (NSData *) find:(NSString *)key;
184 
185 - (NSData *) find:(NSString *)key;
186 
187 - (NSData *) find:(NSString *)key;
188 
189 - (NSData *) find:(NSString *)key;
190 
191 - (NSData *) find:(NSString *)key;
192 
193 - (NSData *) find:(NSString *)key;
194 
195 - (NSData *) find:(NSString *)key;
196 
197 - (NSData *) find:(NSString *)key;
198 
199 - (NSData *) find:(NSString *)key;
200 
201 - (NSData *) find:(NSString *)key;
202 
203 - (NSData *) find:(NSString *)key;
204 
205 - (NSData *) find:(NSString *)key;
206 
207 - (NSData *) find:(NSString *)key;
208 
209 - (NSData *) find:(NSString *)key;
210 
211 - (NSData *) find:(NSString *)key;
212 
213 - (NSData *) find:(NSString *)key;
214 
215 - (NSData *) find:(NSString *)key;
216 
217 - (NSData *) find:(NSString *)key;
218 
219 - (NSData *) find:(NSString *)key;
220 
221 - (NSData *) find:(NSString *)key;
222 
223 - (NSData *) find:(NSString *)key;
224 
225 - (NSData *) find:(NSString *)key;
226 
227 - (NSData *) find:(NSString *)key;
228 
229 - (NSData *) find:(NSString *)key;
230 
231 - (NSData *) find:(NSString *)key;
232 
233 - (NSData *) find:(NSString *)key;
234 
235 - (NSData *) find:(NSString *)key;
236 
237 - (NSData *) find:(NSString *)key;
238 
239 - (NSData *) find:(NSString *)key;
240 
241 - (NSData *) find:(NSString *)key;
242 
243 - (NSData *) find:(NSString *)key;
244 
245 - (NSData *) find:(NSString *)key;
246 
247 - (NSData *) find:(NSString *)key;
248 
249 - (NSData *) find:(NSString *)key;
250 
251 - (NSData *) find:(NSString *)key;
252 
253 - (NSData *) find:(NSString *)key;
254 
255 - (NSData *) find:(NSString *)key;
256 
257 - (NSData *) find:(NSString *)key;
258 
259 - (NSData *) find:(NSString *)key;
260 
261 - (NSData *) find:(NSString *)key;
262 
263 - (NSData *) find:(NSString *)key;
264 
265 - (NSData *) find:(NSString *)key;
266 
267 - (NSData *) find:(NSString *)key;
268 
269 - (NSData *) find:(NSString *)key;
270 
271 - (NSData *) find:(NSString *)key;
272 
273 - (NSData *) find:(NSString *)key;
274 
275 - (NSData *) find:(NSString *)key;
276 
277 - (NSData *) find:(NSString *)key;
278 
279 - (NSData *) find:(NSString *)key;
280 
281 - (NSData *) find:(NSString *)key;
282 
283 - (NSData *) find:(NSString *)key;
284 
285 - (NSData *) find:(NSString *)key;
286 
287 - (NSData *) find:(NSString *)key;
288 
289 - (NSData *) find:(NSString *)key;
290 
291 - (NSData *) find:(NSString *)key;
292 
293 - (NSData *) find:(NSString *)key;
294 
295 - (NSData *) find:(NSString *)key;
296 
297 - (NSData *) find:(NSString *)key;
298 
299 - (NSData *) find:(NSString *)key;
299 
```

7. Connect the UI components to our `viewController.h` file. Import `KeyChain` (the newly added Objective-C file) on top of the file and create an object of the `KeyChain` class in the interface as follows:



The screenshot shows the Xcode interface with the file `ViewController.h` open. The code defines a class `ViewController` that conforms to `UIViewControllerAnimated` and contains a property `KeyChain`. It also includes imports for `UIKit` and `KeyChain`.

```
1 // 
2 // ViewController.h
3 // KeyChainSample
4 // 
5 // Created by Mayank on 18/10/14.
6 // Copyright (c) 2014 Shabana. All rights reserved.
7 // 
8 #import <UIKit/UIKit.h>
9 #import "KeyChain.h"
10 @interface ViewController : UIViewController
11 {
12     KeyChain *keychain;
13 }
14 
15 @property (weak, nonatomic) IBOutlet UITextField *key;
16 
17 @property (weak, nonatomic) IBOutlet UITextField *data;
18 
19 - (IBAction)Save:(id)sender;
20 
21 - (IBAction)Find:(id)sender;
22 
```

8. Add the code in the `IBAction` methods of our button with some more code as follows in the `viewController.m` file:

```
#define SERVICE_NAME @"ANY_NAME_FOR_YOU"
#define GROUP_NAME @"iOS"
@interface ViewController ()
@end
@implementation ViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    keychain = [[KeyChain alloc] initWithService:SERVICE_NAME
withGroup:nil];
}

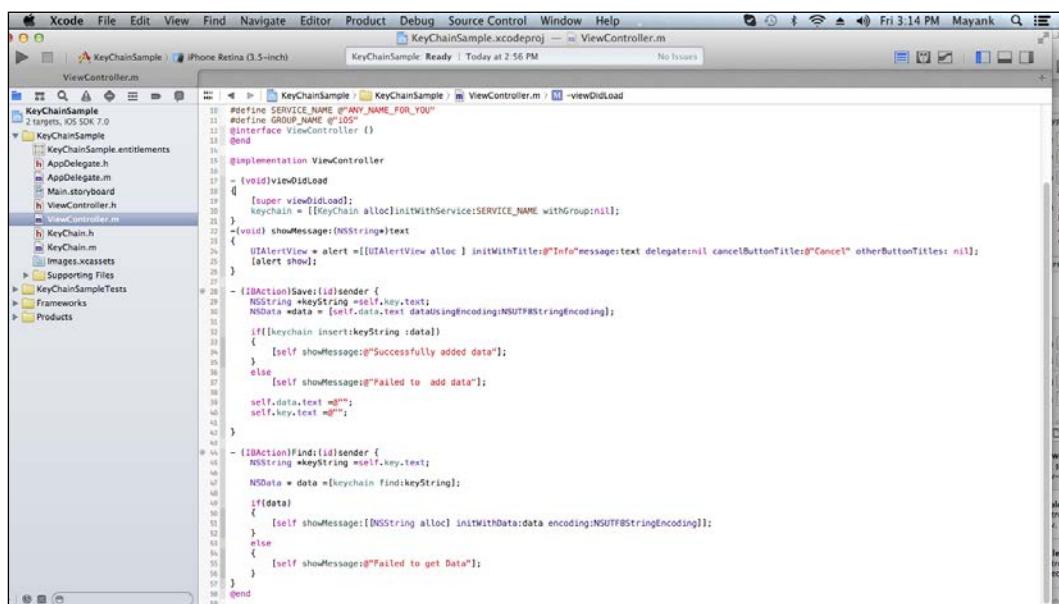
- (void) showMessage:(NSString*)text
{
    UIAlertView * alert =[[UIAlertView alloc ]
    initWithTitle:@"Info" message:text delegate:nil
    cancelButtonTitle:@"Cancel" otherButtonTitles: nil];
    [alert show];
}
- (IBAction)save:(id)sender {
    NSString *keyString =self.keyTextField.text;
    NSData *data = [self.dataTextField.text
    dataUsingEncoding:NSUTF8StringEncoding];
    if([keychain insert:keyString :data])
    {
        [self showMessage:@"Successfully added data"];
    }
    else
        [self showMessage:@"Failed to add data"];
    self.dataTextField.text =@";
    self.keyTextField.text =@";
}
- (IBAction)find:(id)sender {
    NSString *keyString =self.keyTextField.text;
    NSData * data =[keychain find:keyString];
    if(data)
    {
        [self showMessage:[NSString alloc]
        initWithData:data encoding:NSUTF8StringEncoding]];
    }
    else
    {
```

Using iCloud and Security Services

```
[self showMessage:@"Failed to get Data"];
}

- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    return YES;
}
```

Our interface file will probably look like this:



The screenshot shows the Xcode interface with the 'ViewController.m' file open. The code implements a View Controller that interacts with the Keychain. It includes methods for saving data to the keychain and retrieving data from it. The code uses UIAlertView to show messages to the user.

```
#define SERVICE_NAME @"ANY_NAME_FOR_YOU"
#define GROUP_NAME @"105"

@interface ViewController : UIViewController
{
    KeyChain *keychain;
}

@implementation ViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    keychain = [[KeyChain alloc] initWithService:SERVICE_NAME withGroup:nil];
}

- (void)showMessage:(NSString*)text
{
    UIAlertView * alert =[UIAlertView alloc] initWithTitle:@"Info" message:text delegate:nil cancelButtonTitle:@"Cancel" otherButtonTitles: nil];
    [alert show];
}

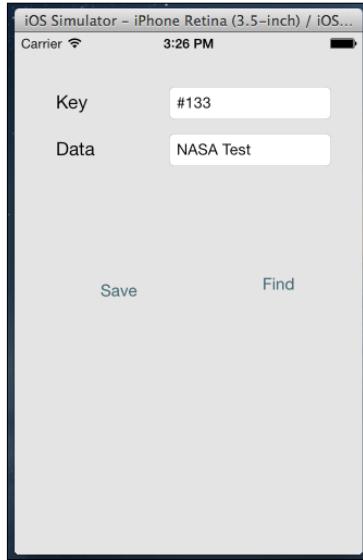
- (IBAction)Save:(id)sender {
    NSString *keyString = self.key.text;
    NSData *data = [self.data.text dataUsingEncoding:NSUTF8StringEncoding];

    if([keychain insert:keyString:data])
    {
        [self showMessage:@"Successfully added data"];
    }
    else
    {
        [self showMessage:@"Failed to add data"];
    }
}

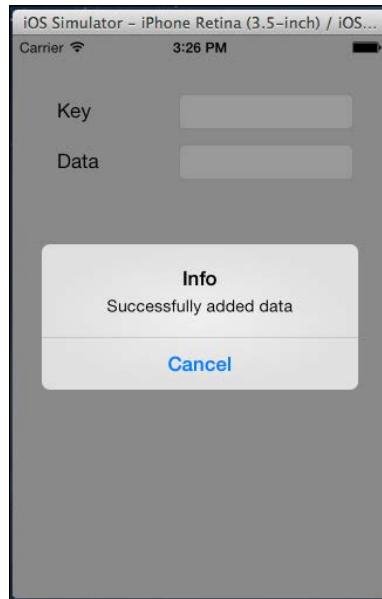
- (IBAction)Find:(id)sender {
    NSString *keyString = self.key.text;
    NSData *data = [keychain find:keyString];
    if(data)
    {
        [self showMessage:[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
    }
    else
    {
        [self showMessage:@"Failed to get Data"];
    }
}

```

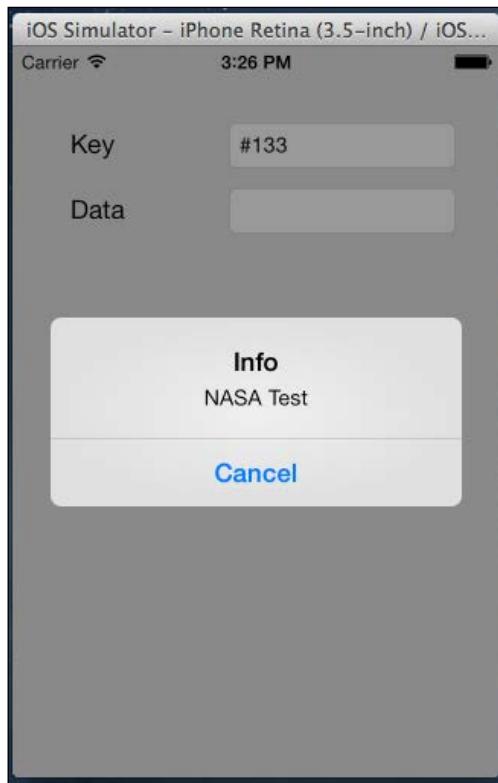
9. Compile and run the project; our simulator looks like the following screenshot. Enter the values and click on the **Save** button:



10. After clicking on the **Save** button, one pop up will appear (as shown in the following screenshot) with a message that our data is saved in the keychain securely.

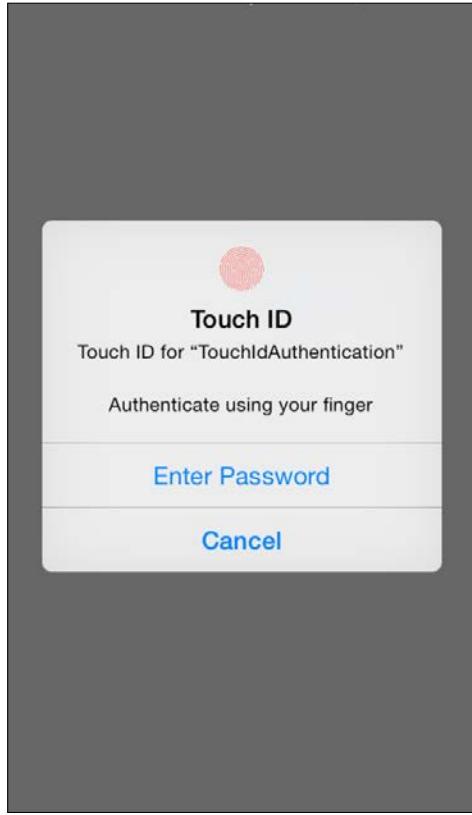


11. Now write the **Key** value in the text field that exists in the keychain storage and click on the **Find** button. It will show data in a pop-up message as follows:



The Touch ID API

Apple introduced a new feature in iOS 7 called **Touch ID authentication**. Previously, there was only four-digit passcode security in iPhones; now, Apple has extended security and introduced a new security pattern in iPhones. In Touch ID authentication, our fingerprint acts as a password. After launching the Touch ID fingerprint-recognition technology in the iPhone 5S last year, Apple is now providing it for developers with iOS 8. Now, third-party apps will be able to use Touch ID for authentication in the new iPhone and iPad OSes. Accounting apps, and other apps that contain personal and important data, will be protected with Touch ID. Now, you can protect all your apps with your fingerprint password.



There are two ways to use Touch ID as an authentication mechanism in our iOS 8 applications. They are explained in the following sections.

Touch ID through touch authentication

The Local Authentication API is an API that returns a Boolean value to accept and decline the fingerprint. If there is an error, then an error code gets executed and tells us what the issue is.

Certain conditions have to be met when using Local Authentication. They are as follows:

- The application must be in the foreground (this doesn't work with background processes)
- If you're using the straight Local Authentication method, you will be responsible for handling all the errors and properly responding with your UI to ensure that there is an alternative method to log in to your apps

Touch ID through Keychain Access

Keychain Access includes the new Touch ID integration in iOS 8. In Keychain Access, we don't have to work on implementation details; it automatically handles the passcode implementation using the user's passcode. Several keychain items can be chosen to use Touch ID to unlock the item when requested in code through the use of the new **Access Control Lists (ACLs)**. ACL is a feature of iOS 8. If Touch ID has been locked out, then it will allow the user to enter the device's passcode to proceed without any interruption.

There are some features of Keychain Access that make it the best option for us. They are listed here:

- Keychain Access uses Touch ID, and its attributes won't be synced by any cloud services. So, these features make it very safe to use.
- If users overlay more than one query, then the system gets confused about correct user, and it will pop up a dialog box with multiple touch issues.

Using the Local Authentication framework

Apple provides a framework to use Touch ID in our app called **Local Authentication**. This framework was introduced for iOS 8. To make an app, including the Touch ID authentication, we need to import this framework in our code. It is present in the framework library of Apple. Let's see how to use the Local Authentication framework:

1. Import the Local Authentication framework as follows:

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

This framework will work on Xcode 6 and above.

2. To use this API, we have to create a Local Authentication context, as follows:

```
LAContext *passcode = [[LAContext alloc] init];
```

3. Now, check whether Touch ID is available or not and whether it can be used for authentication:

```
- (BOOL)canEvaluatePolicy:(LAPolicy)policy error:  
    (NSError * __autoreleasing *)error;
```

4. To display Touch ID, use the following code:

```
- (void)evaluatePolicy:(LAPolicy)policy localizedReason:  
    (NSString *)localizedReason  
    reply:(void(^)(BOOL success, NSError *error))reply;
```

5. Take a look at the following example of Touch ID:

```
LAContext *passcode = [[LAContext alloc] init];
NSError *error = nil;
NSString *Reason =
    <#String explaining why our app needs authentication#>;
if ([passcode canEvaluatePolicy:
    LAPolicyDeviceOwnerAuthenticationWithBiometrics
    error:&error])
{
    [passcode evaluatePolicy:
        LAPolicyDeviceOwnerAuthenticationWithBiometrics
        localizedReason:Reason reply:^(BOOL success, NSError
        *error) {
            if (success)
            {
                // User authenticated successfully
            } else
            {
                // User did not authenticate successfully,
                // go through the error
            }
        }];
}
else
{
    // could not go through policy look at error and show
    // an appropriate message to user
}
```

Summary

In this chapter, you learned how to push your data to iCloud and how to save your private data, such as passwords, account numbers, ATM pins, and so on, to Keychain. We also focused on the Touch ID API, which was introduced in iOS 8. In the next chapter, you will learn how to push your app on the App Store.

7

The App-distribution Program

This chapter will help you understand the anatomy of becoming an iOS application developer. We will start from no account to publishing on the App Store.

In this chapter, we will cover the following topics:

- Understanding and setting up your developer account
- Setting up a provisioning profile
- Publishing the app on the Store

Setting up a developer account

In this chapter, you will learn how to register for Apple's iOS Developer Program, how to generate the various required certificates, ways to configure our app, and finally, the step-by-step process to submit the app to the App Store for approval. For this chapter, we have created a fresh, new App Store account and, down the line, we have also submitted a new app to Apple's App Store.

We are going to submit the improved version of the *AntKiller* game app that we created in *Chapter 4, APIs Introduced in iOS 7*, using the SpriteKit game engine.

The following are the prerequisites to complete this chapter:

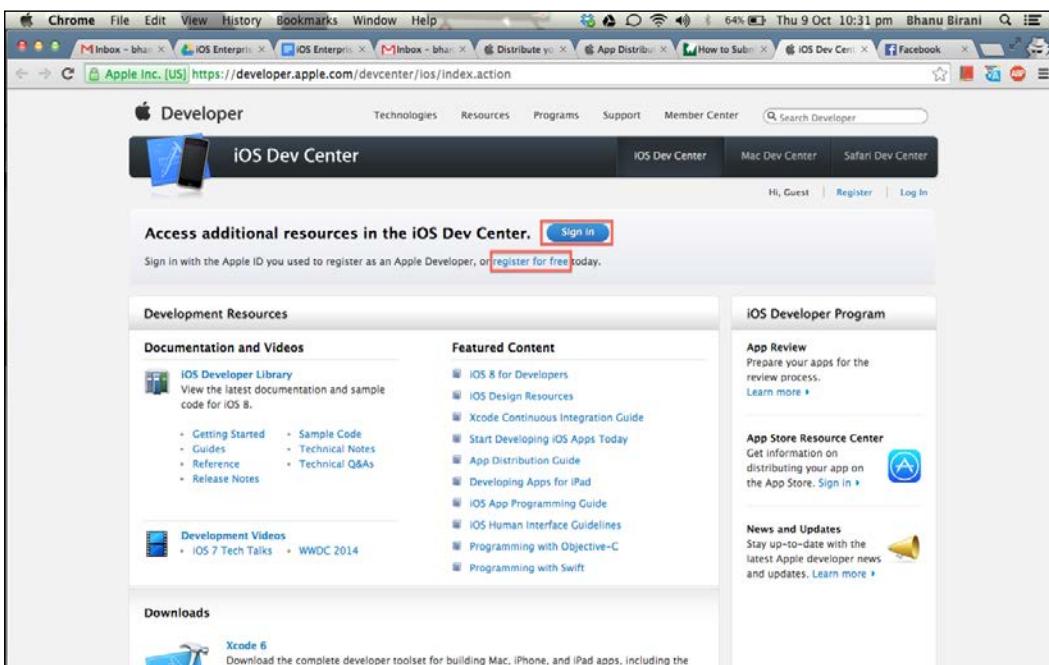
- You will need a web browser
- You will need a valid credit card
- You will also need \$99
- It's mandatory to have a Mac computer, preferably with OS X 10.9 Mavericks installed to develop apps
- Finally, you will need Xcode 6, Apple's development software; you can download it from <https://developer.apple.com/downloads/index.action?name=Xcode>

Xcode can be downloaded once you are registered as an Apple iOS developer. After completing the first few sections of this tutorial, you will be registered in the Apple Developer Program.

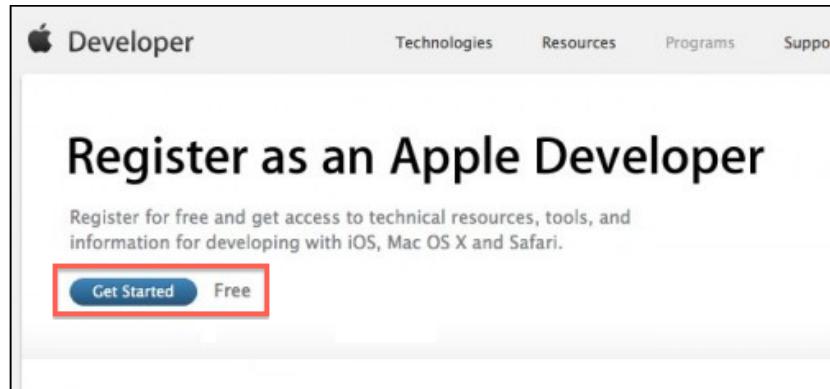
Getting started

Registering on Apple's Developer portal is our first step on the way to publish our app on the App Store. To become an iOS developer, you will have to pay the aforementioned fee of \$99 USD or pay \$299 USD for an Enterprise account. However, for an Enterprise account, you will need an additional **DUNS (Data Universal Numbering System)** number before registering your account.

If you already have a developer account with Apple, feel free to skip this section and move on to the next one. If you don't have an Apple developer account, go to the **iOS Dev Center** (<https://developer.apple.com/devcenter/ios/index.action>) and click on the register for free link, as shown in the following screenshot:

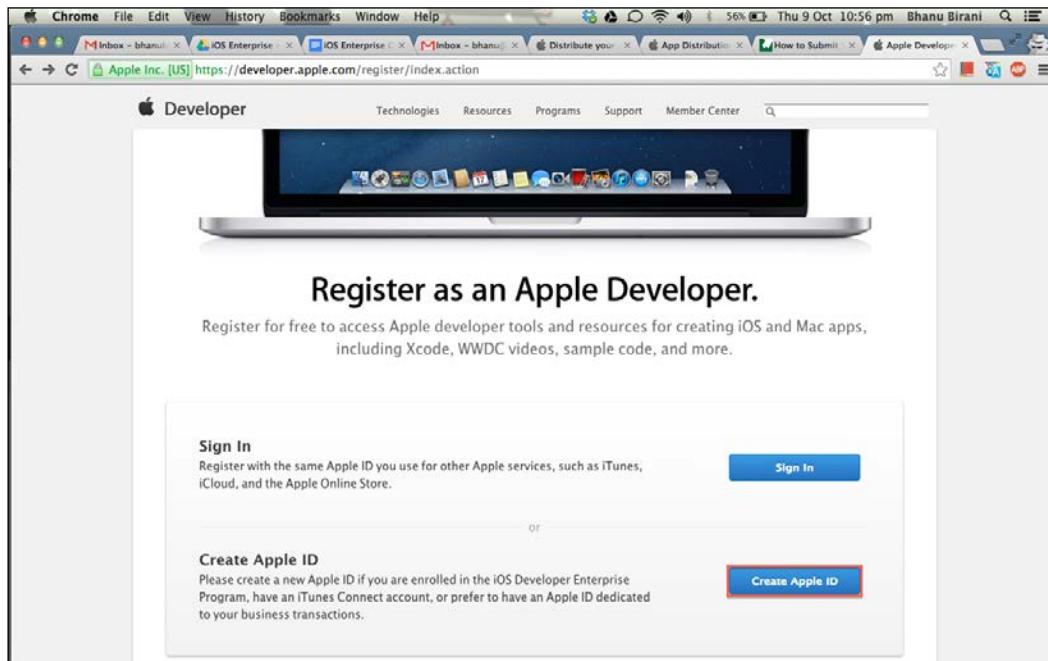


Click on the **Get Started** button as shown in the red highlighted box:



Here, on this page, you can either opt to create a fresh new Apple ID or use your existing Apple ID. If you want to save time, you can use the same Apple ID that you already use for your iTunes account purchases, for example. However, to follow the steps in the chapter, we would suggest that you create a fresh, new Apple ID. This will help you make sure that all your data is correct.

So, to create a new Apple ID, select **CreateApple ID** as shown in the following screenshot:

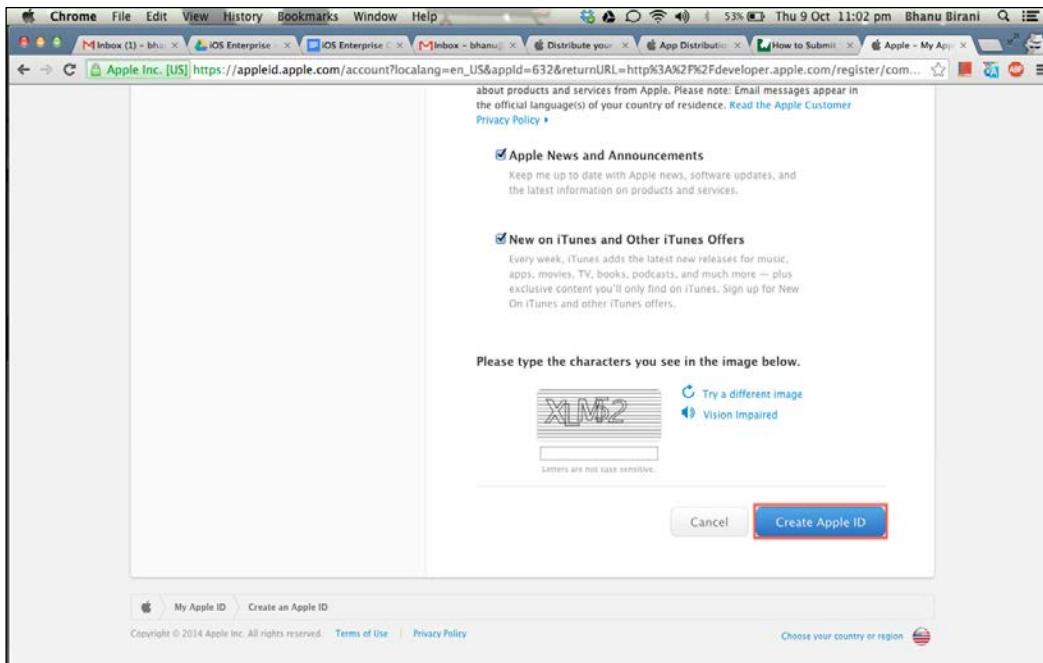


The App-distribution Program

Fill in the form (shown in the following screenshot) with all the required details such as your e-mail, password, and security information. It's recommended that you use the e-mail address that you check often, because Apple sends all the updates of your Developer Program and the status of the apps that you've submitted for approval.

The screenshot shows a web browser window with the URL https://appleid.apple.com/account?localLang=en_US&appId=632&returnURL=http%3A%2F%2Fdeveloper.apple.com%2Fregister%2Fcom.... The page title is "My Apple ID". On the left, there is a sidebar with the heading "Create an Apple ID." and a detailed description of what an Apple ID is. Below the description is a link to "Read the Apple Customer Privacy Policy". On the right, there is a main form titled "Create an Apple ID." It has sections for "Name", "Apple ID and Password", and "Security Questions". The "Name" section includes fields for First Name, Middle Name (optional), and Last Name. The "Apple ID and Password" section includes fields for Apple ID (example: jappleseed@example.com), Password, and Confirm Password. The "Security Questions" section is partially visible at the bottom.

Fill in all your personal information by scrolling down and then click on **Create Apple ID**, as shown in the following screenshot:



The next page holds questions related to your professional profile. This page will also ask you about the apps you develop and so on. Fill in all the answers as per your best knowledge. Scroll down and complete the following form and the click on **Continue**:

Please select the primary category for your application(s)

- Free Applications
- Commercial Applications
- Enterprise (In-House) Applications
- Web Applications

How many years have you been developing on Apple platforms?

- New to Apple platforms
- < 1 year
- 1 to 3 years
- 3 to 5 years
- 5+ years

Do you develop on other mobile platforms?

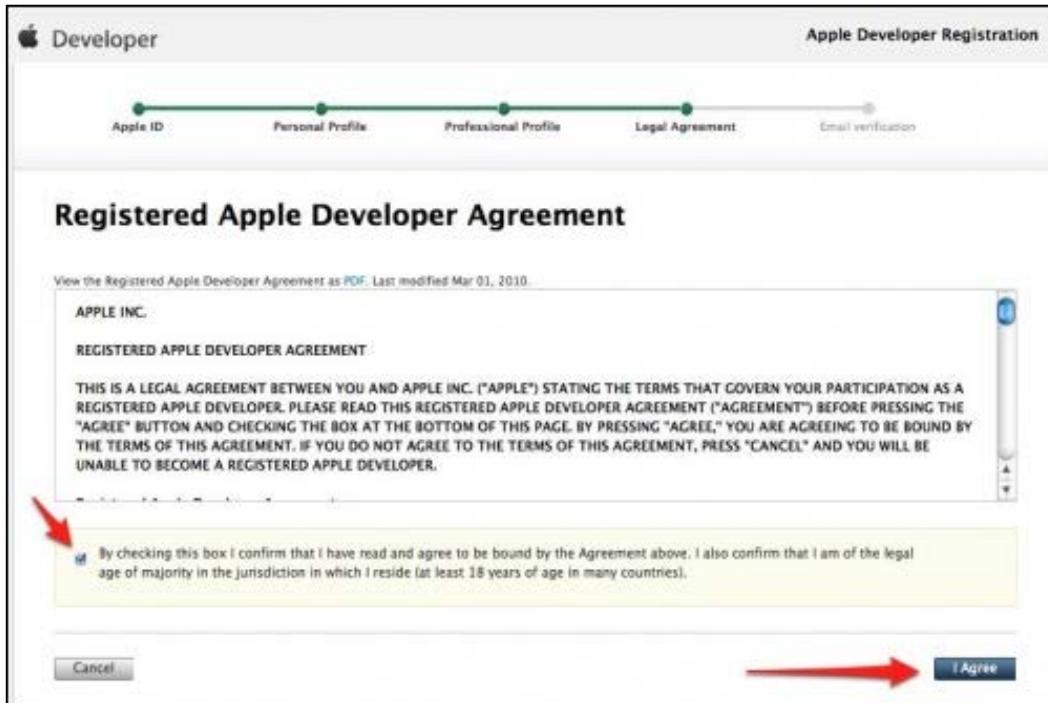
- Yes
- No

 **Continue**

The Continue button

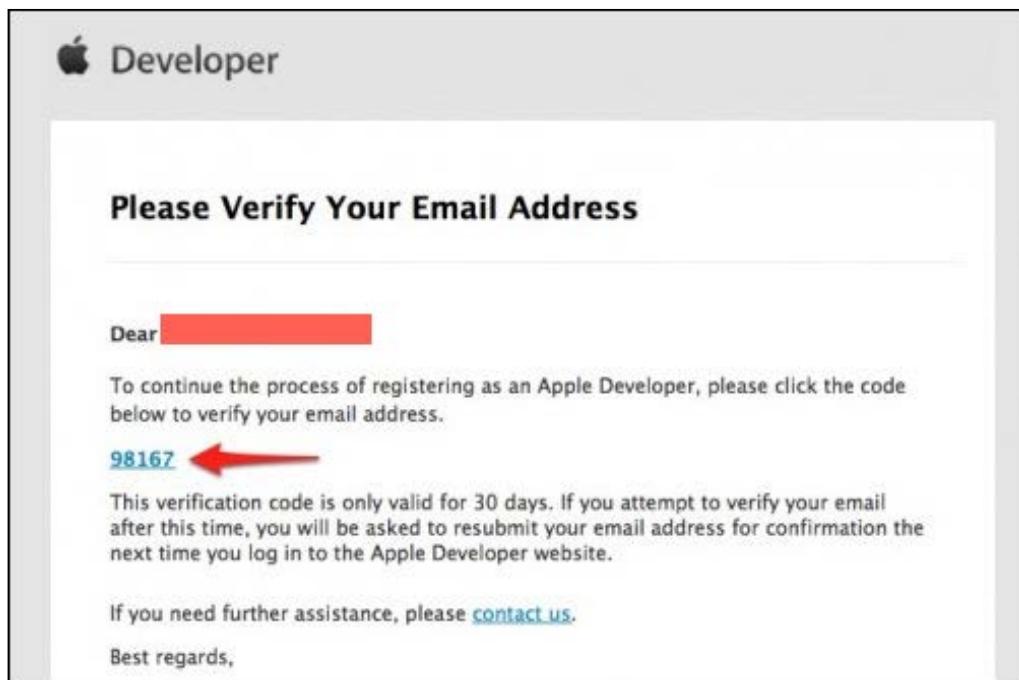
The App-distribution Program

The next page is the legal agreement. Read the document, select the checkbox, and click on **I Agree**, as shown in the following screenshot:

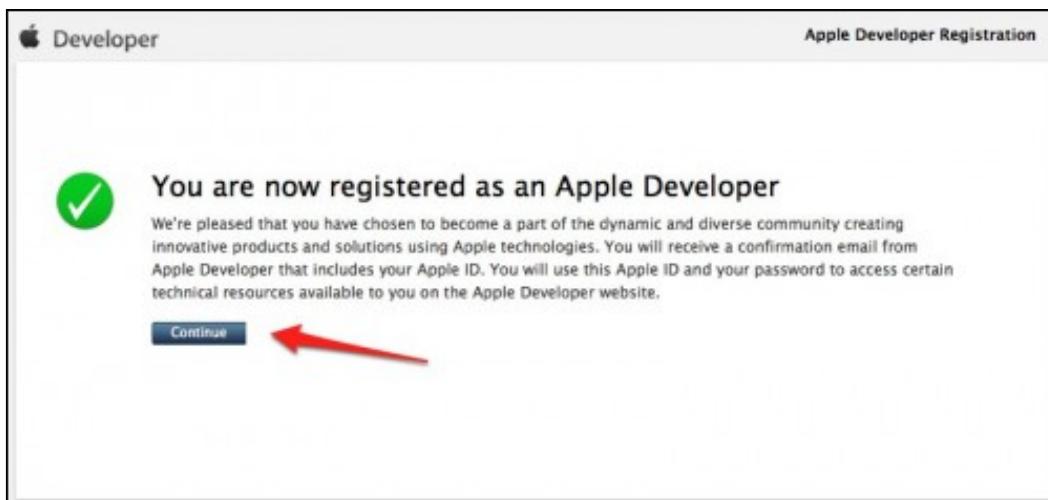


Legal agreement

Now, the registration is completed. Just verify your e-mail address that you used to register. You should receive an e-mail to confirm your e-mail address. Click on the link sent in the Apple registration confirmation mail as shown in the following screenshot:



Congratulations! You are now an Apple developer. Before you can start developing iOS applications and submitting them to Apple's App Store, you need to click on **Continue** (as shown in the following screenshot) and buy the iOS application development program:



The Continue button

Joining the iOS Developer Program

Using your developer Apple ID provides you with access to many resources and information. However, it still doesn't allow you to publish your apps on Apple's App Store. To gain more access to Apple's app-publishing feature, you will have to enroll yourself in Apple's iOS Developer Program. This, in turn, will cost you \$99 USD per year.

After clicking on **Continue** on the page shown in the preceding screenshot, you will be redirected to the right place. Otherwise, if you already have an Apple developer account and have chosen to skip the previous section, then go to the **Developer Member Center**, log in, and you'll be in sync.

Once you are logged in, just click on the **Join Today** link that is on the top-right corner of the page, flagged in the following screenshot:



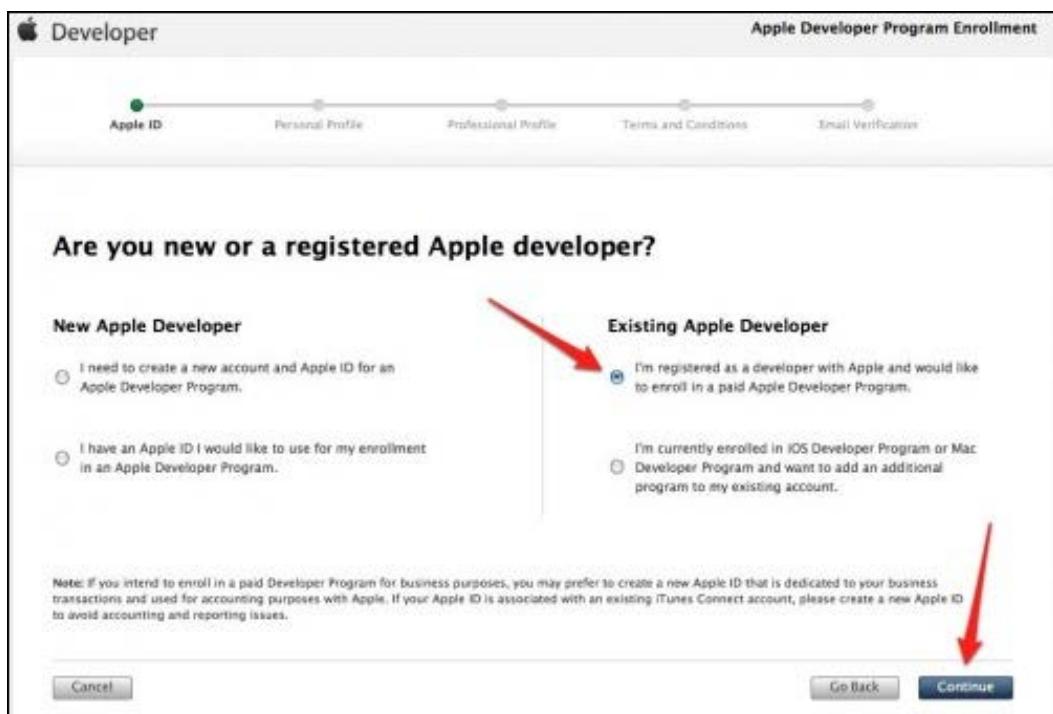
The Join Today link

Then, click on **Continue** on the following screenshot:



The Continue button

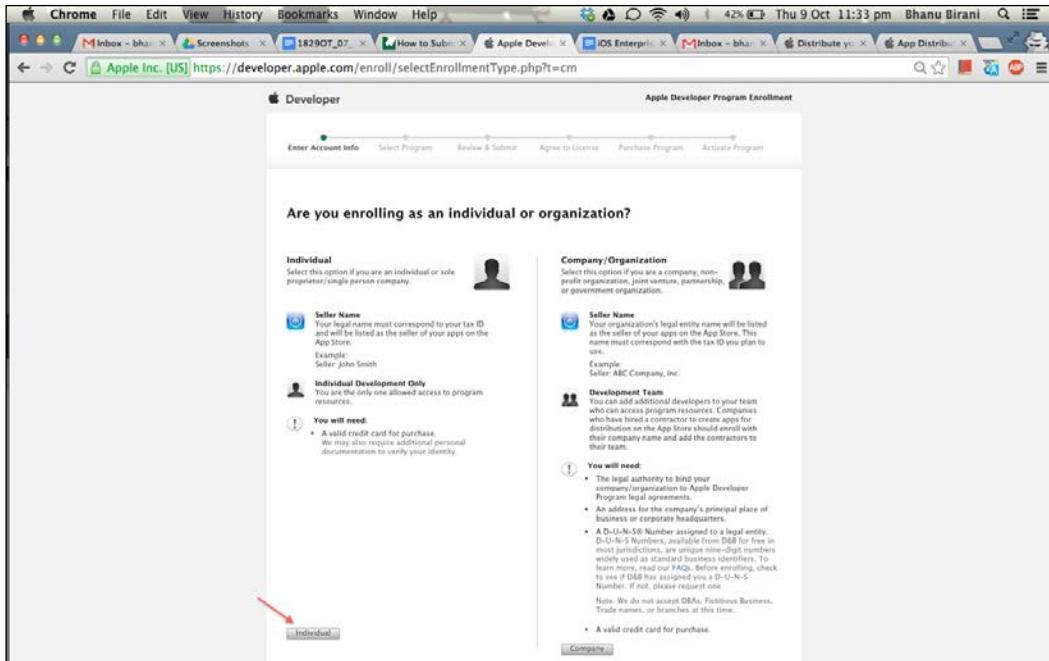
Now, select Existing Apple Developer and click on Continue, as shown here:



Selecting the Existing Apple Developer option and the Continue button

The App-distribution Program

On the next page, we need to decide whether we want to enroll as an individual or as a company. In this chapter, we will enroll ourselves as an individual, as shown in the following screenshot. Enrolling yourself as a company is a little complicated, as it needs you to submit a lot more paperwork to prove your involvement in the company.



The Individual button

Now, when you click on **Individual**, the next screen will be the login screen. Here, you will have to provide your newly created credentials again. After this, click on **Sign In**.

Now, in the next screen after you log in, you will have to provide your billing information to verify your identity, as shown in the following screenshot. To make sure that you have provided the correct card information, Apple attempts to confirm this information with your credit card company. Always make sure that you have provided the correct details.

Developer

Apple Developer Program Enrollment

Enter Account Info Select Program Review & Submit Agree to License Purchase Program Activate Program

Enter your billing information for identity verification

(All form fields are required)

Please enter your information using plain text.
Diacritical characters will be converted to English characters (for example: u instead of ü, n instead of ñ).

Your Name

IMPORTANT – Enter your name EXACTLY as it appears on the credit card you intend to use during the Apple Online Store purchase process. Please do not use an abbreviation or nickname unless this is how your name appears on your credit card.

First Name: Gustavo

Last Name: Ambrozio

Fill in the complete form by scrolling down and then click on **Continue**, as shown here:

Developer

Apple Developer Program Enrollment

Enter Account Info Select Program Review & Submit Agree to License Purchase Program Activate Program

Enter your contact information.

Legal Name
To edit your name, visit My Apple ID.
Name: Bhanu Birani

Credit Card Billing Address
Enter your address in your local language exactly as it appears on your credit card bill.

Country: India
Street Address: _____
City/Town: _____
State/Province: Select State _____
Postal Code: _____
Phone: _____ Country Code, Area/City Code and Number, Extension

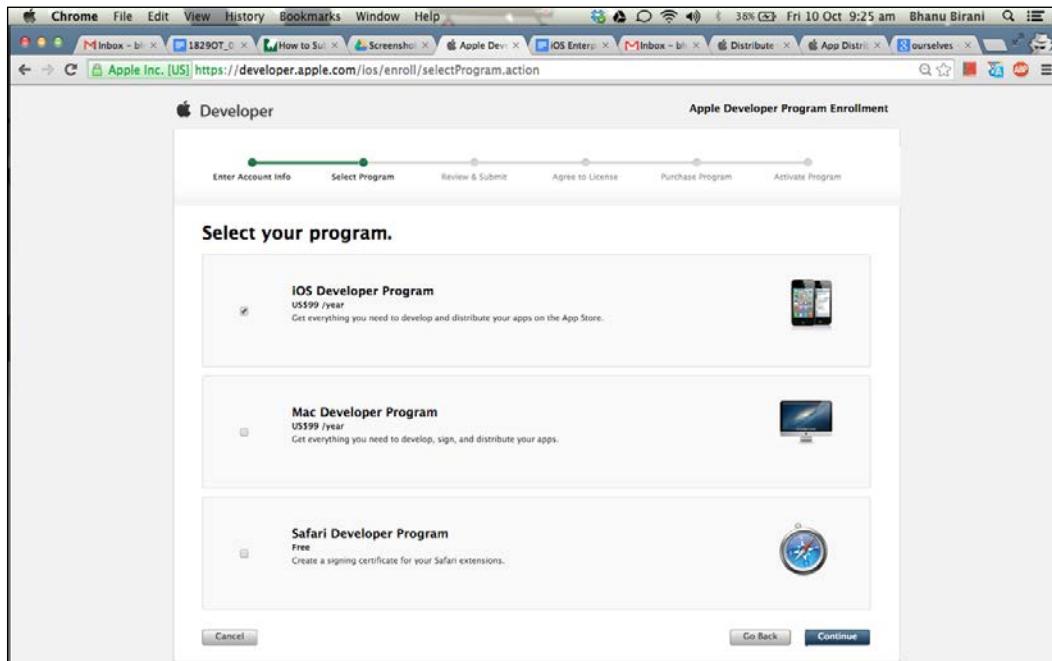
Continue

The Continue button

The App-distribution Program

Clicking on **Continue** will redirect you to the program-selection page. Now, we have to select a program. In this chapter, our primary focus is on the iOS Developer Program, so we are going to select this option only.

After selecting your program, click on **Continue**, as shown in the following screenshot:



The Continue button

Now, once you have clicked on **Continue**, you will be asked to review your information to correct any visible mistakes if necessary. Once you have reviewed this, you can submit it by clicking on **Continue**, as shown in the following screenshot:

Review your enrollment information & submit.

Developer Program  **iOS Developer Program**
US\$99/year

Personal Profile

Name:	Gustavo Ambrozio
Email:	gustavo@codecrop.com
Address:	[REDACTED]
City:	[REDACTED]
State:	[REDACTED]
Postal Code:	[REDACTED]
Country:	United States
Phone:	[REDACTED]

Your Personal Contact Info
To ensure your enrollment is processed properly, use your real first and last name and refrain from using aliases or organization names within the name fields of your Personal Profile.

App Store Distribution
Your name will appear as the "seller" for apps you distribute on the App Store. [View example](#)

Billing Info

Name:	Gustavo Ambrozio
Email:	gustavo@codecrop.com
Address:	[REDACTED]
City:	[REDACTED]
State:	[REDACTED]
Postal Code:	[REDACTED]
Country:	United States
Phone:	[REDACTED]

Identity Verification
Ensure your name and address are EXACTLY as it appears on the credit card you intend to use during the Apple Online Store purchase process.

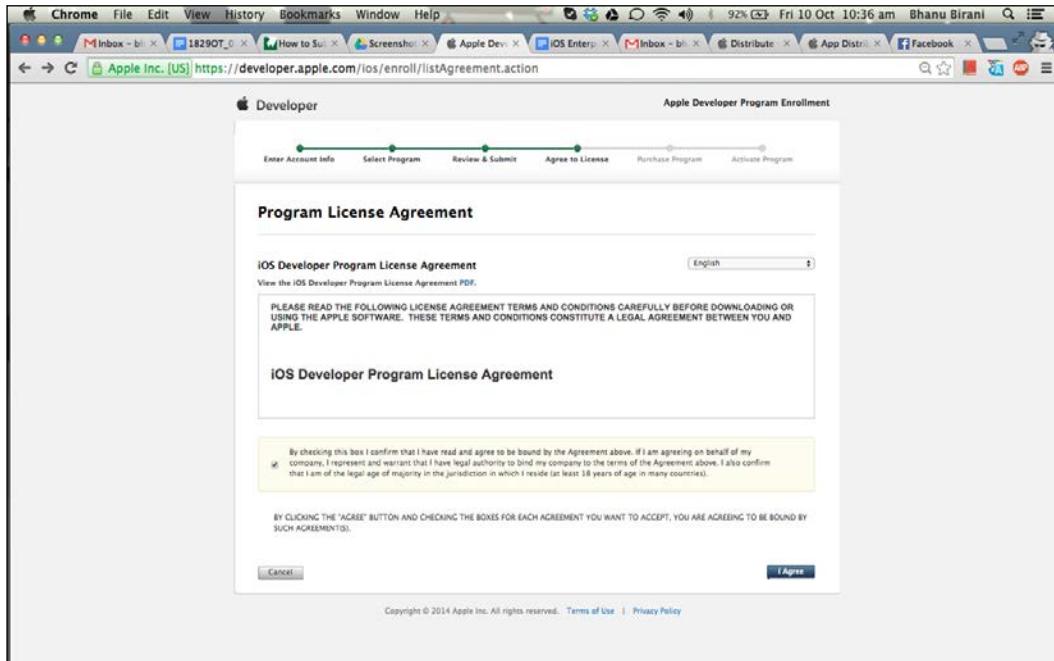
Your privacy is a priority at Apple and we go to great lengths to protect it. To learn how Apple safe-guards your personal information, please review the [Apple Customer Privacy Policy](#).



Cancel **Go Back** **Continue**

The App-distribution Program

Now, accept the license agreement and click on **I Agree**, as shown here:



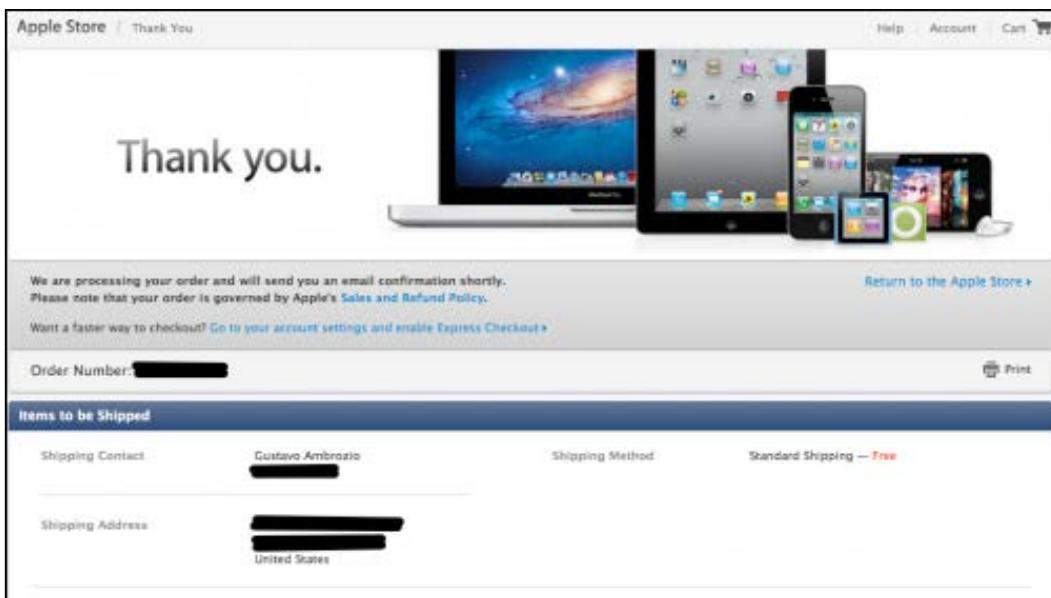
License agreement

You will be redirected to a payment-information page as shown in the following screenshot. Fill in all the details related to your payment and shipping address and then click on **Continue**.

The screenshot shows a payment-information page for the iOS Developer Program. The purchase item is listed as 'iOS Developer Program' with a price of '\$99.00'. Below this, it says 'Your order will be charged in U.S. dollars' and 'Order Total: \$99.00'. The 'Payment Information' section contains fields for a credit card: 'Type' (dropdown menu), 'Number', 'Cardholder's Name', and 'Expires'. The 'Billing information' section contains fields for address details: 'Country' (dropdown menu), 'Address Line 1', 'Address Line 2', 'City/Town', 'State, Province, or Region' (dropdown menu), 'Postal Code', and 'Phone'. At the bottom are 'Cancel' and 'Continue' buttons.

The payment-information page

Now, follow the payment procedure; once the payment of \$99 USD is successfully done, you will be redirected back to the Apple payment-confirmation page, as shown in the following screenshot:



The payment-confirmation page

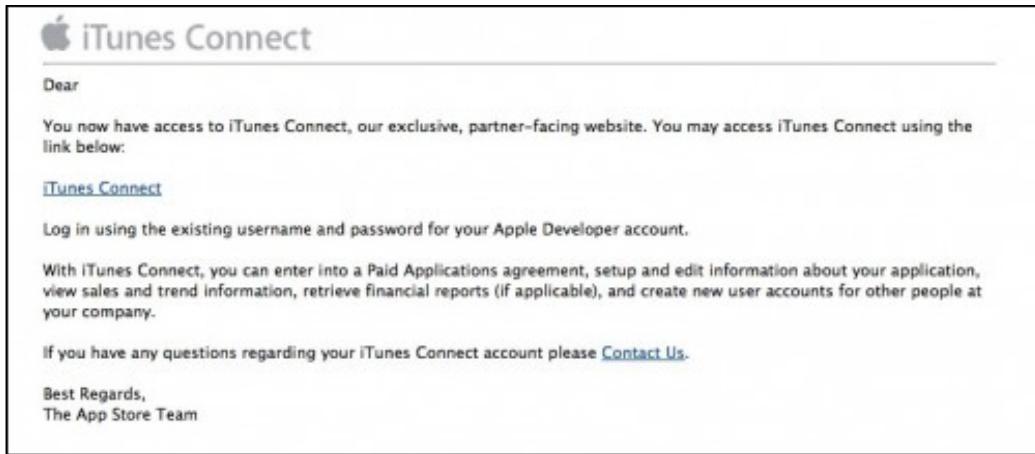
Getting started with the iOS Developer account

Once you submit and pay for your iOS developer registration process, you will have to wait for a day or so for Apple to process your order.

Once your order is successfully processed, you will receive an e-mail from Apple, like the one shown in the following screenshot:

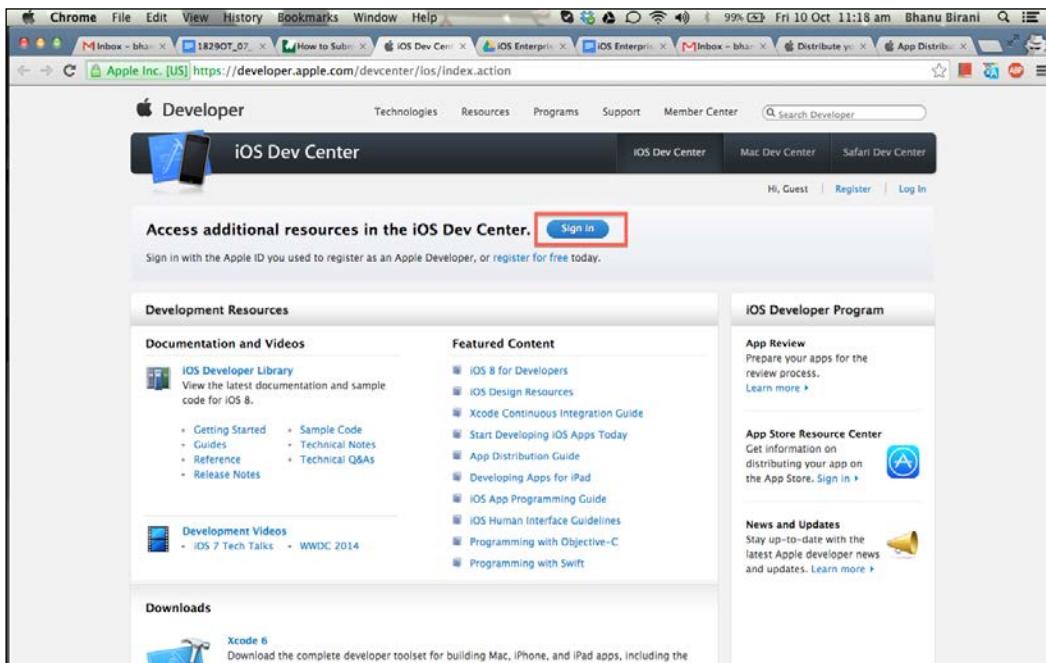


You will also receive an e-mail for your **iTunes Connect** account, as shown here:



Now, you can download Xcode, the tool mainly used for iOS development, if you have not already done so, using your new iOS developer credentials. We will briefly use Xcode in this chapter to understand the process of publishing an app to the App Store.

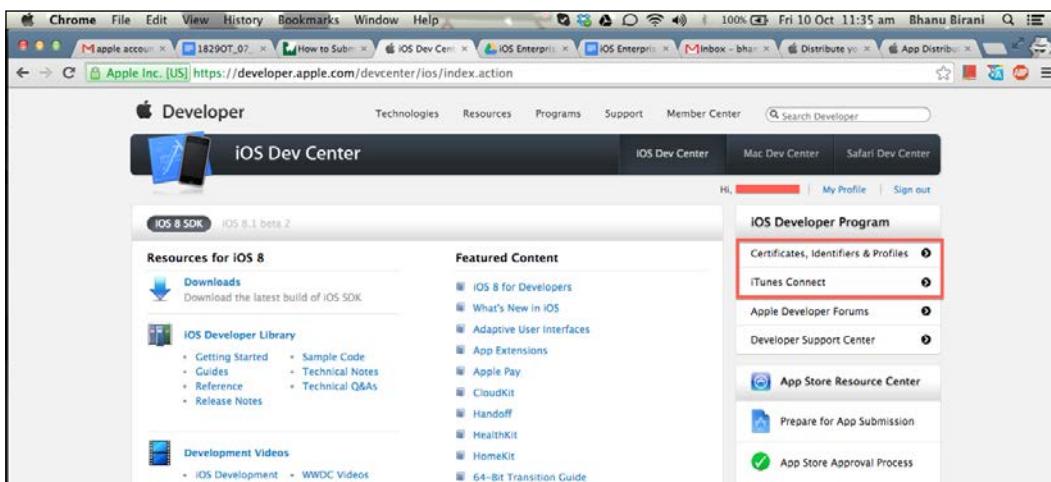
Now, go to the **iOS Dev Center** (<https://developer.apple.com/devcenter/ios/index.action>) and click on **Sign In**, as shown in the following screenshot:



The App-distribution Program

After logging in with your credentials, you will finally observe the changes. The iOS Dev Center has a lot of information for you now. You will find various programming guides, downloads, **Worldwide Developers Conference (WWDC)** videos, and documentations; the most important part is the very helpful developer forum and a support center.

In this chapter, our primary focus is on the two portals that will be widely used while developing your iOS apps: the **iOS Provisioning Portal** and **iTunes Connect**, shown in the following screenshot:



The iOS Provisioning Portal

It's a fact that a non-jailbroken iOS device is only able to install and run apps that are approved by Apple and are available on the App Store.

Apple has a special security check to accomplish this; every app run on an iOS device must have a signed Apple certificate. However, for the App Store, applications come with a bundled certificate that is verified by the system before it allows the app to run. The app is considered as invalid if no signature is found, and the system refuses to run such apps.

Developers have to compile and run their apps very often while developing them. To compile and run apps while developing, developers need a way to create and sign their apps using their own certificates.

This is where the iOS Provisioning Portal comes into picture. You can generate your profiles using this portal. Profiles can also be understood as "code-signing identities". These files are generated by the iOS Provisioning Portal, which is used by Xcode to sign your apps.

It is important to know about the two types of profiles: development profiles and distribution profiles. They are explained here:

- **Development profiles:** These profiles are tied to specific devices, so the app can only run on these devices. Developers can add their device UDID in the Developer portal; this will allow the profile to pass the validation while installing the app on the device.
- **Distribution profiles:** These profiles are specifically used to sign your app before you submit it to Apple for approval. These profiles do not hold any device-specific details; hence, you yourself can't install the app on your device using this profile. These profiles are deviceless profiles as, after approving the app, Apple will sign the app and publish it for all the users.

These Provisioning Portal are also used to generate push-notification certificates if you want your app to send them.

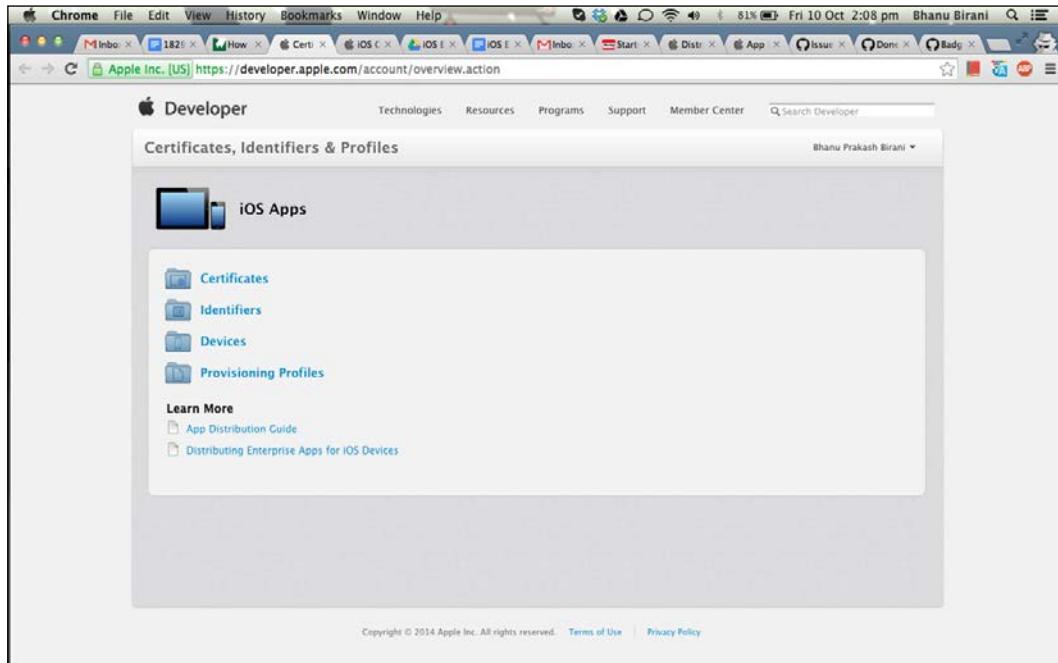
iTunes Connect

The iTunes Connect portal takes control of your app's submission process. In this portal, you will register your new apps along with their description, meta tags, screenshots, price, and in-app purchase to publish them to the App Store.

This also allows you to agree to new contracts. You can also set your financial data in this portal and check your sales. This portal gives access to various analytics such as crash reports, download information, and so on.

Certificates, Identifiers, and Profiles

We will explore the certificates, identifiers, and profiles in this section to set up the profiles for our app-deploying process; this app will be published later on the App Store. Now, click on **Certificates, Identifiers & Profiles**, as shown in the previous screenshot, or click on the link (<https://developer.apple.com/account/overview.action>). The resulting screenshot is as follows:



Click on **Certificates**, and you will see something similar to the following screenshot:

The screenshot shows a web browser window for the Apple Developer portal. The URL is <https://developer.apple.com/account/ios/certificate/certificateList.action>. The page title is "Certificates, Identifiers & Profiles". On the left, there is a sidebar with the following sections and their sub-options:

- iOS Apps**
 - Certificates
 - All
 - Pending
 - Development
 - Production
 - Identifiers
 - App IDs
 - Pass Type IDs
 - Website Push IDs
 - iCloud Containers
 - App Groups
 - Devices
 - All
 - Provisioning Profiles
 - All
 - Development
 - Distribution

iOS Certificates



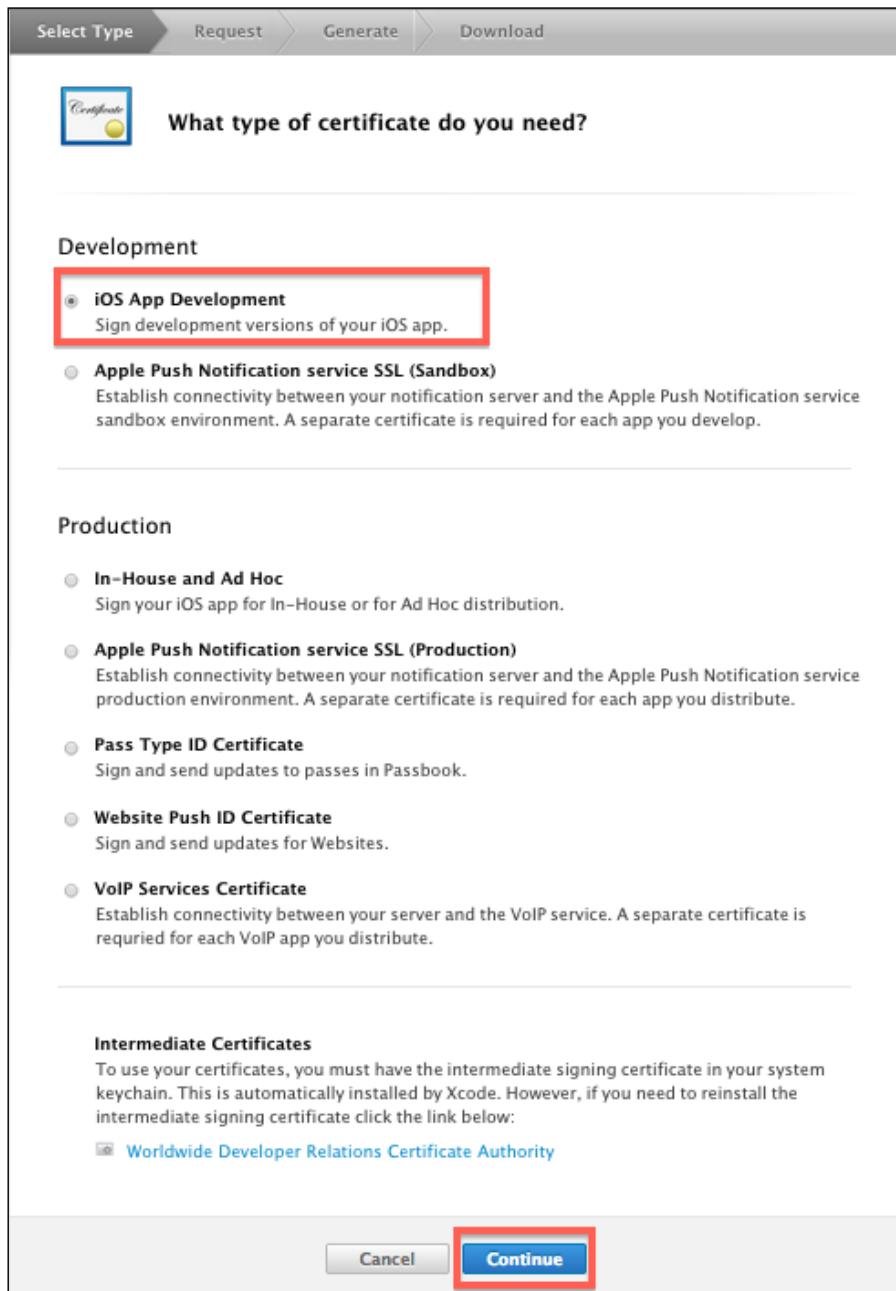
Certificates

Name	Type	Expires

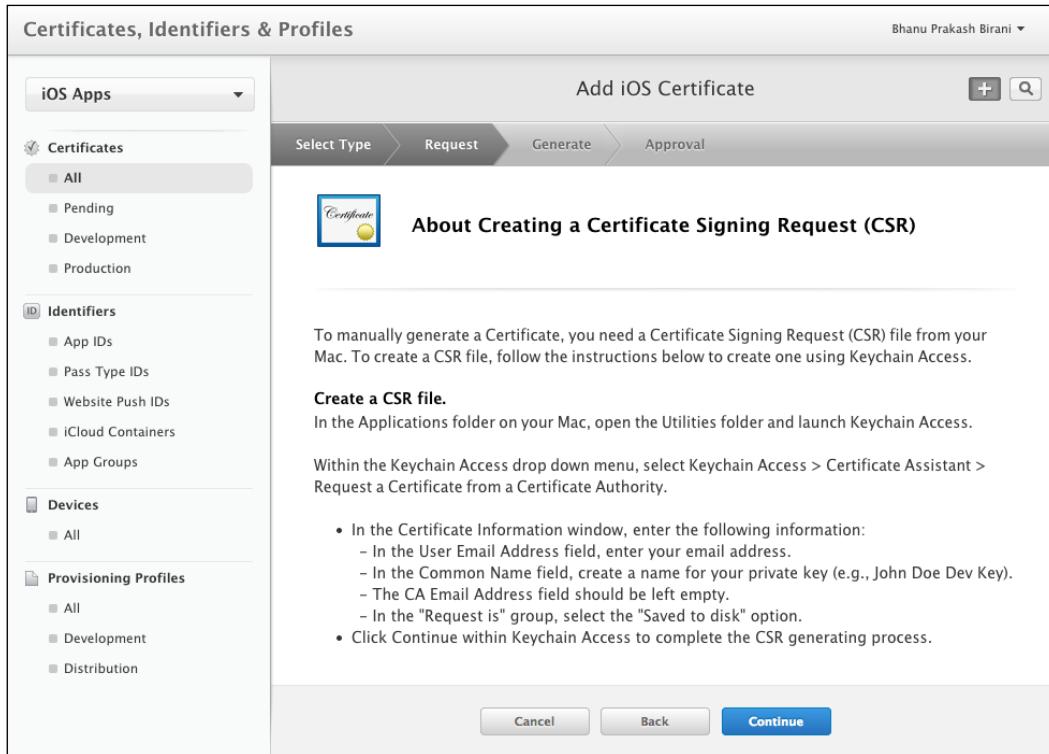
[171]

The App-distribution Program

Now, click on the + button to create a new certificate, and this page will ask you which certificate you want to create. Select the **iOS App Development** certificate from the **Development** section and click on **Continue**, as shown in the following screenshot:

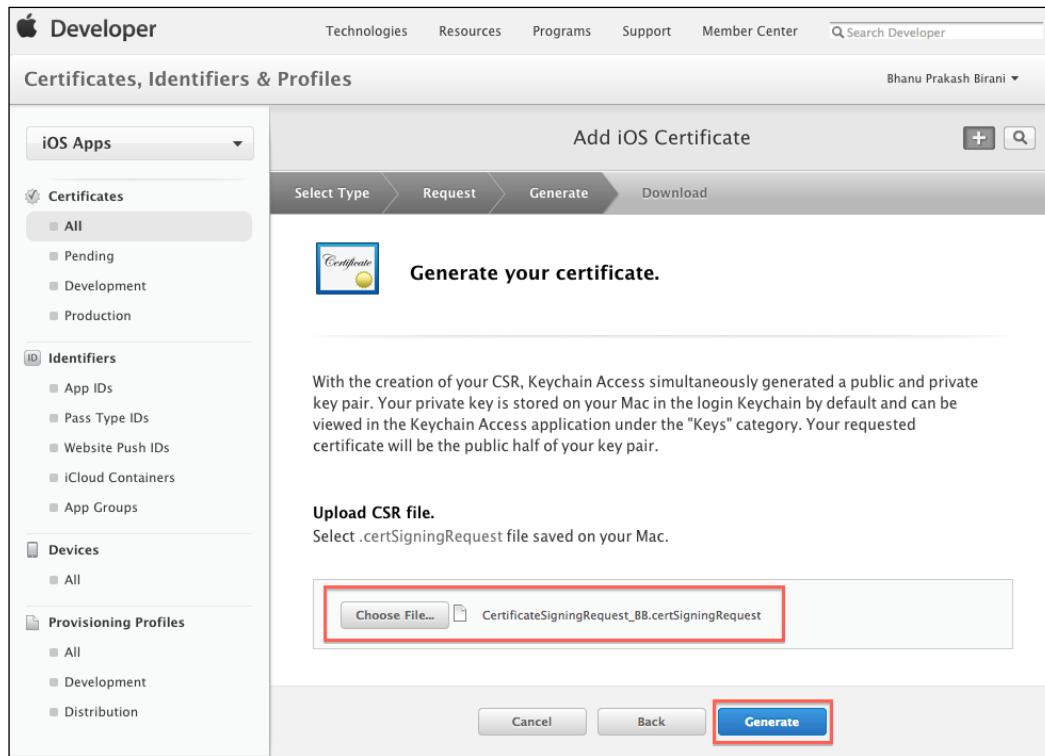


Now, follow the process as specified in the following certificate-generation page and click on **Continue**:



The App-distribution Program

Now, upload the `.certSigningRequest` file you just created using the listed instructions. The page should look something similar to the following screenshot:



Now, clicking on **Generate** will create a development certificate; this might take a few minutes. Once the certificate is created, you will be able to see it in the list of your certificates, as shown here:

The screenshot shows the 'Certificates, Identifiers & Profiles' section of the Apple Developer portal. On the left, there's a sidebar with filters for 'iOS Apps' (selected), 'Certificates' (selected), 'Identifiers', 'Devices', and 'Provisioning Profiles'. The main area displays 'iOS Certificates' with a total of 2 certificates. A table lists the certificate details:

Name	Type	Expires
Bhanu Prakash Birani	iOS Development	Sep 04, 2015

At the top right of the main area are '+', search, and refresh icons.

Click on the certificate to download or revoke. The result is shown in this screenshot:

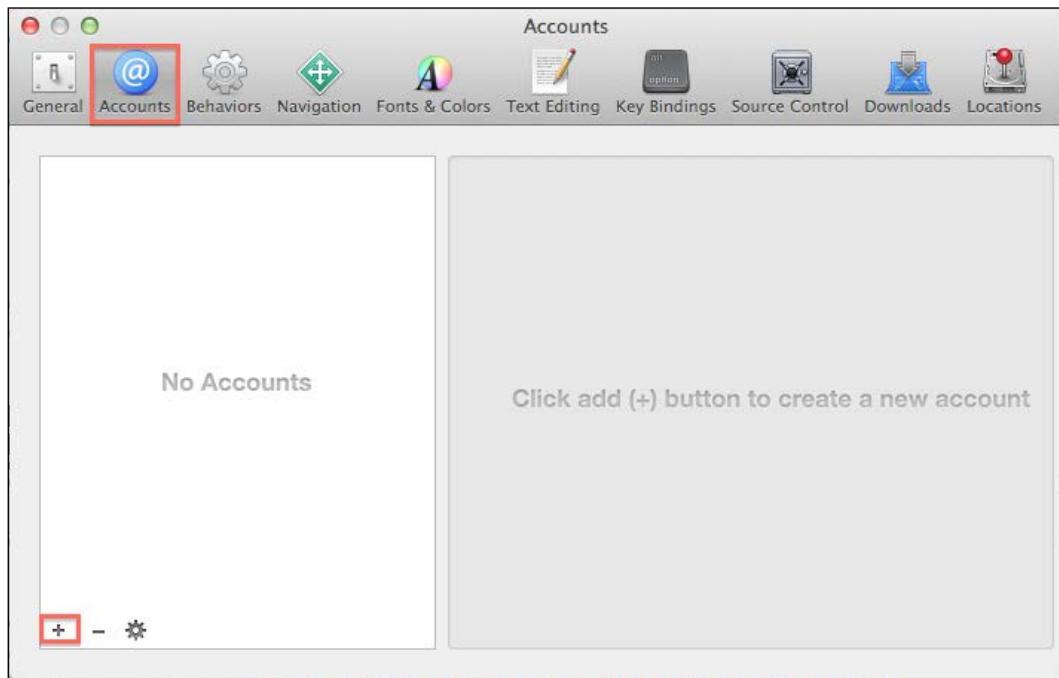
This screenshot shows a detailed view of the selected certificate ('Bhanu Prakash Birani'). It includes the certificate icon, name, type, and expiration date. Below this, the certificate details are listed:

Name	Type	Expires
Bhanu Prakash Birani	iOS Development	Sep 04, 2015

Below the details are 'Revoke' and 'Download' buttons.

The App-distribution Program

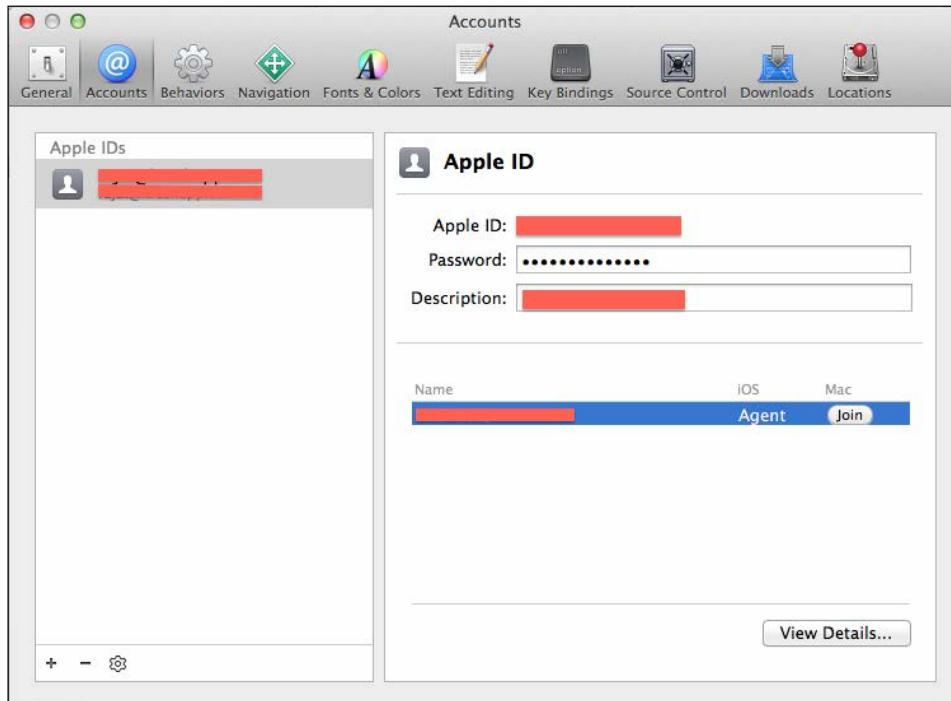
Now, we have successfully created our certificate. It's time to add it in Xcode. Open Xcode and go to **File | Preferences**. Now, click on **Accounts** from the icons at the top of the screen. This should look similar to the following screenshot:



Now, you can see that, currently, there is no account synced with Xcode. So, click on the + button in the bottom-left corner of the screen and add the account we created in the previous steps. The account pop-up should look similar to the one shown in the following screenshot:



After successful login, you will be able to see your account and related profiles in the list, as shown here:



The App-distribution Program

Now, let's move back to the browser in our Apple Developer portal. The next step will be to register our devices in the Developer portal.

Now open the browser and let's continue. The next step is to register our devices. In the left menu, click on **Devices** and select All. Then, on the right-hand side, click on the + button to add devices, as shown in this screenshot:

The screenshot shows the Apple Developer portal interface. The top navigation bar includes links for Technologies, Resources, Programs, Support, Member Center, and a search bar. The main title is "Certificates, Identifiers & Profiles". On the left, a sidebar has sections for Certificates (All, Pending, Development, Production), Identifiers (App IDs, Pass Type IDs, Website Push IDs, iCloud Containers, App Groups), Devices (All), and Provisioning Profiles (All, Development, Distribution). The main content area is titled "iOS Devices" and contains a table with columns "Name" and "UDID". Above the table, it says "You can register devices." A red box highlights the "+" button in the top right corner of the table header.

To add a device, you will need the UDID of the device(s) on which you want to run your apps. You can get the UDID of the device either through Xcode's organizer or iTunes. You can open iTunes and click on **Summary** to get the UDID of the device, as shown here:



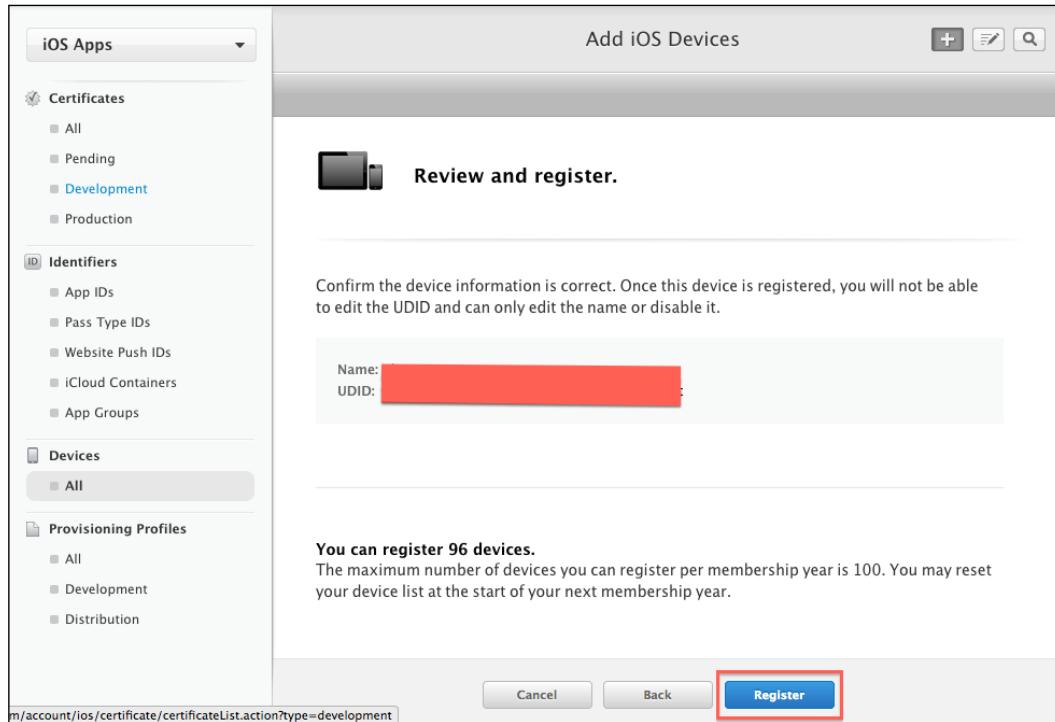
The Summary button

The App-distribution Program

Now, we have all that we need to add the device. Click on the + button and provide the device's UDID along with the device name. After providing the details, click on **Continue** as shown here:

The screenshot shows a software interface titled "Add iOS Devices". At the top right are three buttons: a plus sign (+), a pencil, and a magnifying glass. Below the title is a section titled "Registering a New Device or Multiple Devices" with a small icon of two devices. A "Pre-Release Software Reminder" message follows, stating that users may only share Apple pre-release software with registered employees, contractors, and members of their organization. It also warns against unauthorized distribution, which can lead to termination of the Apple Developer Program and legal liability. Two registration options are presented: "Register Device" and "Register Multiple Devices". The "Register Device" option is selected, showing fields for "Name" and "UDID", both of which are highlighted with a red rectangle. The "Register Multiple Devices" option includes a "Choose File..." button for uploading a file containing multiple device entries. At the bottom, there are "Cancel" and "Continue" buttons, with "Continue" being highlighted by a red rectangle.

Confirm the submission by clicking on **Register** as shown in the following screenshot. You can add up to 100 devices in this list to debug your app.



Now, we have successfully registered the device, so it's time to register an app ID. All the apps that we build have a unique app ID. Click on **App IDs** on the left-hand side of the screen to register your new app ID. Click on the **+** button and fill in all the required details for the app as shown in the following screenshot. You can also read the complete description of the app ID from Apple's help text at the top of the screen.

The screenshot shows the 'Registering an App ID' interface. At the top, there is a large 'ID' button and the title 'Registering an App ID'. Below this, a help text explains that the App ID string contains two parts separated by a period (.)—an App ID Prefix and an App ID Suffix. It also links to 'Learn More'.

App ID Description

Name: You cannot use special characters such as @, &, *, "

App ID Prefix

Value: B42FVLXXQS (Team ID)

App ID Suffix

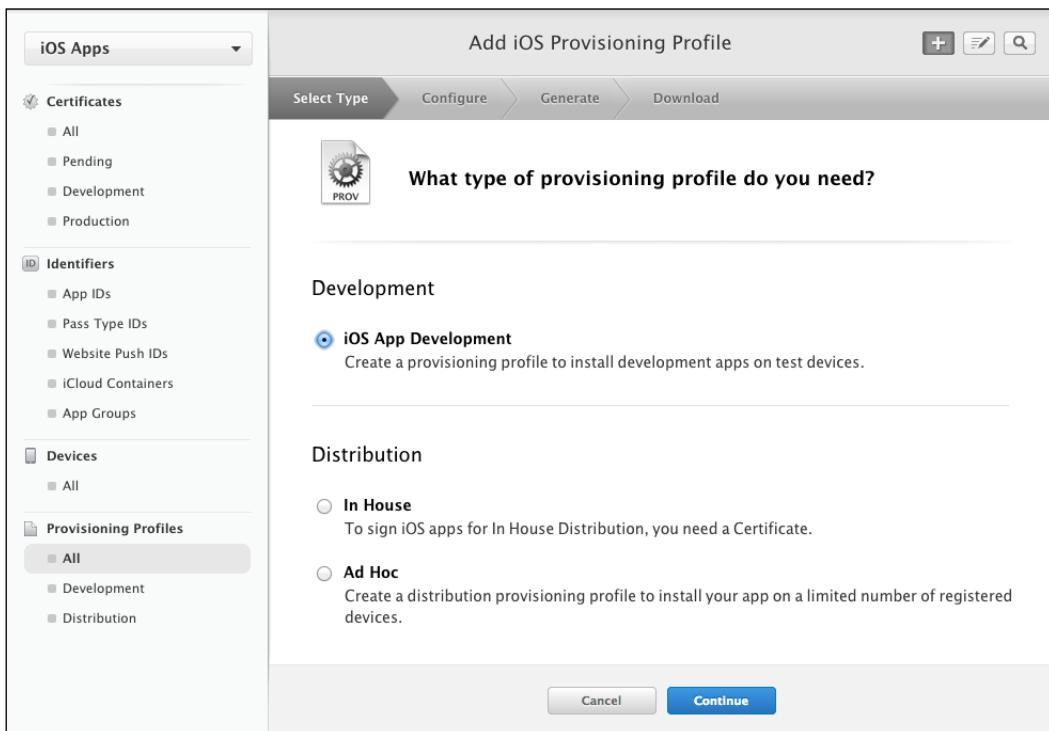
Explicit App ID
If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID:

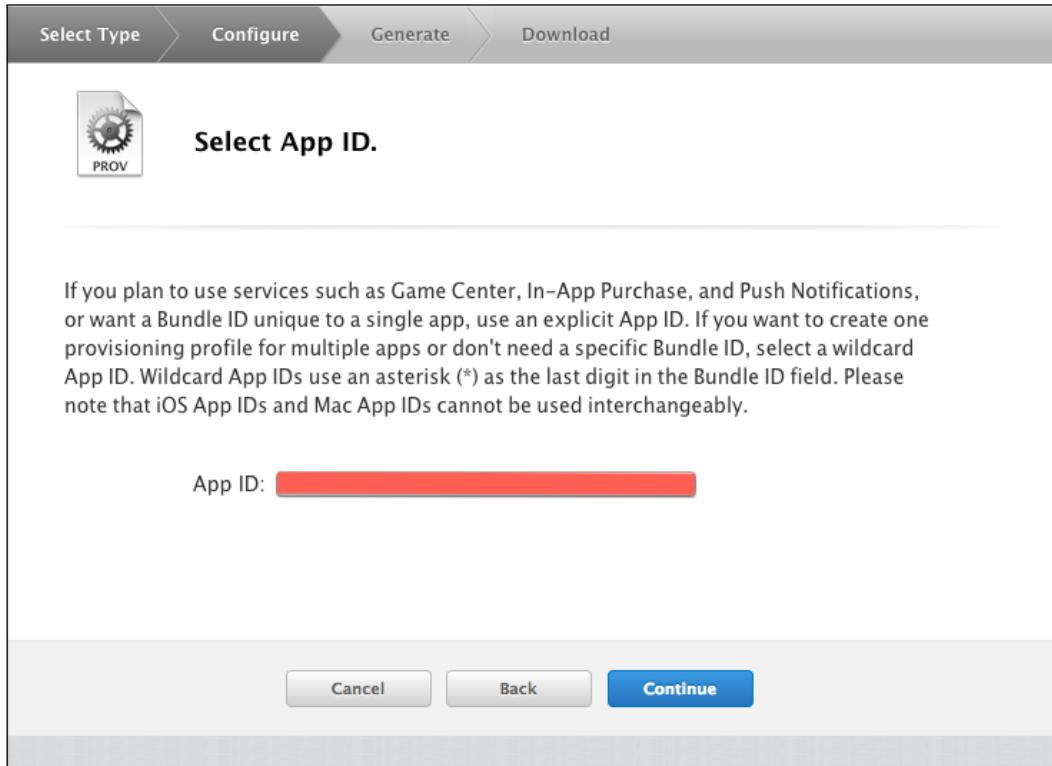
Fill in all the necessary details and click on **Continue**. The app ID will be shown in the list automatically after being successfully created.

Now, it's time for the last step. We will create a provisioning profile that will actually bundle all the steps we followed in the previous sections. The provisioning profile binds the developer's certificate with the app ID to allow the app to run on the device that is permitted in the provisioning profile. Follow the same steps used in creating certificates and identifiers, click on the **Provisioning Profiles** section, and select **All**. Then, click on the **+** button in the top-right corner of the screen. Then, in the form, select **iOS App Development** under the **Development** section. You should see the following screenshot:

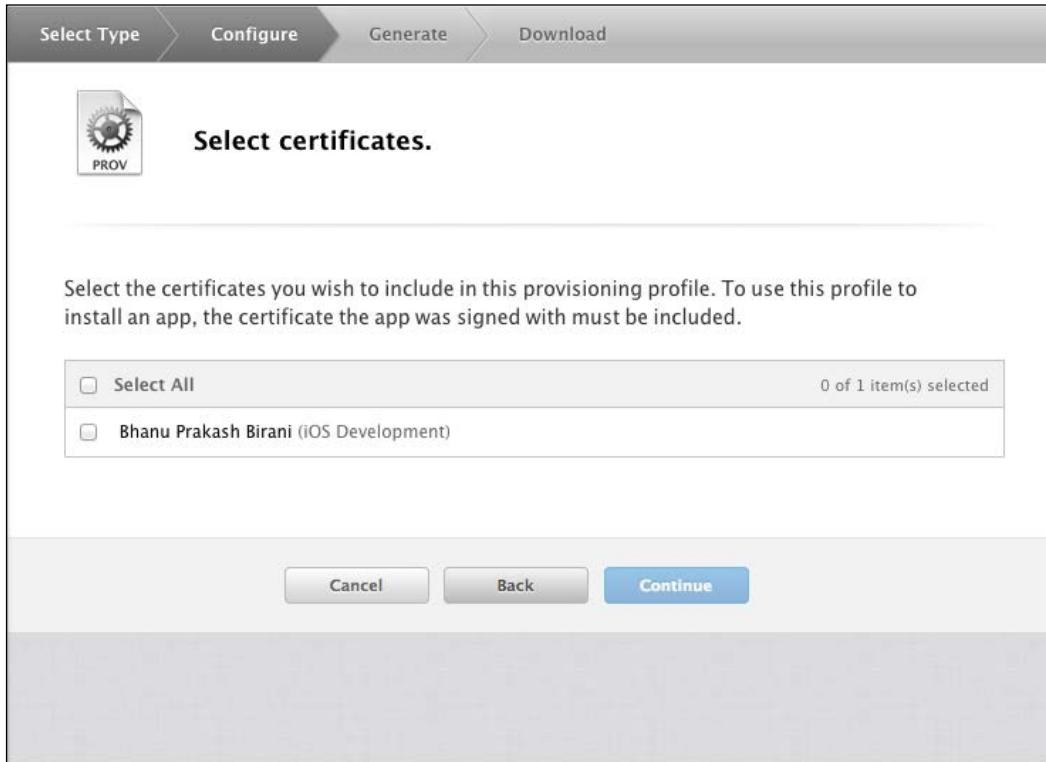


The App-distribution Program

Now, select your app ID that you created in the previous section and click on **Continue**. The selection screen will look similar to the one shown in the following screenshot:

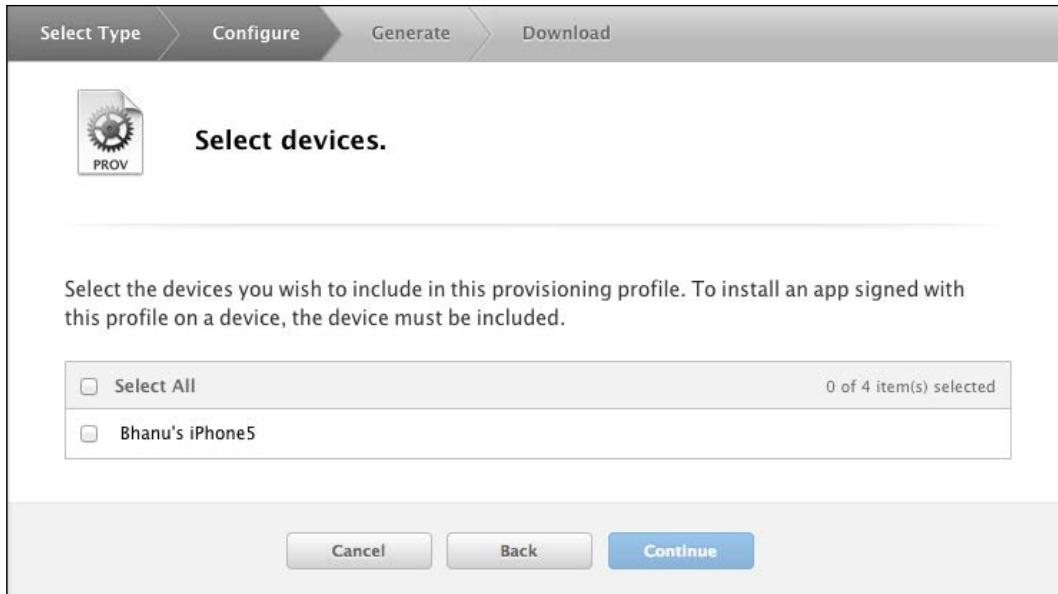


Then, on the next screen, select the list of iOS certificates for which this profile will be available and click on **Continue**. The selection screen should look similar to the one shown here:



The App-distribution Program

Then, we will add the devices for the provisioning profile we created, for which this profile will be available. Then, click on **Continue**. The selection screen should look similar to the one in the following screenshot. Check all the devices you want to run your app on.



Your profile will be created successfully after this process. Now, it's time to create the distribution profile, which will be used to deploy your apps on the App Store. To create the distribution profile, click on the same + button in the **Provisioning Profiles** menu in the left-hand panel.

Now, check the settings for the page, as shown here:

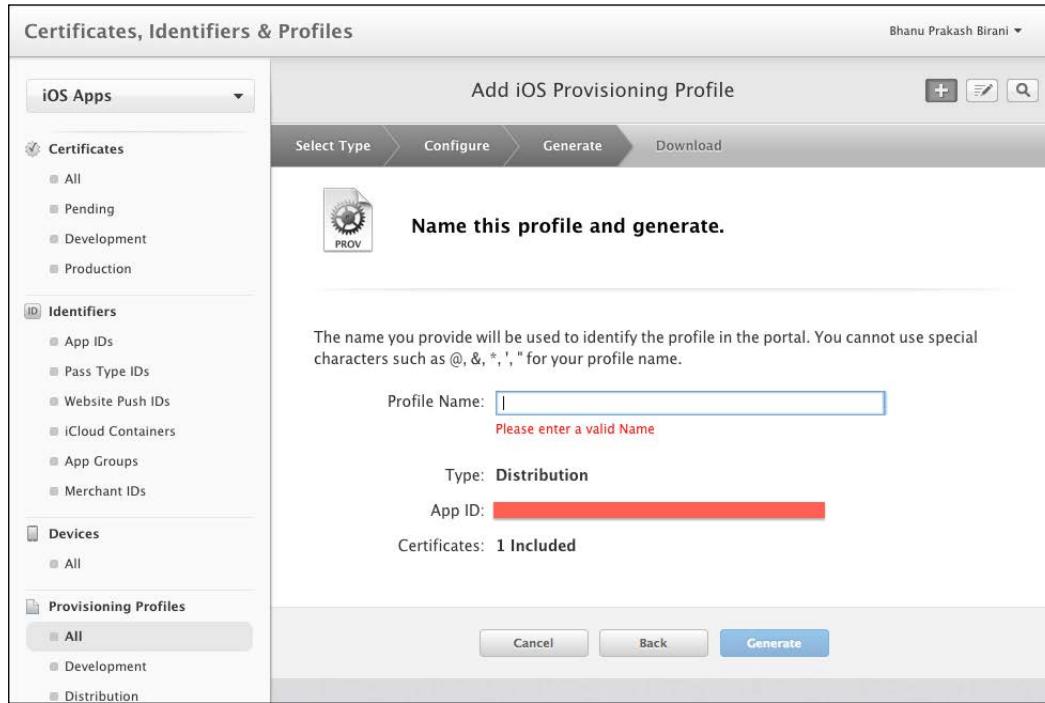
The screenshot shows the 'Certificates, Identifiers & Profiles' section of the Apple Developer portal. On the left, there's a sidebar with a dropdown menu set to 'iOS Apps'. Under 'Certificates', there are four options: All, Pending, Development, and Production. Under 'Identifiers', there are five options: App IDs, Pass Type IDs, Website Push IDs, iCloud Containers, and App Groups. Under 'Devices', there are two options: All and Merchant IDs. Under 'Provisioning Profiles', there are three options: All, Development, and Distribution. The 'Distribution' section is expanded, showing two options: 'App Store' (selected) and 'Ad Hoc'. Below these sections are 'Cancel' and 'Continue' buttons.

To create the distribution profile, we will select **App Store** from the **Distribution** section. Now, click on **Continue** and, on the next page, select the app ID that we created in the previous section. This distribution profile will be bound to that specific selected app ID.

Now, on the next page, select **Certificate** and click on **Continue**.

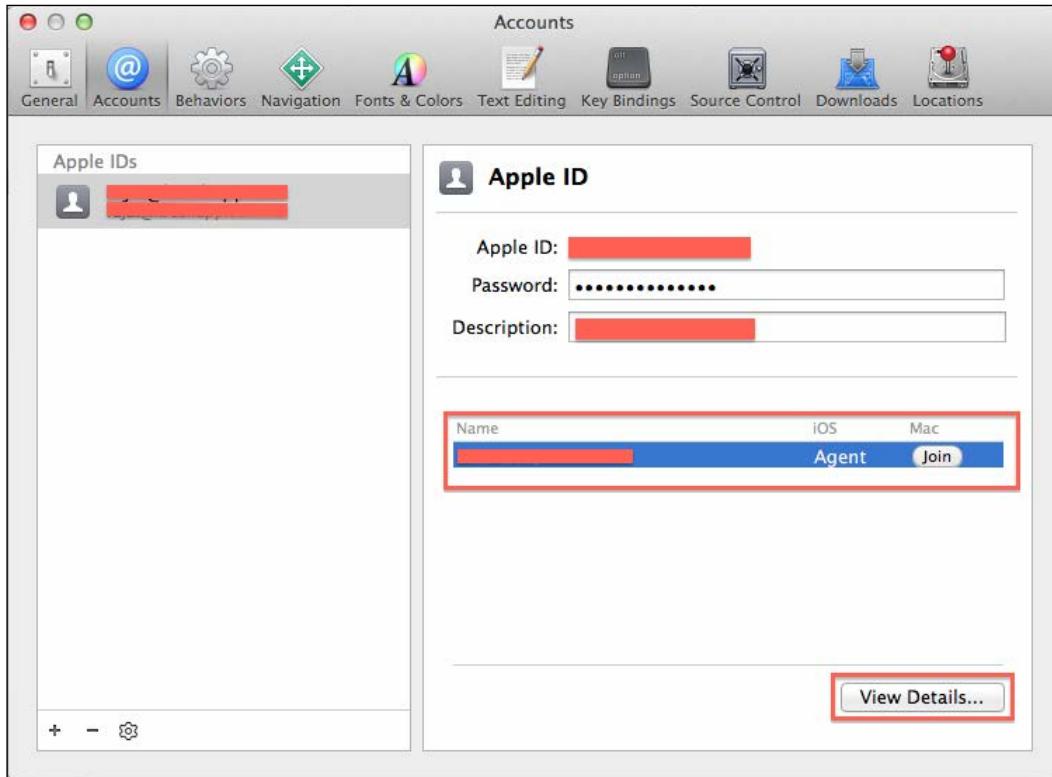
The App-distribution Program

This will make you land on the next page, which will ask for the name of the provisioning profile. Just provide a name for your distribution profile and click on **Generate**, as shown in the next screenshot:



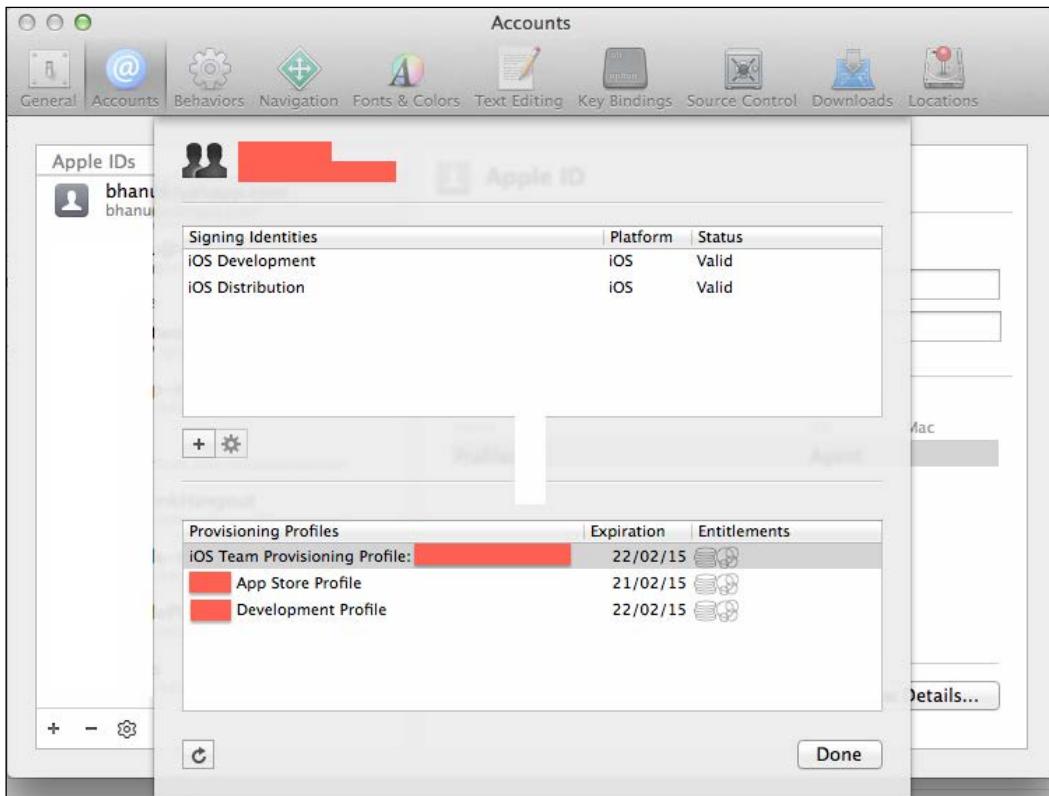
Now, your distribution profile is ready to deploy the apps on the App Store. It's time to link your provisioning profiles and certificate to Xcode. So, open Xcode and go to **File | Preferences**.

Then, click on the account that we added in the previous sections of this chapter. Click on the **View Details...** button, as shown here:



The App-distribution Program

Now, you can see all the provisioning profiles and certificates that we have created in previous sections, in the list (as shown here):



Now, you can test your development provisioning profile by running your app on the device that we have added in the portal. Just connect the device to your system and select it from Xcode once it is detected. Now, you can run your app on the device by clicking on the **Play** button on Xcode.

Awesome! So, we have all that we need to release the app. Now, open your iTunes Connect account and accept all the contacts to start up with the account. It will also ask you for the bank details in to which you want to transfer your profits.

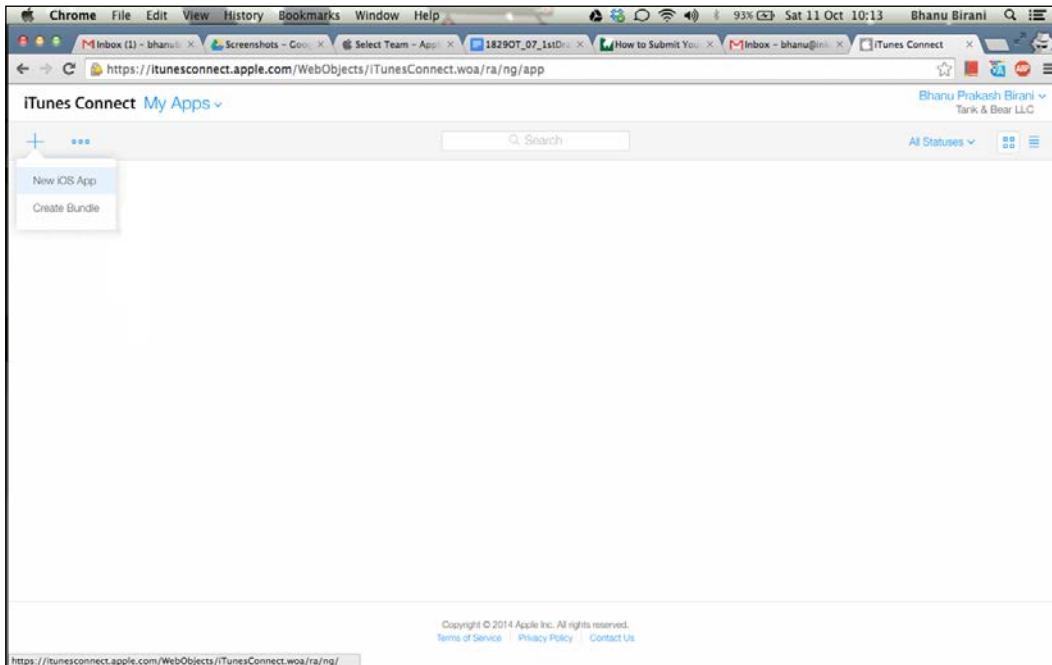
Submitting the app

You need to get the following items in order to submit your app. We need to make sure that we have all of them before we get started:

- The app's name
- The app's description
- The app's icon, sized 512px by 512px

You will need at least one screenshot of the app. The size of the screenshot should be 640 × 920 (retina, no status bar); 640 × 960 (retina); or for landscape, 960 × 640 (retina).

We need to assemble all this and then click on **Manage Your Application** in iTunes Connect. Now, click on the + button in iTunes Connect and the resulting screen is as follows:



The App-distribution Program

Fill in all the required fields of the app as shown in the following screenshot:

The screenshot shows the 'New iOS App' creation form. It includes fields for Name (AntKiller), Version (1.0), Primary Language (English), SKU (com.mb.AntKiller), and Bundle ID Suffix (com.mb.AntKiller). A note at the bottom says 'Register a new bundle ID on the [Developer Portal](#)'. There are 'Cancel' and 'Create' buttons at the bottom.

Select the bundle ID that we created for our app and, similarly, fill in all the required fields. Once you have clicked on **Create**, you will be taken to the next page (as shown in the following screenshot) and asked for more information on the app; this information includes screenshots, videos (if any), a description of your app, and category-related information about your app.

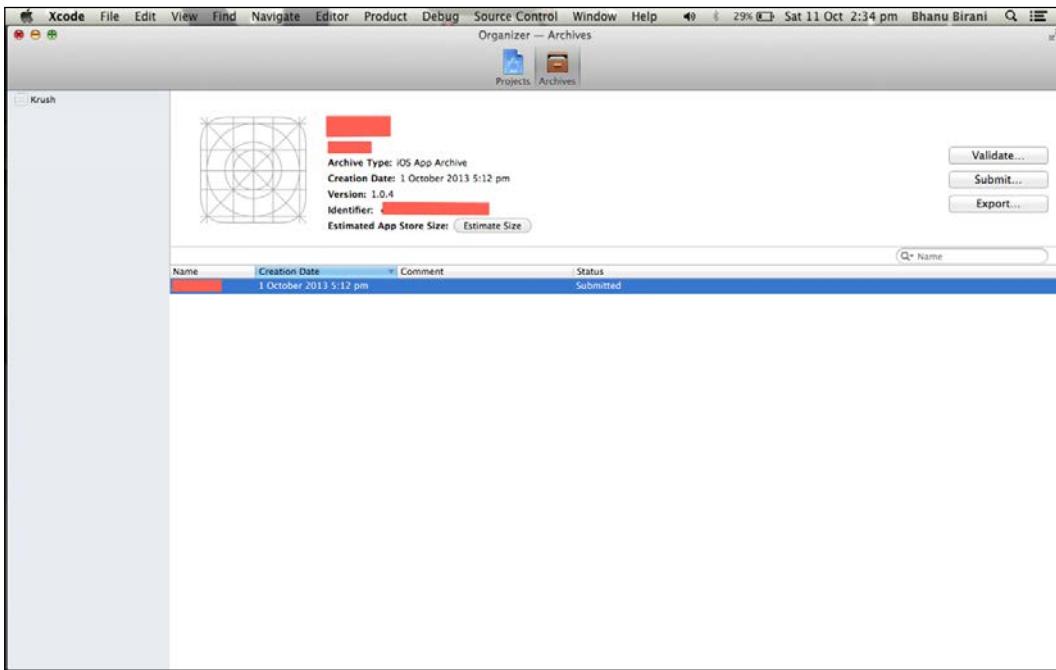
The screenshot shows the 'My Apps' section in iTunes Connect. The 'AntKiller' app is listed with version 1.0 in the 'Prepare for Submission' state. The 'Version Information' tab is selected, showing options for 4.7-Inch, 5.5-Inch, 4-Inch, 3.5-Inch, and iPad. A large text area is available for dragging screenshots or app video previews. A note at the bottom states: 'To upload your app video preview, you must use Safari and OS X v10.10 or later.' There are 'Save' and 'Submit for Review' buttons at the top right.

You can select and update the details in all the tabs shown in the preceding screenshot and click on **Save** once the complete information is provided.

Submitting with Xcode

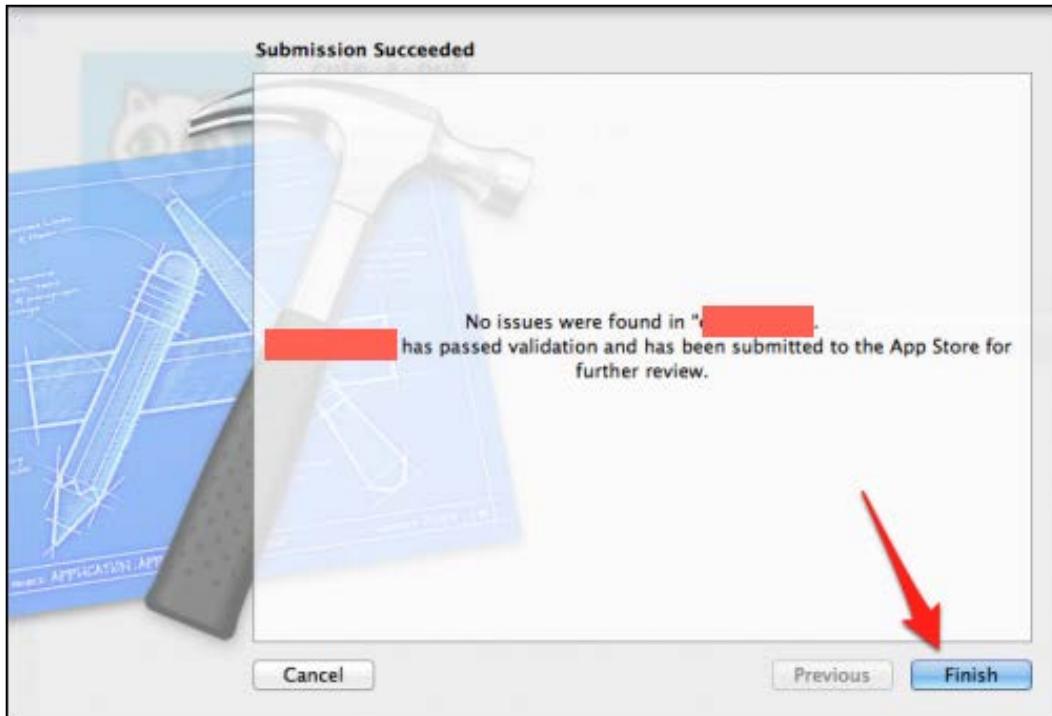
Now, it's time to submit the app using Xcode, so open Xcode and go to **Product | Archive**.

Once the archive is finished, click on **Validate**. This will check the validation of your binary and list all the issues, if any; otherwise, it will say that validation was successful. Now, click on the **Submit** button to submit the application on the App Store, as shown in the following screenshot. This will ask you to select **iOS Distribution profile** to upload the binary for the App Store. Select the profile that we have created in a previous section and click on **Submit**. Now, your app will start uploading to the Apple Store.

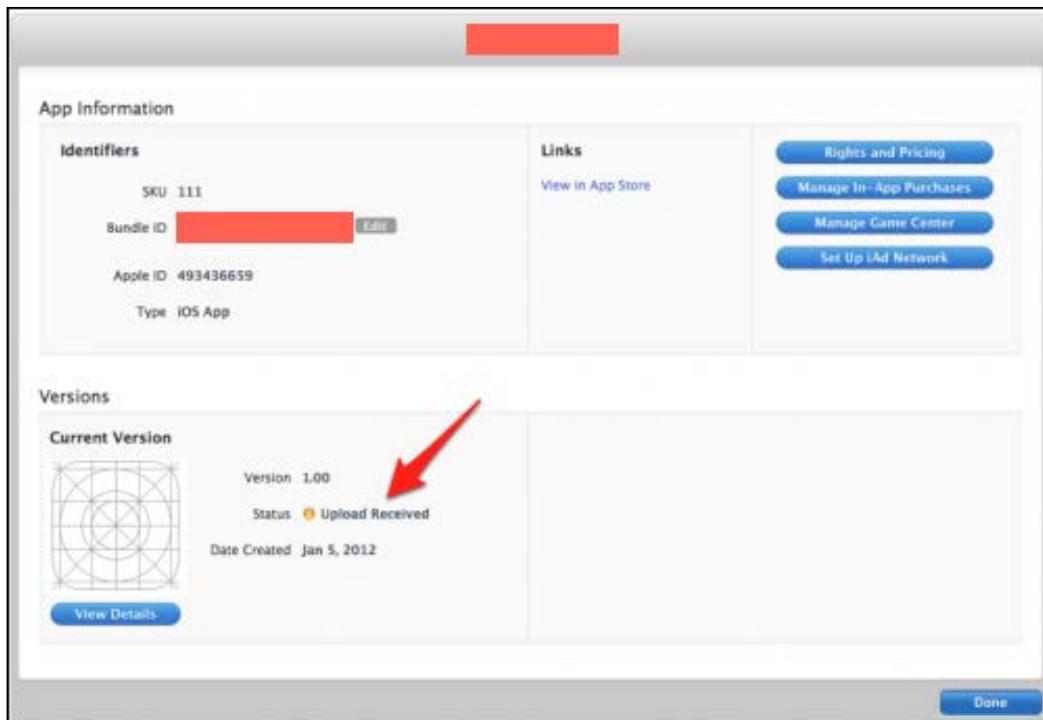


The App-distribution Program

You will get a success message once the binary is uploaded successfully. The message will be something similar to the one shown in the following screenshot:



Click on **Finish**. Now, you can open your iTunes Connect account and check whether your binary has been received. You can also see the **Upload Received** message that will change into **Waiting For Review** in a couple of minutes.



The Upload Received message

Congratulations! You have successfully submitted your app to the App Store!

Summary

In this chapter, you learned how to set up and understand your developer account. You also learned how to set up a provisioning profile to publish your app on the Store, and we discussed the anatomy of Enterprise-level distribution using the Apple Enterprise account.

Index

Symbols

@property method 11
@property method, attributes
 assign 12
 atomic 12
 copy 12
 nonatomic 12
 retain 12
 strong 12
 weak 12
@synthesize method 12

A

Accelerate.framework 50
Access Control Lists (ACLs) 148
Accounts.framework 50
activity, Handoff
 continuing 119, 120
 type, specifying 117, 118
activity, Keychain Services
 starting 137-146
activity object, Handoff
 user info dictionary, populating 119
AddressBook.framework 50
AddressBookUI.framework 50
AdSupport.framework 50
Adventure Game 84
AirDrop 78-83
AntKilling game
 building 85-87
API 77
App
 submitting 191, 192
AppKit 116

Apple Development Center 84
application programming interface. *See API*
array
 about 11
 NSArray 11
 NSMutableArray 11
AssetsLibrary.framework 50
AudioToolbox.framework 50
AudioUnit.framework 50
AVFoundation.framework 50

B

BOOL, datatype 10

C

certificates
 about 170-190
 creating 172-174
 device, adding 179
 downloading 175, 176
CFNetwork.framework 50
CGFloat, datatype 10
continuation streams 123, 124
CoreAudio.framework 50
CoreBluetooth.framework 50
core data
 about 63-70
 managed object context 64
 NSManagedObjectModel 64
 persistent store coordinator 64
CoreData.framework 50
core data storage, iCloud 126
CoreFoundation.framework 50
CoreGraphics.framework 50
CoreImage.framework 50

CoreLocation.framework 51
CoreMedia.framework 51
CoreMIDI.framework 51
CoreMotion.framework 51
CoreTelephony.framework 51
Core Text 97
CoreText.framework 51
CoreVideo.framework 51

D

database integration
about 58
SQLite 59-63
datatypes, iOS
BOOL 10
CGFloat 10
NSInteger 10
NSString 10
delegates 13, 14
developer account
setting up 151-157
development profiles 169

E

EventKit.framework 51
EventKitUI.framework 51
exclusion paths 101
ExternalAccessory.framework 51

F

Facebook SDK
URL 75
Foundation.framework 51

G

GameController.framework 51
GameKit.framework 51
GLKit.framework 51
GSS.framework 51

H

Handoff
about 115
App framework support 116
compatibility with 115, 116
for seamlessly resuming activities 115
interactions 116
Native App-to-Web Browser Handoff 121
Web Browser-to-Native App Handoff 122
Handoff, implementing
about 117
activity, continuing 119, 120
activity object's user info dictionary, populating 119
activity type, specifying 117, 118
in responders, adopting 119
user activity object, creating 117
HUD (Heads-Up Display) layer 84

I

iAd.framework 51
iCloud
about 125
document storage 126
key-value storage 125
working 126-135
IDE (Integrated Development Environment) 7
identifiers. *See certificates*
ImageIO.framework 51
image view
about 36
using 36, 37
implementation class
code snippets 9
interface class
code snippets 9
IOKit.framework 51
iOS
methods, types 9, 10
iOS Dev Center
URL 152, 167
iOS Developer Account 166, 167
iOS Developer Program
joining 158-165

iOS native game framework,
 SpriteKit 84-97

iOS Provisioning Portal

 about 168
 development profiles 169
 distribution profiles 169

iPhone app

 about 15-24
 Debug area 19
 Editor area 18
 Navigation area 18
 Utility area 19

iTunes Connect 168 169

J

JavaScriptCore 51

justification, Text Kit 100-106

K

kerning, Text Kit 99

Keychain Access

 Touch ID through 148

Keychain Services

 about 135
 activity, starting 137-146
 application flow 136
 decryption 136
 encryption 136
 iOS keychain, concepts 136
 iOS keychain, structure 136
 SecItemAdd API 135
 SecItemCopyMatching API 135
 SecItemUpdate API 135

key-value storage, iCloud 125

L

label

 adding 27

libraries

 UI components with 53-58

ligatures, Text Kit 99

line breaking, Text Kit 100

Local Authentication framework, Touch ID

 using 148, 149

M

MapKit.framework 51

map view 29-31

MediaAccessibility.framework 52

MediaPlayer.framework 52

MediaToolbox.framework 52

MessageUI.framework 52

methods, iOS

 types 9, 10

MobileCoreServices.framework 52

model objects

 Asset collections 108

 Assets 108

 Collection lists 108

MultipipeConnectivity.framework 52

N

Native App-to-Web Browser Handoff 121

navigation controller 44-47

NewsstandKit.framework 52

NSArray 11

NSInteger, datatype 10

NSLayoutManager class, Text Kit 98

NSManagedObjectModel 64

NSMutableArray 11

NSString, datatype 10

NSTextContainer class, Text Kit 98

NSTextStorage class, Text Kit 98

O

Objective-C

 about 9

 implementation file 9

 interface file 9

objects

 creating 10

OOP (object-oriented programming) 7

OpenAL.framework 52

OpenGL ES.framework 52

P

PassKit.framework 52

PHAdjustmentData class 110

PHAssetChangeRequest class 110

PHAsset class 111
PHAssetCollectionChangeRequest class 110
PHAssetCollection class 111
PHCachingImageManager class 111
PHChange class 110
PHCollection class 111
PHCollectionListChangeRequest class 110
PHCollectionList class 111
PHContentEditingInput class 110
PHContentEditingInputRequestOptions class 110
PHContentEditingOutput class 110
PHFetchOptions class 110
PHFetchResultChangeDetails class 111
PHFetchResult class 110, 112
PHImageManager class 111
PHImageRequestOptions class 111
PHObjectChangeDetails class 111
PHObject class 111
PHObjectPlaceholder class 111
PhotoKit
 about 107
 features 109, 110
 frameworks 107
 Photos Framework 107
 Photos UI 107
PhotoKit, classes
 PHAdjustmentData 110
 PHAsset 111
 PHAssetChangeRequest 110
 PHAssetCollection 111
 PHAssetCollectionChangeRequest 110
 PHCachingImageManager 111
 PHChange 110
 PHCollection 111
 PHCollectionList 111
 PHCollectionListChangeRequest 110
 PHContentEditingInput 110
 PHContentEditingInputRequestOptions 110
 PHContentEditingOutput 110
 PHFetchOptions 110
 PHFetchResult 110
 PHFetchResultChangeDetails 111
 PHImageManager 111
 PHImageRequestOptions 111
 PHObject 111
 PHObjectChangeDetails 111
 PHObjectPlaceholder 111
 PHPphotoLibrary 111
 PHVideoRequestOptions 111
Photos Framework
 about 107
 Asset collections 108
 Assets 108
 Collection lists 108
 Transient collections 108
Photos UI 108, 109
PHPphotoLibrary class 111
PHVideoRequestOptions class 111
profiles. See **certificates**
properties 11

Q

QuickLook.framework 52

S

SafariServices.framework 52
scroll view 42-44
SecItemAdd API 135
SecItemCopyMatching API 135
SecItemUpdate API 135
Security.framework 52
Segue 44
SLComposeViewController class 71
Social.framework 52 71
social integration 71-76
SpriteKit
 about 84
 iOS native game framework 84, 88-97
SpriteKit.framework 52
SQLite
 about 59-63
 sqlite3_close() function 59
 sqlite3_exec() function 59
 sqlite3_finalize() function 59
 sqlite3_open() function 59
 sqlite3_prepare_v2() function 59
 sqlite3_step() function 59
StoreKit.framework 52
Swift
 URL 112
SystemConfiguration.framework 52

T

table view
about 38
methods 38
working with 39-41

Text Kit
about 97
justification 100-106
kerning 99
ligatures 99
line breaking 100
NSLayoutManager class 98
NSTextContainer class 98
NSTextStorage class 98

touch authentication
Touch ID through 147

Touch ID
Local Authentication framework
used 148, 149
through Keychain Access 148
through touch authentication 147

Touch ID API 146, 147

Touch ID authentication 146, 147

Transient collections 108

Twitter.framework 52

U

UIActivityViewController class 78

UI components
about 29
image view 36, 37
label, adding 27
map view 29-31
new button, adding 28
table view 38

UIPickerView 32-34
view, adding 26
web view 34, 35
with libraries 53-58

UI Elements 25

UIKit framework 25, 116

UIKit.framework 53

UIPickerView 32-34

user activity object, Handoff
creating 117

V

VideoToolbox.framework 53

view
adding 26

W

Web Browser-to-Native App
Handoff 122, 123

web view 34, 35

Worldwide Developers Conference (WWDC) 168

X

Xcode
about 25
submitting with 193-195

Xcode 6
URL 151



**Thank you for buying
Learning iOS 8 for Enterprise**

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.

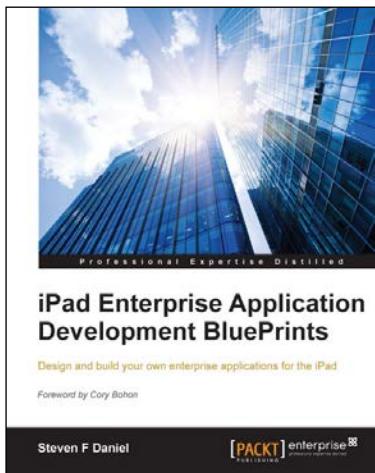
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books 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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at 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, then please 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.

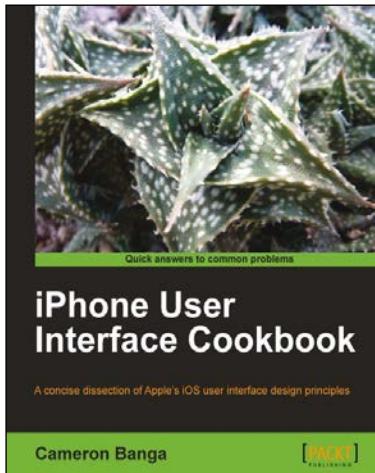


iPad Enterprise Application Development BluePrints

ISBN: 978-1-84968-294-7 Paperback: 430 pages

Design and build your own enterprise applications for the iPad

1. Learn how to go about developing some simple yet powerful applications with ease.
2. Each chapter explains about the technology in-depth, while providing you with enough information and examples to help grasp the technology.
3. Get to grips with integrating Facebook, iCloud, Twitter, and AirPlay into your applications.



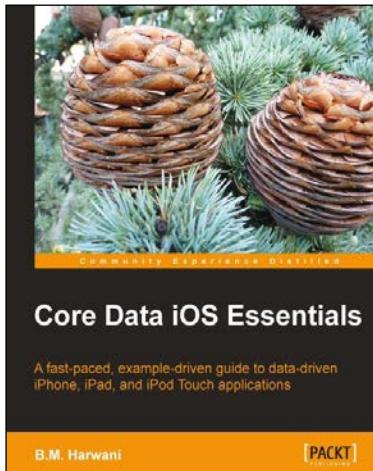
iPhone User Interface Cookbook

ISBN: 978-1-84969-114-7 Paperback: 262 pages

A concise dissection of Apple's iOS user interface design principles

1. Learn how to build an intuitive interface for your future iOS application.
2. Avoid app rejection with detailed insight into how to best abide by Apple's interface guidelines.
3. Written for designers new to iOS who may be unfamiliar with Objective-C or coding an interface.
4. Chapters cover a variety of subjects, from standard interface elements to optimizing custom game interfaces.

Please check www.PacktPub.com for information on our titles



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, lightweight 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.



iOS 7 Game Development

ISBN: 978-1-78355-157-6 Paperback: 120 pages

Develop powerful, engaging games with ready-to-use utilities from Sprite Kit

1. Pen your own endless runner game using Apple's new Sprite Kit framework.
2. Enhance your user experience with easy-to-use animations and particle effects using Xcode 5.
3. Utilize particle systems and create custom particle effects.

Please check www.PacktPub.com for information on our titles