



Quick answers to common problems

iOS Development with Xamarin Cookbook

Over 100 exciting recipes to help you develop iOS applications with Xamarin

Dimitris Tavlikos

[PACKT]
PUBLISHING

iOS Development with Xamarin Cookbook

Over 100 exciting recipes to help you develop iOS applications with Xamarin

Dimitris Tavlikos

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

iOS Development with Xamarin Cookbook

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 2011

Second edition: May 2014

Production reference: 1160514

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84969-892-4

www.packtpub.com

Cover image by Kelly Gibson (gibsonkelly36@yahoo.com)

Credits

Author

Dimitris Tavlikos

Project Coordinator

Amey Sawant

Reviewers

Ryan Alford

Yaroslav Bigus

William Smith

Proofreaders

Simran Bhogal

Bridget Braund

Lauren Harkins

Acquisition Editors

Joanne Fitzpatrick

Usha Iyer

Indexer

Mariammal Chettiyar

Content Development Editor

Amit Ghodake

Production Coordinators

Aparna Bhagat

Arvindkumar Gupta

Saiprasad Kadam

Nilesh R. Mohite

Aditi Gajjar Patel

Technical Editors

Neha Mankare

Humera Shaikh

Faisal Siddiqui

Cover Work

Nilesh R. Mohite

Copy Editors

Dipti Kapadia

Sayanee Mukherjee

Deepa Nambiar

Karuna Narayanan

Stuti Srivastava

Laxmi Subramanian

About the Author

Dimitris Tavlikos is a freelance software developer living in Greece. With over 10 years of professional experience as a programmer, he specializes in mobile development with clients all over the world. Dimitris has a passion for programming, and has recently been awarded the Xamarin MVP designation for his work. He has written a book on iOS development and various articles on his blog.

About the Reviewers

Ryan Alford is a .NET software engineer who works from home. Ryan has been a .NET developer for over 7 years, with the majority of his focus being on C#. In his early years, he worked almost exclusively on WinForms and Windows Mobile. He then started working with ASP.Net, AJAX, and Silverlight. In the past few years, as mobile development really started to take off, he took an interest in Xamarin and MonoTouch.

Ryan was able to help convince the management at his employer to use Xamarin for their upcoming enterprise application on iOS, as the company was using .Net and C# in other projects. It was at this point that Ryan was added to the three-person development team to write the new iOS enterprise application.

Ryan has written and released two Android applications: MotoTorch LED and Phase 10 Score Center. MotoTorch LED has more than 500,000 downloads and was one of the first applications on Android that used the camera LEDs as a flashlight.

Today, Ryan is currently rewriting Phase 10 Score Center in Xamarin.Android to ease the development of new features. He is still on his iOS team and continues to add new features to his company's enterprise application.

Yaroslav Bigus is an expert in building cross-platform web and mobile applications. He has over 4 years experience in development and has worked for companies in Leeds and New York. He has been using the .NET Framework stack for developing backend systems, JavaScript for the frontend side, and Xamarin for mobile devices.

He is now working for an Israeli startup called yRuler. Previously, Yaroslav reviewed *Xamarin Mobile Application Development for iOS*, Paul F. Johnson, Packt Publishing.

I am thankful to my family and friends.

William Smith has been developing with Xamarin Studio for over 3 years and has been developing software since 2001. He currently works as a Geospatial Developer at Geographic Information Services, Inc., specializing in mobile-platform development. He is also the founder of Websmiths, LLC (www.websmithsllc.com), a consulting firm that offers services in cross-platform mobile application development and web development. William holds two BSc degrees in Computer Science and Business Administration from the University of Maryland.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



<http://PacktLib.PacktPub.com>

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: Development Tools	7
Introduction	7
Installing prerequisites	8
Creating an iOS project with Xamarin Studio	13
Interface Builder	23
Creating the UI	26
Accessing the UI with Outlets	29
Adding Actions to controls	34
Compiling an iOS project	36
Debugging our application	39
Chapter 2: User Interface – Views	43
Introduction	43
Adding and customizing views	44
Receiving user input with buttons	48
Displaying images	53
Displaying and editing text	57
Using the keyboard	60
Displaying progress	64
Displaying content larger than the screen	67
Navigating through the content divided into pages	70
Displaying alerts	74
Creating a custom view	78
Styling views	81
Chapter 3: User Interface – View Controllers	85
Introduction	85
Loading a view with a view controller	86
Navigating through different view controllers	88

Providing controllers in tabs	91
Modal view controllers	94
Creating a custom view controller	96
Using view controllers efficiently	98
iPad view controllers	100
UI flow design with storyboards	105
Unwinding in storyboards	109
Chapter 4: Data Management	113
Introduction	113
Creating files	113
Using an SQLite database	116
Preparing for iCloud support	121
iCloud key/value storage	122
Chapter 5: Displaying Data	127
Introduction	127
Providing lists	128
Displaying data in a table	132
Customizing rows	136
Editing a table	140
Table indexing	143
Searching through the data	145
Creating a simple web browser	149
Displaying data in a grid	151
Customizing the grid	155
Chapter 6: Web Services	159
Introduction	159
Consuming web services	159
Consuming REST services	163
Communicating with native APIs	165
Using WCF services	168
Chapter 7: Multimedia Resources	173
Introduction	173
Selecting images and videos	174
Capturing media with the camera	177
Playing videos	180
Playing music and sounds	183
Recording with the microphone	185
Managing album items directly	189

Chapter 8: Integrating iOS Features	193
Introduction	193
Starting phone calls	194
Sending text messages and e-mails	196
Using text messaging in our application	199
Using e-mail messaging in our application	202
Managing the address book	205
Displaying contacts	209
Managing the calendar	212
Chapter 9: Interacting with Device Hardware	217
Introduction	217
Detecting the device orientation	218
Adjusting the UI orientation	220
Proximity sensor	224
Retrieving the battery information	226
Handling motion events	228
Handling touch events	230
Recognizing gestures	233
Custom gestures	236
Using the accelerometer	239
Using the gyroscope	242
Chapter 10: Location Services and Maps	247
Introduction	247
Determining location	248
Determining heading	252
Using region monitoring	255
Using a significant-change location service	258
Location services in the background	260
Displaying maps	263
Geocoding	266
Adding map annotations	270
Adding map overlays	274
Chapter 11: Graphics and Animation	279
Introduction	279
Animating views	280
Transforming views	282
Animating images	284
Animating layers	286
Drawing lines and curves	290
Drawing shapes	293

Drawing text	295
A simple drawing app	297
Creating an image context	301
Chapter 12: Multitasking	305
Introduction	305
Detecting application states	306
Receiving notifications for app states	308
Running code in the background	310
Playing audio in the background	313
Updating data in the background	315
Chapter 13: Localization	319
Introduction	319
Creating an app for different languages	319
Localizable resources	323
Regional formatting	325
Chapter 14: Deploying	329
Introduction	329
Creating profiles	329
Creating an ad hoc distribution bundle	335
Preparing an app for the App Store	337
Submitting an app to the App Store	340
Chapter 15: Advanced Features	343
Introduction	343
Reproducing the page curl effect	344
Integrating content sharing	348
Implementing custom transitions	353
Using physics in UI elements	358
Implementing the text-to-speech feature	360
Index	363

Preface

This book will provide you with all the necessary skills to develop and deploy rich and powerful applications for the iPhone and iPad, with the C# programming language. Xamarin.iOS, formerly known as MonoTouch, is already established as a powerful software development kit that brings iOS development to .NET programmers. Packed with easy-to-understand and detailed examples, this book will be your best companion in your iOS development journey.

What this book covers

Chapter 1, Development Tools, teaches you how to install and use the development tools necessary to create your first iOS app. From there, you will create and debug your first Xamarin.iOS project.

Chapter 2, User Interface – Views, discusses the essential User Interface components of the iOS SDK. Covering the most commonly used views and controls and many more in detail, we will get familiar with the platform through a number of example projects. We will also discuss the similarities and differences with standard .NET components.

Chapter 3, User Interface – View Controllers, introduces you to the view controllers, the objects that are responsible for providing the interaction mechanism between your app and the user. Explained with simple step-by-step processes, you will start creating complete apps that can run on both the iPhone and iPad devices.

Chapter 4, Data Management, covers data management practices available on the iOS platform and how to use them efficiently with the convenience of C#. You will learn to manage locale SQLite database files, but also work on using iCloud to store data across different devices.

Chapter 5, Displaying Data, focuses on another important part of data management. Through a series of simple and complete projects, you will learn about the available components to display data on the screen of the iPhone, which are smaller than computer screens. Displaying various types of data in a user-friendly manner is essential for mobile devices, and by the time you finish reading this chapter, you will certainly be more skillful in this area.

Chapter 6, Web Services, guides you through .NET SOAP, WCF, and REST services for creating apps that connect the user to the world. These powerful .NET features would not have been part of iOS development without Xamarin.iOS.

Chapter 7, Multimedia Resources, will teach you to create applications that capture, reproduce, and manage multimedia content through the device's hardware. You will not only learn to use the camera to capture images and video, but also learn how to play back and record audio.

Chapter 8, Integrating iOS Features, will walk you through the ways to incorporate the platform's native applications and components. You will learn how to provide e-mail, text messaging, and address book features in your application and how to use the native calendar to create events.

Chapter 9, Interacting with Device Hardware, discusses creating applications that are fully aware of their surrounding environment through the device's sensors. You will learn to adjust the User Interface according to device orientations and respond to accelerometer and gyroscope events.

Chapter 10, Location Services and Maps, is a detailed guide for using the built-in location services to create applications that provide location information to the user. You will not only learn to use the GPS hardware, but also how to display and layout information on maps.

Chapter 11, Graphics and Animation, introduces 2D graphics and animation. You will learn to animate components and draw simple graphics on the screen. By the end of this chapter, you will create a small finger-drawing application.

Chapter 12, Multitasking, will walk you through the details of implementing multitasking in iOS applications. This dramatically enhances the user experience by executing code behind the scenes.

Chapter 13, Localization, discusses how to provide localized content in applications. You will learn how to prepare your application to target users worldwide.

Chapter 14, Deploying, will not only walk you through the required steps to deploy your finished application to devices, but also to prepare and distribute it to the App Store.

Chapter 15, Advanced Features, introduces some of the key features introduced in newer iOS versions, such as implementing physics to User Interface components through the power of iOS 7's UIKit Dynamics, customizing animated transitions between view controllers, and more!

What you need for this book

The minimum requirement for this book is a Mac computer running at least Mac OS X Lion (10.7.*). Almost all projects you will create with the help of this book work on iOS Simulator. However, some projects will require a device to work properly. You will find all the appropriate details in *Chapter 1, Development Tools*.

Who this book is for

This book is essential for C# and .NET developers with no previous experience in iOS development, but it is also for Objective-C developers who want to make a transition to the benefits of Xamarin.iOS and C# language to create complete, compelling iPhone, iPod, and iPad applications and deploy them to the App Store.

Conventions

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

Code words in text, cookbook names, recipe names, scripts, database table names, folder names, filenames, file extensions, and pathnames are shown as follows: "The `Register` attribute is used to expose classes to the underlying Objective-C runtime."

A block of code is set as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using MonoTouch.Foundation;
using MonoTouch.UIKit;
```

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

```
EKEvent newEvent = EKEvent.FromStore(evStore);
newEvent.StartDate = DateTime.Now.AddDays(1);
newEvent.EndDate = DateTime.Now.AddDays(1.1);
newEvent.Title = "Xamarin event!";
```

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

```
cd <code_directory>/CH06_code/WcfService/WcfService
./start_wcfservice.sh
```


New terms and **important words** are shown in bold. Words you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "Go to the **Library** pane and select **Objects** from the drop-down list."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

Piracy of 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 we can pursue a remedy.

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

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

Questions

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

1

Development Tools

In this chapter, we will cover:

- ▶ Installing prerequisites
- ▶ Creating an iOS project with Xamarin Studio
- ▶ Interface Builder
- ▶ Creating the UI
- ▶ Accessing the UI with Outlets
- ▶ Adding Actions to controls
- ▶ Compiling an iOS project
- ▶ Debugging our application

Introduction

One of the most important things professionals care about is the tools that are required to complete their work with. Just like carpenters need a chisel to scrape wood, or photographers need a camera to capture light, we developers need certain tools which we cannot work without.

In this chapter, we will provide information on what **IDEs (Integrated Development Environments)** and **SDKs (Software Development Kits)** are needed to develop applications for iOS, Apple's operating system, for the company's mobile devices. We will describe what the role of every tool in the development cycle is, and go through the features that are essential to develop our first application.

The following are the tools needed to develop applications with Xamarin.iOS:

- ▶ **An Apple Mac computer running at least the Lion (10.7.*) operating system:**
The essential programs we need cannot be installed on other computer platforms.



Xamarin also offers Visual Studio development integration for their products. A Mac computer is still required for compiling, testing, debugging, and distributing the application. More information can be found on Xamarin's website at http://docs.xamarin.com/guides/ios/getting_started/introduction_to_xamarin_ios_for_visual_studio/.

- ▶ **Latest iOS SDK:** To be able to download iOS SDK, a developer must be registered as an Apple developer. iOS SDK, among other things, includes two essential components:
 - ❑ **Xcode:** This is Apple's IDE for developing native applications for iOS and Mac with the *Objective-C* programming language.
 - ❑ **iOS Simulator:** This is an essential program to debug iOS apps on the computer, without the need of a device. Note that there are many iOS features that do not work on the simulator. Hence, a device is needed if an app uses these features.



Both the registration and SDK download are free of charge from Apple's developer portal (<http://developer.apple.com>). If we want to run and debug our apps on the device or distribute them on the App Store, we need to enroll to iOS Developer Program, which requires a subscription fee.

- ▶ **Xamarin Installer:** Xamarin offers all their necessary tools in one installation bundle. This bundle includes the Xamarin.iOS SDK and Xamarin Studio, the IDE for developing iOS applications with C#. A free registration is required for downloading the Xamarin Installer, and it can be found by clicking on the link <http://xamarin.com/download>.

This chapter will also describe how to create our first iPhone project with Xamarin Studio, construct its UI with Xcode, and access the app's user interface from within our code, with the concepts of **Outlets** and **Actions**.

Last, but not least, we will learn how to compile our app, the available compilation options we have, and how to debug on the simulator.

Installing prerequisites

This section gives you information on how to download and install the necessary tools to develop with Xamarin.iOS.

Getting ready

We need to download all the necessary components on our computer. The first thing to do is register as an Apple developer on <http://developer.apple.com>. The registration is free and easy, and it provides access to all the necessary development resources. After the registration is confirmed through e-mail, we can login and download the iOS SDK from the address <https://developer.apple.com/devcenter/ios/index.action#downloads>. At the time of writing, Xcode's latest version is 5.0.1 and iOS SDK's latest version is 7.0.3.

How to do it...

To prepare our computer for iOS development, we need to download and install the necessary components in the following order:

- ▶ **Xcode and iOS SDK:** A login to the Mac App Store is required. You can either search for Xcode in the App Store or click on the **Download Xcode** button in the iOS developer portal's download section. After the download is complete, follow the onscreen instructions to install Xcode. The following screenshot shows Xcode in the Mac App Store:



- **Xamarin Starter Edition:** Download and run the Xamarin Starter Edition from Xamarin's website <http://xamarin.com/download>. Follow the onscreen instructions to install Xamarin Studio and Xamarin.iOS.



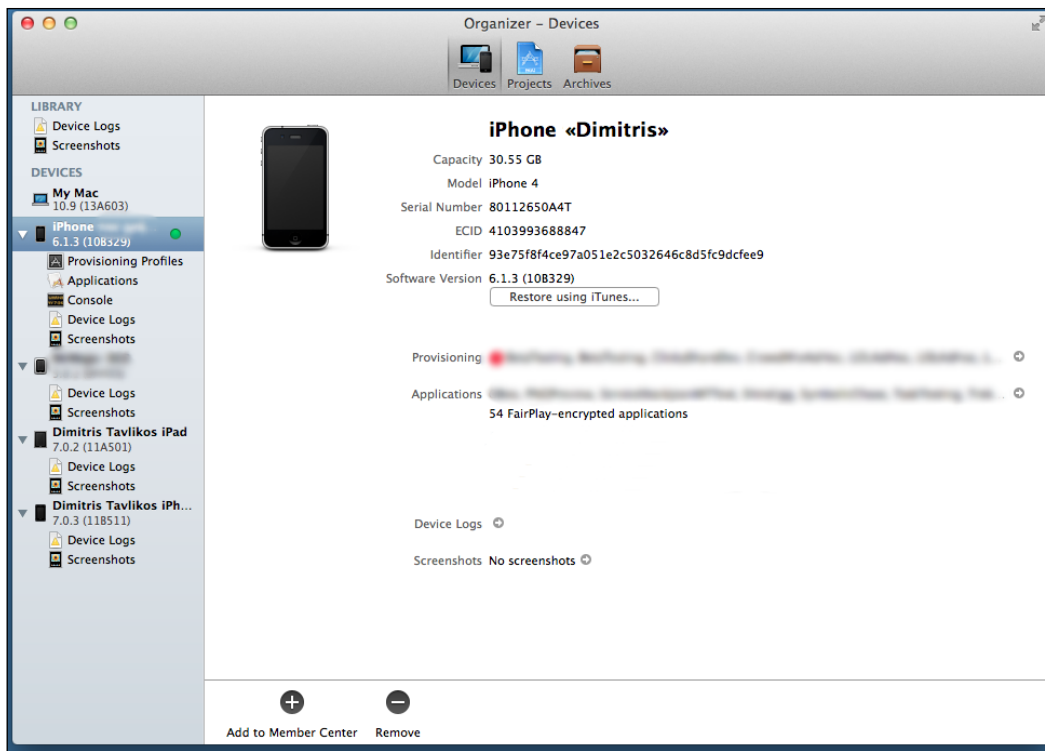
The Xamarin Starter Edition is free, but there are some restrictions, such as a limit on the maximum app bundle size and no Visual Studio support. It does support, however, deploying to a device and to the App Store. At the time of writing, all recipes shown in this book are fully supported by the Starter Edition, except for the *Using WCF services* recipe in *Chapter 6, Web Services*. A Business or Enterprise Edition is needed for WCF support.

How it works...

Now that we have everything ready, let's see what each component is needed for.

Xcode

Xcode is Apple's IDE for developing applications for both iOS and Mac platforms. It is targeted on the Objective-C programming language, which is the main language to program in with the iOS SDK. Since Xamarin.iOS is an SDK for the C# language, one might ask what we would need it for. Apart from providing various tools for debugging iOS apps, Xcode provides us with the **Organizer** window. Shown in the following screenshot, we can use it to view a device's console logs, install and manage the necessary provisioning profiles, and even view the device's crash logs. To open the **Organizer** window, navigate to **Window | Organizer** on the menu bar, or press *Cmd + Shift + 2* on the keyboard.



Interface Builder

The second component is Interface Builder. This is the user interface designer, which was formerly a standalone application. Starting with Xcode 4.0, it is integrated into the IDE. Interface Builder provides all the necessary functionality to construct an application user interface. It is also quite different from what .NET developers are accustomed to.

iOS Simulator

The third component is iOS Simulator. It is exactly what its name suggests: a device simulator that we can use to run our apps on, without the need for an actual device. The most important thing about iOS Simulator is that it has the option of simulating older iOS versions (if they are installed on the computer), both iPhone and iPad interfaces and device orientations. However, the simulator lacks some device features that are dependent on hardware such as the compass or accelerometer. Applications using these features must be tested and debugged on an actual device.

Xamarin.iOS is the SDK that allows .NET developers to develop apps for iOS, using the C# programming language. All APIs available to Objective-C developers are also available to C# developers through Xamarin.iOS. It is not a standalone framework with its own APIs for, say, user interfaces. A Xamarin.iOS programmer can use the same UI elements as an Objective-C programmer, along with the added benefits of C# such as generics, LINQ, and asynchronous programming with `async/await`.

There's more...

Applications developed with Xamarin.iOS have the same chances of making it to the App Store as all other applications developed with the native Objective-C programming language. This means that if an app does not conform to Apple's strict policy about app acceptance, it will fail, whether is written in Objective-C or C#. The Xamarin.iOS team has done a great job in creating an SDK that leaves the developer to worry only about the design and best practice of the code, and nothing else.

Useful links

The following are useful links that you can go through:

- ▶ **Apple iOS developer portal:** <http://developer.apple.com/devcenter/ios/index.action>
- ▶ **Xamarin.iOS:** <http://xamarin.com/ios>
- ▶ **Xamarin installation guide for Mac:** http://docs.xamarin.com/guides/ios/getting_started/installation/mac/
- ▶ **Information about Apple developer tools:** <http://developer.apple.com/technologies/tools/xcode.html>

Updates

Xamarin Studio has a feature for checking available updates. Whenever a program starts, it checks for updates of Xamarin.iOS. It can be turned off, but this is not suggested since it helps with staying up to date with the latest versions. It can be found under **Xamarin Studio | Check for Updates**.

See also

- ▶ The *Compiling an iOS project* and *Debugging our application* recipes
- ▶ The *Preparing our app for the App Store* recipe in *Chapter 14, Deploying*

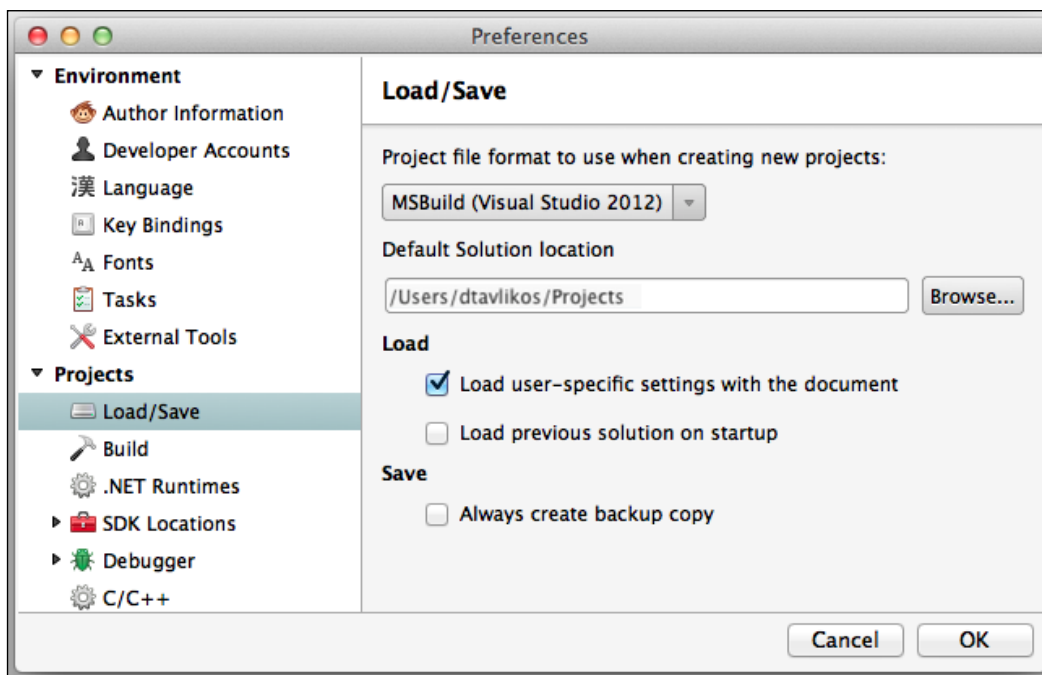
Creating an iOS project with Xamarin Studio

In this recipe, we will discuss how to create our first iOS project with Xamarin Studio.

Getting ready...

Now that we have all the prerequisites installed, we will discuss how to create our first iOS project with Xamarin Studio.

Start Xamarin Studio. It can be found in the Applications folder. Xamarin Studio's default project location is `/Users/{yourusername}/Projects`. If it does not exist on the hard disk, it will be created when we create our first project. If you want to change the folder, go to **Xamarin Studio | Preferences** from the menu bar. Select **Load/Save** in the pane on the left, enter the preferred location for the projects in the **Default Solution location** field, and click on **OK**.

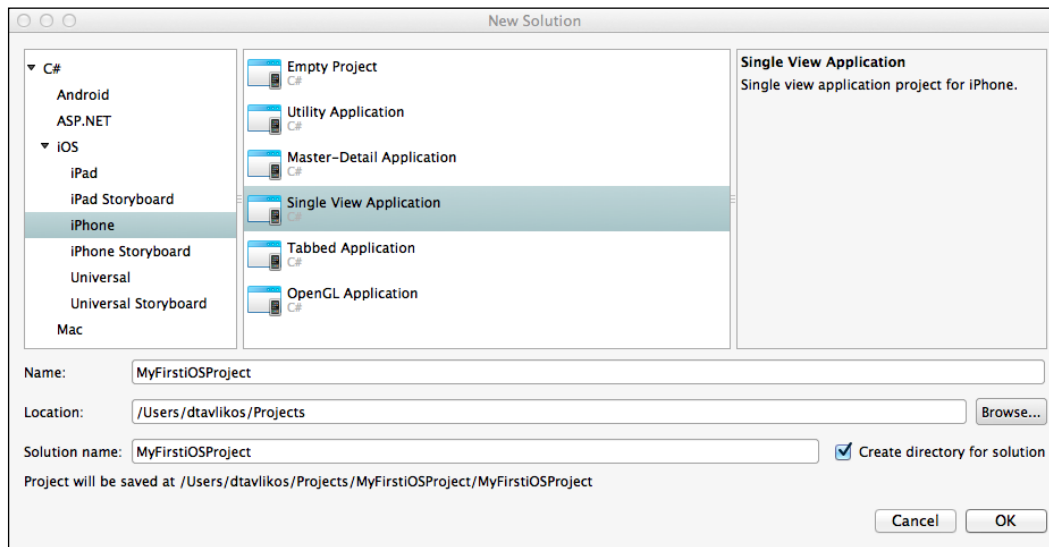


How to do it...

The first thing that is loaded when starting Xamarin Studio is its start page. Perform the following steps to create an iOS project with Xamarin Studio:

1. Navigate to **File | New | Solution...** from the menu bar. A window that provides us with the available project options will appear.

2. In the pane on the left of this window, go to **C# | iOS | iPhone**. The iPhone project templates will be presented on the middle pane.
3. Select **Single View Application**.
4. Finally, enter `MyFirstiOSProject` for **Solution name** and click on **OK**.
The following screenshot displays the **New Solution** window:



That was it. You just created your first iPhone project. You can build and run it; iOS Simulator will start, with a blank light-gray screen nevertheless.



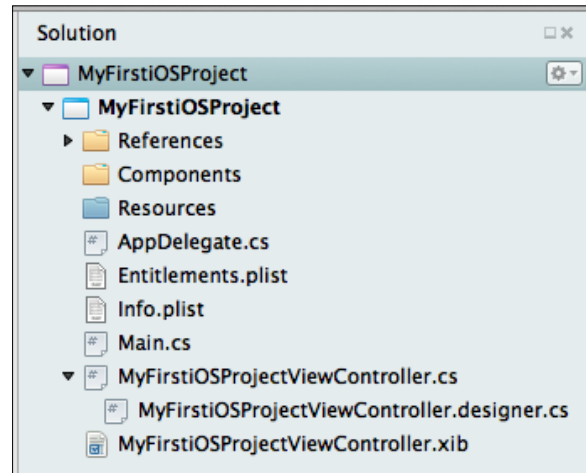
The project templates may be different from the ones shown in the preceding screenshot.

How it works...

Let's see what goes on behind the scenes.

When Xamarin Studio creates a new iOS project, it creates a series of files. The solution files can be viewed in the **Solution** pad on the left side of Xamarin Studio window. If the **Solution** pad is not visible, it can be activated by checking on **View | Pads | Solution** from the menu bar.

The files shown in the following screenshot are the essential files that form an iPhone project:



MyFirstiOSProjectViewController.xib

`MyFirstiOSProjectViewController.xib` is the file that contains the view of the application. XIB files are basically XML files with a specific structure that Xcode can read. The files contain information about the user interface, such as the type of controls it contains, their properties, and Outlets.



If `MyFirstiPhoneProjectViewController.xib`, or any other file with the `.xib` suffix, is double-clicked, Xamarin Studio automatically opens the file in Xcode's Interface Builder.

When we create a new interface with Interface Builder and save it, it is saved in the XIB format.

MyFirstiOSProjectViewController.cs

`MyFirstiOSProjectViewController.cs` is the file that implements the view's functionality. These are the contents of the file when it is created:

```
using System;
using System.Drawing;
using MonoTouch.Foundation;
using MonoTouch.UIKit;

namespace MyFirstiOSProject
{
    public class MyFirstiOSProjectViewController :
        UIViewController
    {
    }
```

```
public MyFirstiOSProjectViewController () :
    base ("MyFirstiOSProjectViewController", null)
{
}

public override void DidReceiveMemoryWarning ()
{
    // Releases the view if it doesn't have a superview.
    base.DidReceiveMemoryWarning ();

    // Release any cached data, images, etc that aren't in
    use.
}

public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    // Perform any additional setup after loading
    the view, typically from a nib.
}
}
```



Xamarin.iOS was formerly known as MonoTouch. For proper code compatibility, the namespaces have not been renamed.

The code in this file contains the class which corresponds to the view that will be loaded, along with some default method that overrides. These methods are the ones that we will use more frequently when we create view controllers. A brief description of each method is listed as follows:

- ▶ **ViewDidLoad:** This method is called when the view of the controller is loaded. This is the method we use to initialize any additional component.
- ▶ **DidReceiveMemoryWarning:** This method is called when the app receives a memory warning. This method is responsible for releasing resources that are not needed at the time.

MyFirstiOSProjectViewController.designer.cs

`MyFirstiOSProjectViewController.designer.cs` is the file that holds our main window's class information in C# code. Xamarin Studio creates one `.designer.cs` file for every XIB that is added in a project. The file is autogenerated every time we save a change in our XIB through Interface Builder. This is taken care of by Xamarin Studio so that the changes we make in our interface are reflected right away in our code. We must not make changes to this file directly, since when the corresponding XIB is saved with Interface Builder, they will be lost. Also, if nothing is saved through Interface Builder and if changes are made to it manually, it will most likely result in a compilation error.

These are the contents of the file when a new project is created:

```
//
// This file has been generated automatically by MonoDevelop to
// store outlets and
// actions made in the Xcode designer. If it is removed, they will
// be lost.
// Manual changes to this file may not be handled correctly.
//
using MonoTouch.Foundation;

namespace MyFirstiOSProject
{
    [Register ("MyFirstiOSProjectViewController")]
    partial class MyFirstiOSProjectViewController
    {
        void ReleaseDesignerOutlets ()
        {
        }
    }
}
```




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.

This file contains the other partial declaration of our `MyFirstiOSProjectViewController` class. It is decorated with the `Register` attribute.

The `Register` attribute is used to expose classes to the underlying Objective-C runtime. The string parameter declares by what name our class will be exposed to the runtime. It can be whatever name we want it to be, but it is a good practice to always set it to our C# class' name. The attribute is used heavily in the internals of Xamarin.iOS, since it is what binds all the native `NSObject` classes with their C# counterparts.



`NSObject` is a root class or base class. It is the equivalent of `System.Object` in the .NET world. The only difference between the two is that all .NET objects inherit from `System.Object`, but most, not all, Objective-C objects inherit from `NSObject` in Objective-C. The C# counterparts of all native objects that inherit from `NSObject` also inherit from its Xamarin.iOS `NSObject` counterpart.

AppDelegate.cs

The `AppDelegate.cs` file contains the `AppDelegate` class. The contents of the file are listed below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using MonoTouch.Foundation;
using MonoTouch.UIKit;

namespace MyFirstiOSProject
{
    // The UIApplicationDelegate for the application. This class
    // is responsible for launching the
    // User Interface of the application, as well as listening
    // (and optionally responding) to
    // application events from iOS.
    [Register ("AppDelegate")]
    public partial class AppDelegate : UIApplicationDelegate
    {
        // class-level declarations
        UIWindow window;
        MyFirstiOSProjectViewController viewController;
        //
        // This method is invoked when the application has loaded
        // and is ready to run. In this
        // method you should instantiate the window, load the UI
        // into it and then make the window
        // visible.
        //
        // You have 17 seconds to return from this method,
        // or iOS will terminate your application.
    }
}
```

```
//
public override bool FinishedLaunching (UIApplication app,
    NSDictionary options)
{
    window = new UIWindow (UIScreen.MainScreen.Bounds);

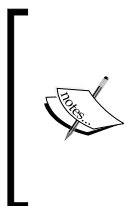
    viewController =
        new MyFirstiOSProjectViewController ();
    window.RootViewController = viewController;
    window.MakeKeyAndVisible ();

    return true;
}
}
```

The first part is familiar to .NET developers and consists of the appropriate `using` directives that import the required namespaces to use. Consider the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using MonoTouch.Foundation;
using MonoTouch.UIKit;
```

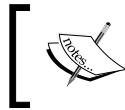
The first three `using` directives allow us to use the specific and familiar namespaces from the .NET world with Xamarin.iOS.



System, System.Collections.Generic, System.Linq: Although the functionality that the three namespaces provide is almost identical to their well-known .NET counterparts, they are included in assemblies specifically created for use with Xamarin.iOS and shipped with it, of course. An assembly compiled with .NET cannot be directly used in a Xamarin.iOS project.

The `MonoTouch.Foundation` namespace is a wrapper around the native Objective-C Foundation Framework, which contains classes that provide basic functionality. These objects' names share the same `NS` prefix that is found in the native Foundation Framework. Some examples are `NSObject`, `NSString`, `NSValue`, and so on. Apart from the `NS`-prefixed objects, the `MonoTouch.Foundation` namespace contains all of the attributes that are used for binding to native objects, such as the `Outlet` and `Register` attributes we saw earlier. The `MonoTouch.UIKit` namespace is a wrapper around the native Objective-C UIKit Framework. As its name suggests, the namespace contains classes, delegates, and events that provide us with interface functionality. Almost all the objects' names share the same `UI` prefix. It should be clear at this point that these two namespaces are essential for all Xamarin.iOS apps, and their objects will be used quite frequently.

The class inherits from the `UIApplicationDelegate` class, qualifying it as our app's delegate object.

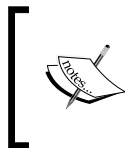


The concept of a delegate object in the Objective-C world is somewhat different from `delegate` in C#. It will be explained in detail in *Chapter 2, User Interface – Views*.

The `AppDelegate` class contains two fields and one method:

```
UIWindow window;  
MyFirstiOSProjectViewController viewController;  
//..  
public override bool FinishedLaunching (UIApplication app,  
    NSDictionary options) {
```

The `UIWindow` object defines the main window of our application, while the `MyFirstiOSProjectViewController` object is the variable that will hold the app's view controller.



An iOS app typically has only one window of type `UIWindow`. `UIWindow` is the first control that is displayed when an app starts, and every subsequent view is hierarchically added below it.

The `FinishedLaunching` method, as its name suggests, is called when the app has completed its initialization process. This is the method where we must present the user interface to the user. The implementation of this method must be lightweight; if it does not return in time from the moment it is called, iOS will terminate the app. This provides faster user interface loading time to the user by preventing developers from performing complex and long-running tasks upon initialization, such as connecting to a web service to receive data. The `app` parameter is the application's `UIApplication` object, which is also accessible through the static property `UIApplication.SharedApplication`. The `options` parameter may or may not contain information about the way the app was launched. We do not need it for now.

The default implementation of the `FinishedLaunching` method for this type of project is as follows:

- The `UIWindow` object is initialized with the size of the screen as follows:

```
window = new UIWindow (UIScreen.MainScreen.Bounds);
```

- The view controller is initialized and set as the window's root view controller as follows:

```
viewController = new MyFirstiPhoneProjectViewController();
window.RootViewController = viewController;
window.MakeKeyAndVisible ();
return true;
```

The window is displayed on the screen with the `window.MakeKeyAndVisible()` call and the method returns. This method must be called inside the `FinishedLaunching` method, otherwise the app's user interface will not be presented as it should be to the user. Last but not least, the `return true` line returns the method by marking its execution completion.

Main.cs

Inside the `Main.cs` file is where the runtime life cycle of the program starts as shown in the following code:

```
namespace MyFirstiOSProject
{
    public class Application
    {
        // This is the main entry point of the application.
        static void Main (string[] args)
        {
            // if you want to use a different Application
            // Delegate class from "AppDelegate"
            // you can specify it here.
            UIApplication.Main (args, null, "AppDelegate");
        }
    }
}
```

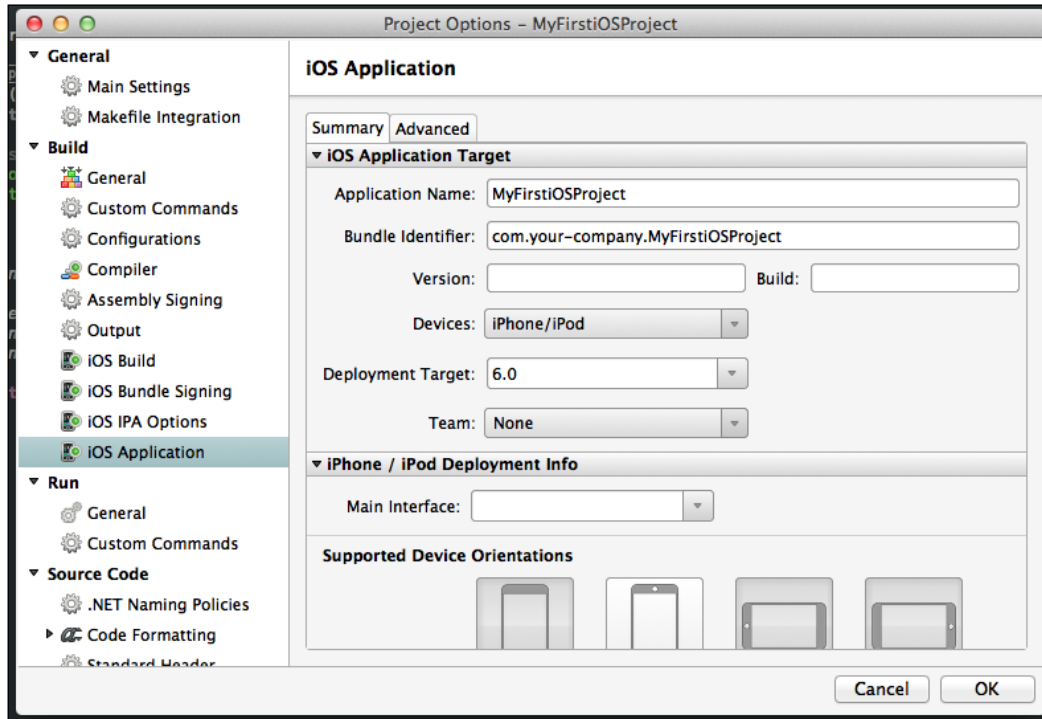
It is much like the following call in a .NET System.Windows.Forms application:

```
Application.Run(new Form1());
```

The `UIApplication.Main` method starts the message loop or run loop that is responsible for dispatching notifications to the app through the `AppDelegate` class with event handlers that we can override. Event handlers such as `FinishedLaunching`, `ReceiveMemoryWarning`, or `DidEnterBackground` are only some of these notifications. Apart from the notification dispatching mechanism, the `UIApplication` object holds a list of all `UIWindow` objects that exist, typically one. An iOS app must have one `UIApplication` object, or a class that inherits from it, and this object must have a corresponding `UIApplicationDelegate` object. This is the `AppDelegate` class implementation we saw earlier.

Info.plist

The `Info.plist` file is basically the app's settings file. It has a simple structure of properties with values that define various settings for an iOS app, such as the orientations it supports, its icons, supported iOS versions, what devices it can be installed on, and so on. If we double-click on this file in Xamarin Studio, it will open in the embedded editor specifically designed for this file. Our file in a new project looks like the following screenshot:



We can also access `Info.plist` through the project's options window under **iOS Application**.

There's more...

Xamarin Studio provides many different project templates for developing iOS apps. Here is a list that describes what each project template is for:

- ▶ **Empty project:** This is an empty project without any views.
- ▶ **Utility application:** This is a special type of iOS app that provides one screen for functionality and, in many cases, another one for configuration.
- ▶ **Master-detail application:** This type of project creates a template that supports navigating through multiple screens. It contains two view controllers.
- ▶ **Single view application:** This template type is the one we used in this recipe.

- ▶ **Tabbed application:** This is a template that adds a tab bar controller, which manages two view controllers in a tab-like interface.
- ▶ **OpenGL application:** This is a template for creating OpenGL-powered applications or games.

These templates are available for the iPhone, iPad, and Universal (both iPhone and iPad) projects. They are also available in Interface Builder's storyboarding app design.



Unless stated, all project templates referring to the iPhone are suitable for the iPod Touch as well.

List of Xamarin.iOS assemblies

Xamarin.iOS-supported assemblies can be found at <http://ios.xamarin.com/Documentation/Assemblies>.

See also

- ▶ The *Creating the UI* and *Accessing the UI with Outlets* recipes
- ▶ The *Adding and customizing views* recipe in *Chapter 2, User Interface – Views*

Interface Builder

In this recipe, we will take a look at Xcode's Interface Builder. Since we cannot use Xcode to write our code, Xamarin Studio provides a transparent way of communicating with Xcode when it comes to user interface files.

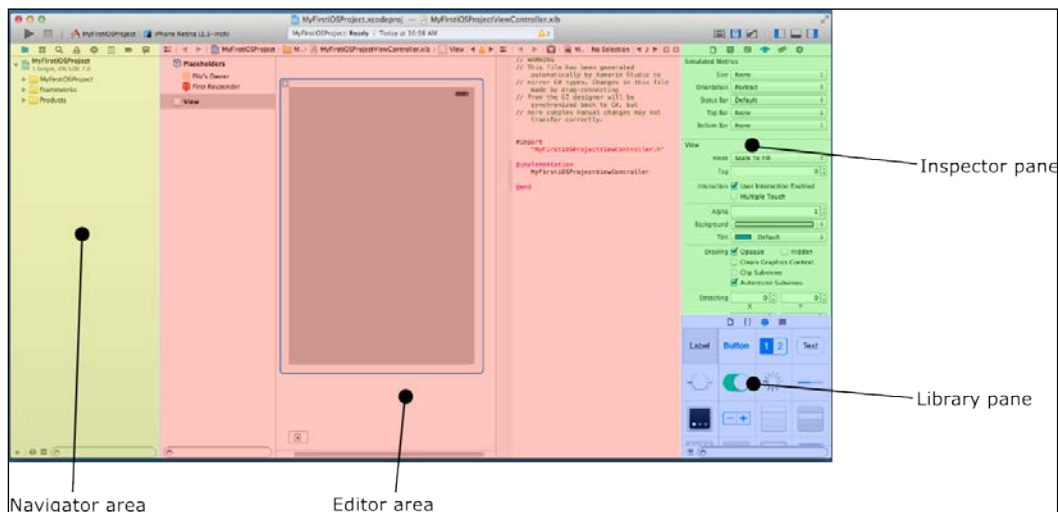
How to do it...

Let's take a look at Interface Builder by performing the following steps:

1. If you have installed the iOS SDK, then you already have Xcode with Interface Builder installed on your computer. Go to Xamarin Studio and open the project `MyFirstiOSProject` we created earlier.
2. In the **Solution** pad on the left, double-click on **MyFirstiOSProjectViewController.xib**. Xamarin Studio starts Xcode with the file loaded in Interface Builder.
3. On the top of the Xcode window in the right side of the toolbar, select the appropriate editor and viewing options, as shown in the following screenshot:



- The following screenshot demonstrates what Interface Builder looks like with an XIB file open:



How it works...

Now that we have loaded Interface Builder with the view controller of our app, let's familiarize ourselves with it.

The user interface designer is directly connected to an Xcode project. When we add an object, Xcode automatically generates code to reflect the change we made. Xamarin Studio takes care of this for us, so that when we double-click on an XIB file, it automatically creates a temporary Xcode project. This allows us to make the changes we want in the user interface. Therefore, we have nothing more to do than just design the user interface for our app.

Interface Builder is divided into three areas. A brief description of each area is given as follows:

- **Navigator area:** In this area, we can see the files included in the Xcode project.
- **Editor area:** This area is where we design the user interface. The editor area is divided into two sections. The one on the left is the designer, and the one on the right is the assistant editor. Inside the assistant editor, we see the underlying Objective-C source code file that corresponds to the selected item in the designer. Although we do not need to edit the Objective-C source, we will need the assistant editor later.
- **Utility area:** This area contains the inspector and library panes. The inspector pane is where we configure each object, and the library pane is where we find the objects.

There's more...

We saw what an XIB file looks like in Interface Builder, but there is more as far as these files are concerned. We mentioned earlier that XIB files are XML files with appropriate information readable by Interface Builder. The thing is that when a compilation is done in a project, the compiler compiles the XIB file converting it to its binary equivalent, the NIB file. Both XIB and NIB files contain the same information. The only difference between them is that XIB files are in a human-readable form while the NIB files are not. For example, when we compile the project we created, the `MyFirstiOSProjectViewController.xib` file will become `MyFirstiOSProjectViewController.nib` in the output folder. Apart from the binary conversion, the compiler also performs a compression on NIB files. So, NIB files will be significantly smaller in size than XIB files.

That's not all about XIB files. The way a developer manages the XIB files in a project is very important in an app's performance and stability. It is better to have many small-sized XIB files, instead of one or two large ones. This is because of the way iOS manages its memory. This can be accomplished by dividing the user interface into many XIB files. It may seem a bit difficult, but as we'll see later in this book, it is actually very easy.

When an app starts, iOS loads the NIB files as a whole in memory, and all the objects in it are instantiated. So, it is a waste of memory to keep objects in NIB files that are not always going to be used. Also, remember that you are developing for a mobile device whose available resources are not a match against that of desktop computers, no matter what its capabilities are.

As of iOS 5, Apple introduced storyboarding, which simplifies user interface design.

More information

You may have noticed that in the **Attributes** tab of the **Inspector** pane, there is a section called **Simulated Metrics**. Options under this section help us see directly what our interface looks like in the designer area with the device's status bar, a toolbar, or a navigation bar. Although these options are saved in the XIB files, they have nothing to do with the actual app at runtime. For example, if we set the **Status Bar** option to **None**, it does not mean that our app will start without a status bar.



Status Bar is the bar that is shown on the top portion of the device's screen, which displays certain information to the user, such as the current time, battery status, and carrier name on the iPhone.

See also

- ▶ The *Creating the UI*, *Accessing the UI with Outlets*, and *Adding Actions to controls* recipes
- ▶ The *Adding and customizing views* recipe in *Chapter 2, User Interface – Views*
- ▶ The *Loading a view with a view controller* recipe in *Chapter 3, User Interface – View Controllers*

Creating the UI

In this recipe, we will learn how to add and manage controls in the user interface.

Getting ready

Let's add a few controls in an interface. Start by creating a new iPhone single view application project in Xamarin Studio. Name the project `ButtonInput`. When it opens, double-click on **ButtonInputViewController.xib** in the **Solution** pad to open it with Interface Builder.

How to do it...

Now that we have a new project, and Interface Builder has opened the `ButtonInputViewController.xib` file, we'll add some controls to it.

Adding a label

Perform the following steps:

1. Go to the **Library** pane and select **Objects** from the drop-down list, if it is not already selected.
2. Select the **Label** object. Drag-and-drop **Label** onto the gray space of the view in the designer, somewhere in the top half.
3. Select and resize the **Label** object from both the left and right sides so that it snaps to the dashed line that will show up when you reach close to the edges of the view.
4. Again, with the **Label** object selected, go to the **Inspector** pane, select the **Attributes** tab, and in the **Layout** section, click on the middle alignment button.

Congratulations, you have just added **Label** in your app's main view!